Universidad Complutense de Madrid

Proyecto de Innovación Educativa en Computación Cuántica

2023-2024

Friday group 1

Authors:

Ruth Macías Asenjo, Fernando Poblet Estévez, Juan Antonio Viñuelas Iniesta
Eduardo Martínez Valdelvira, Miguel González Cuenca, Martín Zapata Ferguson

Session 1: Visual introduction to quantum circuits with Quirk

2. Exercises

2.1. What are the following quantum gate products equivalent to?

(a) $H \cdot H = I$

(b) $X \cdot X = I$

(c) $Y \cdot Y = I$

(d) $Z \cdot Z = I$

(e) $X^{1/2} \cdot X^{1/2} = X$

(f) $X^{1/4} \cdot X^{1/4} = X^{1/2}$

$$X^{1/2} = \pm\frac{1}{2}\begin{pmatrix} 1 \pm i & 1 \mp i \\ 1 \mp i & 1 \pm i \end{pmatrix}$$

However, in QUIRK's implementation

$$X^{1/2} = \frac{1}{2}\begin{pmatrix} 1 + i & 1 - i \\ 1 - i & 1 + i \end{pmatrix}$$

(g) $H \cdot Y \cdot H = -Y$

2.2. Remember that a *global* phase change does not alter a qubit. Build a quantum circuit that, from a single qubit initialized to $|0\rangle$, generates the following 1-qubit states:

(a) $|1\rangle = X|0\rangle$

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

(c) $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = H \cdot X|0\rangle$

(d) $|R\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) = H \cdot X^{1/2}|0\rangle$

(e) $|L\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle) = H \cdot X^{1/2} \cdot X\,|0\rangle$

3. Unitary transformations

3.1. (conceptual, not on QUIRK) Which of the following operations is/are NOT unitary, if any? Not unitary: a), c), d), e), f)

(a) Sorting a list of elements.

A sorted list could have been given in any initial configuration, so this operation is not invertible and thus not unitary.

(b) Applying a certain permutation.

Say we have a quantum state $|\psi\rangle = |\psi\rangle_1 |\psi\rangle_2 ... |\psi\rangle_n$ and a permutation operator which acts as $P|\psi\rangle = |\psi\rangle_{P_1} |\psi\rangle_{P_2} ... |\psi\rangle_{P_n}$, $\{P_i\}_{i=1}^n \in S_n$. It is clear that it preserves the inner product, since

$$\langle\phi| P^\dagger P |\psi\rangle = \langle\phi|_{P_1} |\psi\rangle_{P_1} \langle\phi|_{P_2} |\psi\rangle_{P_2} ... \langle\phi|_{P_n} |\psi\rangle_{P_n} = \langle\phi|_1 |\psi\rangle_1 \langle\phi|_2 |\psi\rangle_2 ... \langle\phi|_n |\psi\rangle_n = \langle\phi|\psi\rangle$$

Therefore the permutation operator P is unitary.

(c) Adding two elements (a and b) keeping none of them.

Any two numbers in a field can add to another number in said field, therefore the operation is not invertible and thus not unitary.

* This question was later changed to: *Adding two elements* $a, b \in \{0, 1, ..., d-1\} \mod d$, *keeping none of them* ($|a\rangle |b\rangle \to |a \overset{d}{\oplus} b\rangle |0\rangle$).

For the same reason as previously the operation is not invertible. Even if a, b belong to some finite list, there are multiple possible combinations which result in the same sum.

(d) Adding two elements (a and b) keeping one and only one of them

This operation may be invertible, however in matrix form:

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a+b \\ b \end{pmatrix}$$

The matrix inverse of this operation is not its adjoint, in fact it is

$$\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

* This question was later changed to: *Adding two elements* $a, b \in \{0, 1, ..., d-1\} \mod d$, *keeping one and only one of them* ($|a\rangle |b\rangle \to |a \overset{d}{\oplus} b\rangle |b\rangle$).

In this case the operation *would* be unitary. If $a, b \in \{0, 1, ..., d-1\}$, clearly also $a \overset{d}{\oplus} b \in \{0, 1, ..., d-1\}$. We can then build a single element Hilbert space $\mathcal{H} = \text{span}\{|i\rangle\}_{i=0}^{d-1}$ and a tensor product space $\mathcal{H}^2 = \mathcal{H} \otimes \mathcal{H}$. In these spaces both $|a\rangle |b\rangle$ and $|a \overset{d}{\oplus} b\rangle |b\rangle$ would be basis kets — the operation would then be norm preserving only if it permuted the

basis kets, i.e. if it didn't map multiple kets to one. This is certainly the case, since $a \overset{d}{\oplus} b \neq a' \overset{d}{\oplus} b \ \forall \ a \neq a'$.

(e) Symmetryzing two states. That is, if we have $|i\rangle\,|j\rangle$, generating $\frac{1}{\sqrt{2}}(|i\rangle\,|j\rangle + |j\rangle\,|i\rangle)$.

The symmetrisation operator $P_S = \frac{1}{2}\sum_{\pi\in S_2}\pi$ is a projector to the subspace of symmetric states and thus not invertible. Originally $|i\rangle\,|j\rangle$ could have had an unknown antisymmetric component which symmetrisation would eliminate.

(f) Given a list of elements, generating a new list of elements that verify certain properties without keeping a copy of the original list. For instance, if the list is numerical, keeping only numbers that are greater that some given number.

In the very example provided, the original list could have had any number below the cutoff, so the operation is not in general invertible.

3.2. Check on QUIRK that the Pauli-X, Y and Z operations, represented on the computational basis $\{|0\rangle, |1\rangle\}$ by the Pauli matrices σ_x, σ_y and σ_z, are unitary. Compute their inverses and check that they are equal to their transpose conjugates.

We can easily check by hand that

$$\sigma_x = \sigma_x^{-1} = \sigma_x^\dagger = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\sigma_y = \sigma_y^{-1} = \sigma_y^\dagger = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\sigma_z = \sigma_z^{-1} = \sigma_z^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

4. Multi-qubit quantum gates

4.1. Control gates: `CNOT` and others

(a) Hover the mouse pointer over the two default qubit lines and check that both qubits are on the $|0\rangle$ state.

(b) Drop a `NOT` gate over the second line and annotate the change on the corresponding Bloch sphere.

`NOT`$|0\rangle = |1\rangle$

(c) Drop a `control` gate (small black circle) over the first line and on the same horizontal position that the first Pauli-X (`NOT`) gate. A new vertical segment joining the `control` and the `NOT` gate should appear. Annotate any change on any of the Bloch spheres.

`CNOT`$|00\rangle = |00\rangle$

A controlled gate only takes effect if the control qubit has a $|1\rangle$ component.

(d) Change the state of the first qubit to $|1\rangle$ by clicking on the $|0\rangle$ symbol on the left of the corresponding qubit line. Annotate any changes on the Bloch spheres.

`CNOT`$|01\rangle = |11\rangle$

The controlled gate produces a bit flip.

(e) Repeat the experiment using an `anti-control` (small white circle) instead of the `control`.

The `anti-control` does the opposite, it takes effect if the control qubit has a $|0\rangle$ component.

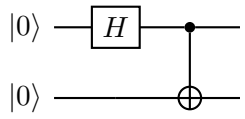
(f) Repeat the experiment using a Hadamard gate instead of the `NOT` one.

The control logic is the same as before, the only difference is $H|0\rangle = |+\rangle$.

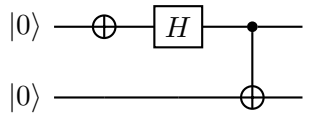
4.2. Building entangled Bell states

Using a Hadamard gate and a `CNOT` gate whose control is applied to the output of the Hadamard gate, build a quantum circuit that turns a 2-qubit initial state $|0\rangle \otimes |0\rangle$ to the following Bell states:

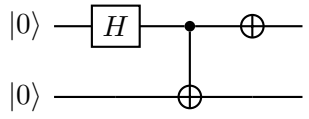
(a) $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle)$



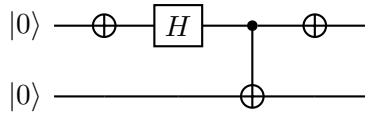
(b) $|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle - |1\rangle \otimes |1\rangle)$



(c) $|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle)$



(d) $|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle - |1\rangle \otimes |0\rangle)$



4.3. No qubit cloning, even by the CNOT gate

4.3.1. The 1-qubit state $|-\rangle$ is defined as above in 2b by

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

Using the basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ to describe a 2-qubit quantum state, answer the following questions:

(a) What is the 2-qubit state if each qubit has been initialized to $|-\rangle$?

The state is $|-\rangle \otimes |-\rangle = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle)$.

(b) The first qubit is initialized to $|-\rangle$ and the second, to $|0\rangle$. What is the 2-qubit state

if a `CNOT` gate has been applied targeting the second qubit, the first one being the control?

The state is $\texttt{CNOT}(|0\rangle \otimes |-\rangle) = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$.

(c) Can you use a `CNOT` gate for breaking the no-clone theorem? Please, justify your answer on the previous questions of this exercise.

At the very least, as this counterexample shows, we cannot use the `CNOT` gate to clone states as $\texttt{CNOT}(|0\rangle \otimes |\psi\rangle) = |\psi\rangle \otimes |\psi\rangle$.

4.3.2. Using Hadamard gates, prepare 2 quantum registers, each one composed of 2 qubits. The states of this registers will be:

- Register A: $|-\rangle \otimes |-\rangle = (HX|0\rangle) \otimes (HX|0\rangle)$
- Register B: the first qubit is initialized to $|-\rangle = HX|0\rangle$. The second one is initialized by applying a `CNOT` gate to the quantum state $|0\rangle$. The control of this `CNOT` gate is the first qubit, previously initialized to $|-\rangle$.

Operator X is the Pauli-X operator (`NOT` gate). Answer the following questions:

(a) Compute the probability of measuring a $|1\rangle$ state in each of the four qubits.

The probability to measure a $|1\rangle$ in each qubit is 50%.

(b) Is there any difference among the four Bloch spheres?

The Bloch spheres for each qubit are very different, we can only obtain states of the form $|00xx\rangle$ or $|11xx\rangle$, meaning the qubits in register B can only be measured in the same state. The first two are in a superposition of all possible states.

(c) Include an additional Hadamard gate in each qubit before the measurement. Taking into account that $H^2 = I$, do you see anything strange? Can a `CNOT` gate alter the *control* qubit? Does the concept of individual qubit make any sense in the presence of quantum entanglement effects?

Given that $H^2 = I$, the first two qubits now become $X \otimes X|00\rangle = |11\rangle$, whereas the second two become the Bell state $|\Psi^+\rangle$ (as in exercise 4.2). A `CNOT` gate *does* in general alter the control qubit — in fact, it only doesn't when the control qubit is either in the $|0\rangle$ or in the $|1\rangle$ state. The notion of individual qubit breaks down when logic gates apply operations to all qubits and they become entangled (their measurement outcomes become correlated).

(d) Using the Kronecker (tensor) product and the matrix formalism, compute by hand (or by a matricial language like Octave or Matlab) the effect of 2 Hadamard gates, $H \otimes H$, on a Bell state, $\left|\Phi^-\right\rangle = \dfrac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle - |1\rangle \otimes |1\rangle)$. Can you explain the QUIRK results?

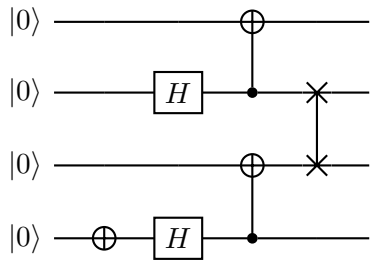
The result is $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. In QUIRK we can simply read out the results of the applied gate in the amplitude display.

4.4. Quantum entanglement of distant qubits via SWAP gates

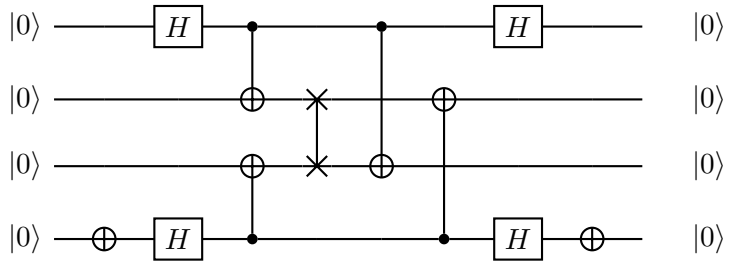
On the memory of current state-of-the-art quantum computers the qubits are coupled only to their near neighbours. In order to entangle distant qubits, the 2-qubit swapping gate (`SWAP`) is used. Such a gate is implemented on QUIRK. Please, use it to solve the following

exercise:

(a) Build a 4-qubit quantum circuit. It should generate 2 Bell pairs, $|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ and $|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. These pairs are formed by qubits $(1,3)$ for $|\Psi^+\rangle$ and by $(2,4)$ for $|\Psi^-\rangle$. As a restriction, consider that the `CNOT` gates can only be applied to nearest-neighbour qubits. That is, they can entangle qubits $(1,2)$, $(2,3)$ or $(3,4)$. You should use a single `SWAP` gate, as it would be done on an actual quantum computer.



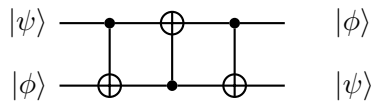
(b) Check that qubits $(1,3)$ and $(2,4)$ are entangled, so that they form Bell pairs. In order to prove this, use the fact that $UU^\dagger = I$. Here, $U = \texttt{CNOT}\cdot(H \otimes I)$ and $U^\dagger = (H \otimes I)\cdot \texttt{CNOT}$, since $H^\dagger = H^{-1} = H$ and $\texttt{CNOT}^\dagger = \texttt{CNOT}^{-1} = \texttt{CNOT}$. Hence, if you reverse the quantum circuit that generated the Bell pairs $|\Psi^-\rangle\,|\Psi^+\rangle$, a $|0000\rangle$ state should be recovered. Check this with the pairs $(1,3)$ and $(2,4)$. Repeat the process with qubits $(1,2)$ and $(3,4)$, that should not be entangled. Repeat the process, changing the initial state of each of the 4 qubits.



By applying the circuit which creates the Bell pairs in reversed order (swapping the corresponding qubits) we recover $|0000\rangle$ as expected.

(c) Build a gate that swaps the values of 2 qubits with 3 `CNOT` gates. Compare its action over the 4 possible input values with the `SWAP` gate.

The `SWAP` gate may be decomposed as follows:



This composite gate has the same effect as a `SWAP` gate. In the $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ basis, the matrices of these gates read

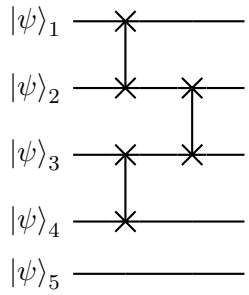
$$\texttt{CNOT}_{1\to 2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\texttt{CNOT}_{2\to1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$\texttt{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrix product $\texttt{CNOT}_{1\to2}\texttt{CNOT}_{2\to1}\texttt{CNOT}_{1\to2}$ indeed yields a `SWAP` gate. Its action on the basis elements is given by its column vectors.

(d) Can you implement the permutation of 5 qubits $(1, 2, 3, 4, 5) \to (2, 4, 1, 3, 5)$ in Quirk?



Session 2: An introduction to Qiskit and to IBM's interface

[59]:
```
# A classical simulator of quantum circuits
from qiskit_aer import AerSimulator

# Class QuantumCircuit, with methods (functions) to define quantum circuits
from qiskit import QuantumCircuit

# A function yielding a histogram with the probabilities of each of the␣
 ↪eigenstates taken by the measured qubits
from qiskit.visualization import plot_histogram

# Class Statvector, to retrieve the state of the circuit
from qiskit.quantum_info import Statevector

# Class PhaseGate, essential for quantum Fourier transform
from qiskit.circuit.library import PhaseGate

# Other usual imports
import numpy as np
import matplotlib.pyplot as plt
```

Sections 1 and 2 walk us through the basics of Qiskit.

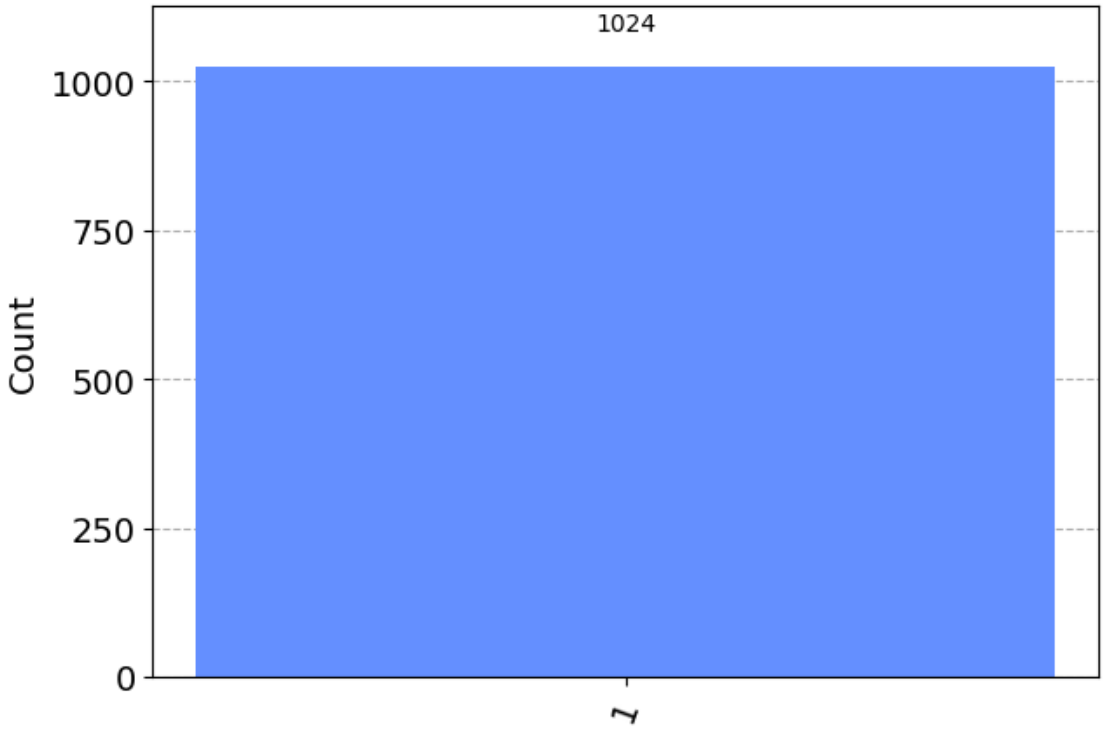
3. Executing circuits

Question: Why does the histogram have only one bar corresponding to |1> ?

Answer: Because the qubit is initialized to the state $|1\rangle$, it has no component along $|0\rangle$.

[60]:
```
sim = AerSimulator()
qc11 = QuantumCircuit(1,1) # circuit with 1 qubit y 1 classical bit
state_1 = [0,1] # this would be |1>, trivially normalized
qc11.initialize(state_1,0)
qc11.measure(0,0)
resultmeasure = sim.run(qc11).result() # Execute the circuite qc11 in the␣
 ↪simulator
counts = resultmeasure.get_counts() # Results of the measurements
plot_histogram(counts) # Plot a histogram with that variable
```

[60]:



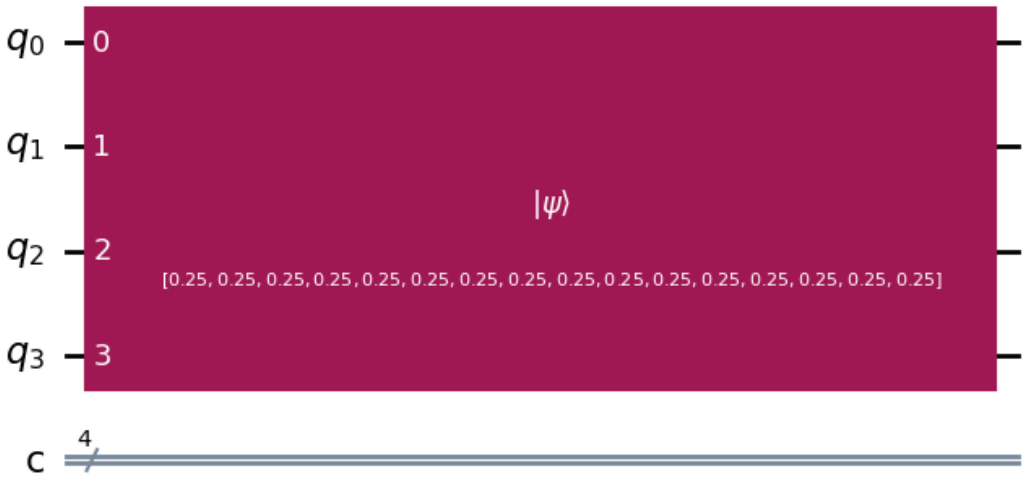
4. Generation of truly random numbers with a quantum computer

Question: Can you write in a piece of paper the initial state in the second line of the next paragraph?

Answer: Let $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ in the computational, single-qubit basis — the initial state is $|+\rangle \otimes |+\rangle \otimes |+\rangle \otimes |+\rangle$. In the $\{|n\rangle\}_{n=0}^{15}$ basis, where $|n\rangle$ is the binary representation of n, taking the digits 0 or 1 to be qubits $|0\rangle$, $|1\rangle$ tensored with each other, $|\text{init}\rangle = \frac{1}{4}\sum_{n=0}^{15}|n\rangle$. Indeed, this is what the `draw` method shows, an equal superposition of states, which is what we would need to simulate a uniform probability distribution.

[61]:
```
qrng = QuantumCircuit(4,4)
init_state = np.kron([1,1],np.kron([1,1],np.kron([1,1],[1,1]))) # Kron is a␣
 ↪Python command for Cartesian (or Kronecker) products
init_state_normal = init_state/np.linalg.norm(init_state)
qrng.initialize(init_state_normal,[0,1,2,3])
qrng.draw(output='mpl')
```

[61]:



We may encapsulate random number generation in the following function:

[62]:
```
def QRNG(N_qubits): #a quantum random number generator
    qrng = QuantumCircuit(N_qubits)
    qrng.h(range(N_qubits))
    qrng.measure_all()
    result = sim.run(qrng, shots=1).result()
    counts = result.get_counts()
    return int(*counts, 2)
```

Question: run the circuit a few times and write down the number obtained

Answer: After running it a few times we obtained the following list of numbers: `14, 3, 5, 9, 8, 12, 1, 10, 6, 11, 2, 6, 0, 10, 7, 13, 6, 14, 5, 1, 0, 14, 4`

[63]:
```
QRNG(4)
```

[63]:
```
9
```

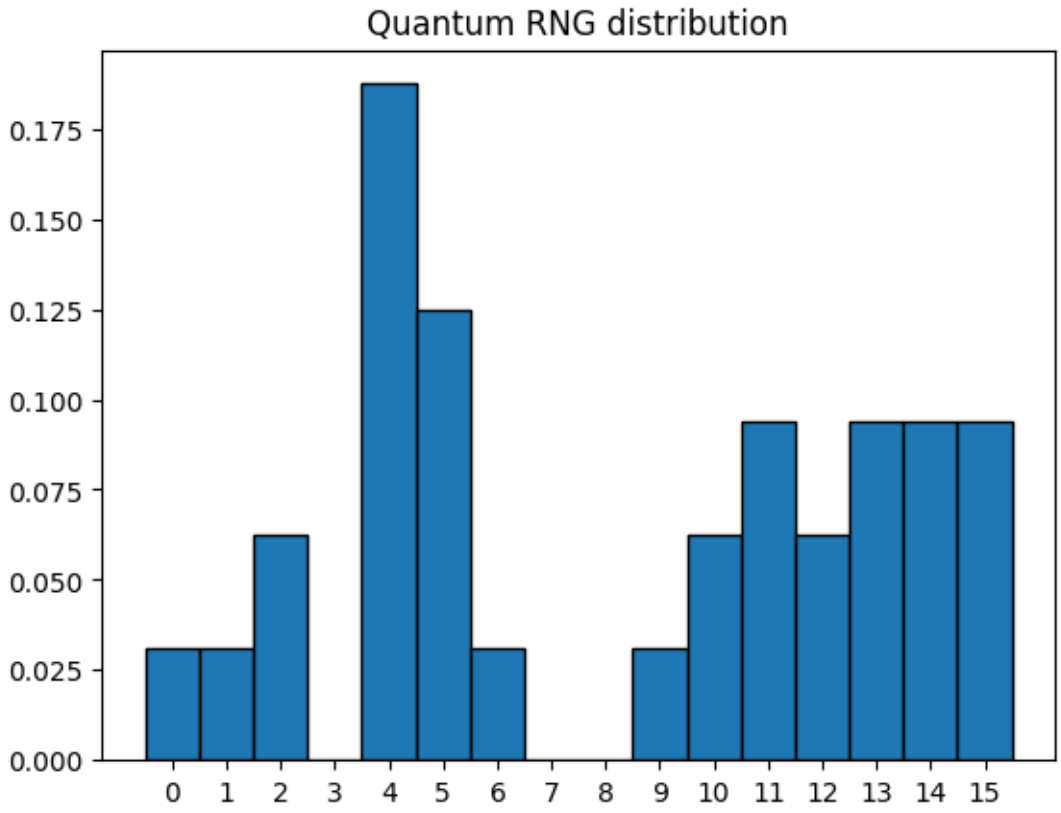
Question: Write a loop that generates 32 random numbers and plot a histogram with their distribution.

Answer:

[64]:
```
N_qubits = 4
N = 32
distrib = [QRNG(N_qubits) for _ in range(N)]

dim = 2**N_qubits
```

```
plt.hist(distrib, bins=range(dim+1), density=True, align='left',␣
  ↪edgecolor='black', linewidth=1)
plt.xticks(ticks=range(dim), labels=range(dim))
plt.title('Quantum RNG distribution')
```

[64]: `Text(0.5, 1.0, 'Quantum RNG distribution')`



Question: Write down in a piece of paper all the methods that you have learned here

Answer: We have learnt the following methods:

Applied to a `QuantumCircuit`

- `draw`: Plots a diagram of the circuit.
- `initialize`: Given a normalised statevector of the appropriate dimension, prepares the circuit or subset of the circuit in that state.
- `measure`: Collapses a particular qubit and stores result in a particular cbit.
- `measure_all`: Adds a cbit for each qubit in the circuit and measures all.

Applied to an `AerSimulator`

- `run`: Runs `QuantumCircuit` on the Aer backend and returns asynchronous simulation job.

Applied to an `AerJob`

- `result`: Awaits job result.

Applied to a `Result`

- `get_counts`: Gets the histogram data of a quantum simulation.

5. Quantum Fourier Transform

The discrete Fourier transform (DFT) changes the elements of a computational basis B to the Fourier basis $F(B)$ according to the formula

$$|j\rangle_B \to |j\rangle_{F(B)} = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle_B$$

It is worth noting that the elements of the computational basis in a 2^n dimensional Hilbert space may be labeled by the number j, or by its representation as an n-bit binary string,

$$j \equiv j_1 j_2 ... j_n \equiv \sum_{\ell=1}^{n} j_\ell 2^{n-\ell}$$

, i.e. we can choose the basis states to be the tensor products of the single-qubit basis states $|j_\ell\rangle_B$, $j_\ell \in \{0, 1\}$.

We call the application of the DFT to a quantum state the quantum Fourier transform (QFT), and it has a computationally efficient alternative formula:

$$QFT_n|j\rangle_B = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle_B = \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi ijk2^{-n}} |k\rangle_B =$$

$$= \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} ... \sum_{k_n=0}^{1} e^{2\pi ij \sum_{\ell=1}^{n} k_\ell 2^{-\ell}} |k_1 k_2 ... k_n\rangle_B =$$

$$= \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} e^{2\pi ijk_1 2^{-1}} |k_1\rangle_B \sum_{k_2=0}^{1} e^{2\pi ijk_2 2^{-2}} |k_2\rangle_B ... \sum_{k_n=0}^{1} e^{2\pi ijk_n 2^{-n}} |k_n\rangle_B = \frac{1}{2^{n/2}} \bigotimes_{m=1}^{n} \sum_{k_m=0}^{1} e^{2\pi ijk_m 2^{-m}} |k_\ell\rangle_B =$$

$$= \frac{1}{2^{n/2}} \bigotimes_{m=1}^{n} (|0\rangle + e^{2\pi ij2^{-m}} |1\rangle) = \frac{1}{2^{n/2}} \bigotimes_{m=1}^{n} \left(|0\rangle + e^{2\pi i \sum_{\ell=1}^{n} j_\ell 2^{n-\ell-m}} |1\rangle \right)$$

Since integer rotations in the complex plane don't add any phase, the expression for the QFT finally simplifies to

$$QFT_n|j\rangle_B = \frac{1}{2^{n/2}} \bigotimes_{m=1}^{n} \left(|0\rangle + e^{2\pi i \sum_{\ell=1}^{m} j_{n+\ell-m} 2^{-\ell}} |1\rangle \right)$$

In the exercise notebook we're walked through programming the QFT:

The operation

$$\alpha|0\rangle + \beta|1\rangle \rightarrow \alpha|0\rangle + \beta e^{2\pi i \ / \ 2^k}|1\rangle$$

Can be represented in matrix form as

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix},$$

while one of the basic transformations in the Qiskit package is *PhaseGate*, with matrix representation

$$\text{PhaseGate}(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}.$$

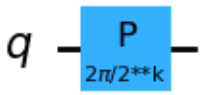
Hence,

Question 5.1. Check that the following function implements the R_k transformation:

Answer:

[65]:
```
def R(k):
    return PhaseGate((2.*np.pi)/pow(2,k))

# Indeed, we can plot R(k)   k with a parametrised circuit.
# We see the transformation is implemented as desired.
from qiskit.circuit import Parameter
k = Parameter('k')
ans = QuantumCircuit(1)
ans.append(R(k),[0])
ans.draw(output='mpl')
```

[65]:



We can try this gate and compare it to what we should get for a couple of values of k

[66]:
```
maxK = 6
result = [0] * maxK
expected = [0] * maxK
```

```
for k in range(maxK):
  # Rk circuit
  circ1 = QuantumCircuit(1)
  circ1.x(0) # Initialise to |1>
  circ1.append(R(k), [0]) # Apply Rk
  psi = Statevector(circ1)
  result[k] = psi[1]
  # Expected phase
  expected[k] = np.exp(2*np.pi*(1j) / 2**k)

print(
f'''
What we got        {result}
What we should get {expected}''')
```

```
What we got        [(1-2.4492935982947064e-16j), (-1+1.2246467991473532e-16j),
(6.123233995736766e-17+1j), (0.7071067811865476+0.7071067811865476j),
(0.9238795325112867+0.3826834323650898j),
(0.9807852804032304+0.19509032201612825j)]
What we should get [(1-2.4492935982947064e-16j), (-1+1.2246467991473532e-16j),
(6.123233995736766e-17+1j), (0.7071067811865476+0.7071067811865476j),
(0.9238795325112867+0.3826834323650898j),
(0.9807852804032304+0.19509032201612825j)]
```

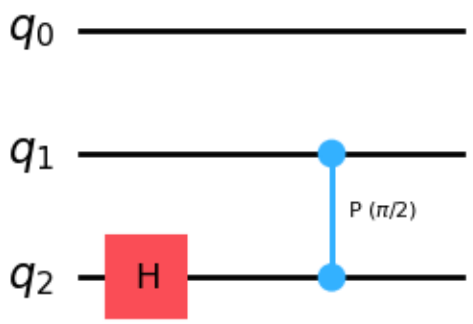
We can appreciate the R_k gate performs the previously described transformation perfectly, although there may be some precision error in both calculations.

In sections 5.1-5.2 we're walked through programming a 3-register QFT. At the step at which we accomplish

$$|j_1 j_2 j_3\rangle \rightarrow |\psi_1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i(j_1/2+j_2/4)}|1\rangle\right) \otimes |j_2 j_3\rangle.$$

[67]:
```
qft_circuit = QuantumCircuit(3, name = 'QFT')
qft_circuit.h(2)
qft_circuit.draw()
qft_circuit.append(R(2).control(),[2,1])
qft_circuit.draw(output='mpl')
```

[67]:



Question 5.2 Write the state resulting from applying that transformation to the initial ket $|010\rangle$. Check, with the Statevector class that the circuit generates the correct output.

Answer: We may compute the resulting state by hand. It is $\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \otimes |10\rangle$ or, in the binary basis, $\frac{1}{\sqrt{2}}(|2\rangle + i|6\rangle)$.

[68]:
```
init_state = np.zeros(8)
init_state[2] = 1
qc = QuantumCircuit(3)
qc.initialize(init_state, range(3))
qc.append(qft_circuit, range(3))
phians = Statevector(qc)

# By printing the result, this checks out
phians
```

```
Statevector([0.00000000e+00+0.j        , 0.00000000e+00+0.j        ,
             7.07106781e-01+0.j        , 0.00000000e+00+0.j        ,
             0.00000000e+00+0.j        , 0.00000000e+00+0.j        ,
             4.32978028e-17+0.70710678j, 0.00000000e+00+0.j        ],
            dims=(2, 2, 2))
```

5.3 QFT full exercise

Please implement now the Quantum Fourier Transform as a gate acting over 5 qubits. Apply this gate to the following states.

$$|\phi_1\rangle = \frac{1}{2^{5/2}} (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle - |1\rangle)$$

$$|\phi_2\rangle = \frac{1}{2^{5/2}} (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle - |1\rangle) (|0\rangle + |1\rangle)$$

$$|\phi_3\rangle = \frac{1}{2^{5/2}} (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle - |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle)$$

$$|\phi_4\rangle = \frac{1}{2^{5/2}} (|0\rangle + |1\rangle) (|0\rangle - |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle) (|0\rangle + |1\rangle)$$

Briefly comment on the results.

Answer: A QFT circuit implements the following transformation:

$$|j_1 j_2 \cdots j_n\rangle_B \rightarrow \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i \frac{j_n}{2}} |1\rangle\right) \left(|0\rangle + e^{2\pi i (\frac{j_{n-1}}{2} + \frac{j_n}{4})} |1\rangle\right) \ldots \left(|0\rangle + e^{2\pi i (\frac{j_1}{2} + \ldots + \frac{j_n}{2^n})} |1\rangle\right)$$

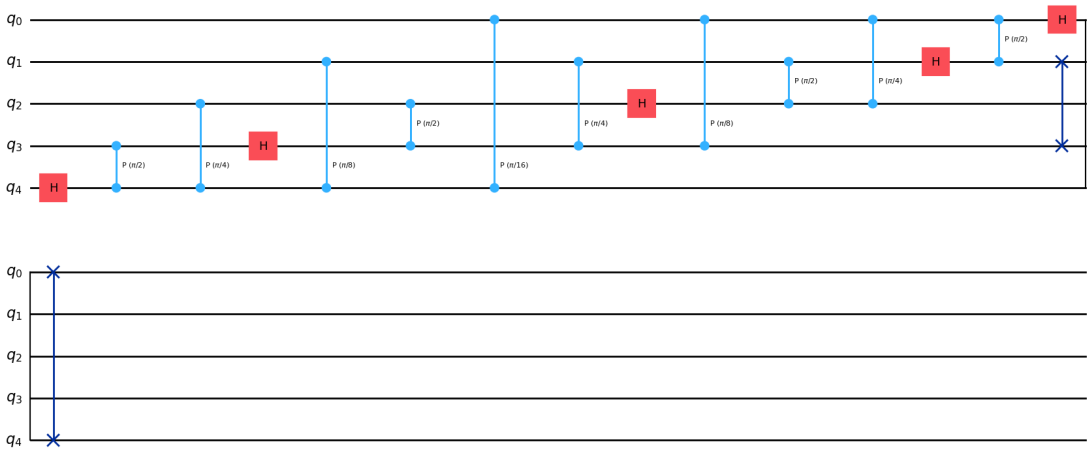
As we can see, the value of qubit 1 (j_n) appears in every term (and j_{n-1} appears in every term up to the penultimate, so on and so forth). This implies we have to act on the least significant bit last, so as to not modify j_n while we still need its value (likewise with j_{n-1}, etc.).

On the other hand, the least significant half of the transformed bits require information from bits more significant than them, so we have to apply the QFT from least significant bit to most significant bit.

The way to solve this conundrum is to store the transformed qubits in reverse order, and to afterwards apply a series of swaps to recover the proper ordering. We can apply the required phases with Hadamard and controlled R_k gates.

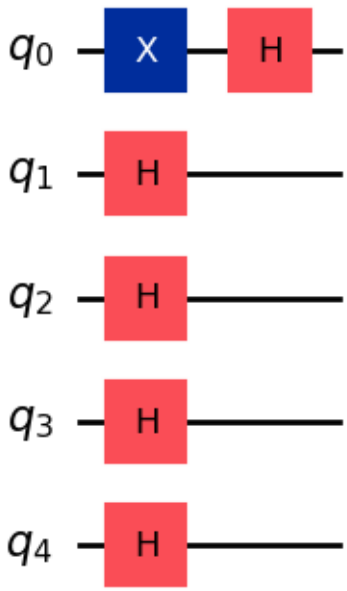
[69]:
```
# First, it is actually easier and more convenient to implement an N register␣
↪QFT

def QFT(N):
    qc = QuantumCircuit(N)
    for i in np.arange(N-1, -1, -1):
        qc.h(i)
        for j in range(i-1, -1, -1):
            qc.append(R(i-j+1).control(),[i,j])
    for i in range(N//2):
        qc.swap(i, N-i-1)
    return qc

QFT(5).draw(output='mpl')
```

[69]:



[70]:
```
# Next, let's notice the kets are
# |phi1> = |+>|+>|+>|+>|->
# |phi2> = |+>|+>|+>|->|+>
# |phi3> = |+>|+>|->|+>|+>
# |phi4> = |+>|->|+>|+>|+>
# So it will be easiest to prepare the initial states with Hadamard gates as follows:


def phi(n):
    qc = QuantumCircuit(5)
    qc.x(n)
    for i in range(5):
        qc.h(i)
    return qc

phi(0).draw(output='mpl')
```

[70]:



[71]:
```
# Next we must create a 5-register QFT gate
qft5 = QFT(5).to_gate()
# And apply it to each prepared state
```

```
prec = 2 # Decimal precision
phihat_q = np.empty((4, 32), dtype=np.cdouble)
ket_q = []
for k in range(4):
    qc = phi(k)
    # Quantum Fourier transform
    qc.append(qft5, range(5))
    phihat_q[k] = Statevector(qc)
    # Format ket as a linear combination of basis vectors (string)
    idx = np.where(np.abs(phihat_q[k]) > 10**-prec)
    ket_q.append('+'.join([f'({phihat_q[k][j]:.{prec}f})|{j}>' for j in␣
 ↪idx[0]]))

# Print out Fourier transformed kets
ket_q
```

[71]:
```
['(1.00+0.00j)|16>',
 '(0.50+0.50j)|8>+(0.50-0.50j)|24>',
 '(0.25+0.60j)|4>+(0.25+0.10j)|12>+(0.25-0.10j)|20>+(0.25-0.60j)|28>',
 '(0.12+0.63j)|2>+(0.12+0.19j)|6>+(0.12+0.08j)|10>+(0.12+0.02j)|14>+(0.12-
0.02j)|18>+(0.12-0.08j)|22>+(0.12-0.19j)|26>+(0.12-0.63j)|30>']
```

Let QFT_5 be the Quantum Fourier transform of 5 qubits, we obtained the following results:

- $QFT_5|\phi_1\rangle = |16\rangle$
- $QFT_5|\phi_2\rangle = \frac{1}{2}(1+i)|8\rangle + \frac{1}{2}(1-i)|24\rangle$
- $QFT_5|\phi_3\rangle = (0.25+0.60i)|4\rangle + (0.25+0.10i)|12\rangle + (0.25-0.10i)|20\rangle + (0.25-0.60i)|28\rangle$
- $QFT_5|\phi_4\rangle = (0.12+0.63i)|2\rangle + (0.12+0.19i)|6\rangle + (0.12+0.08i)|10\rangle + (0.12+0.02i)|14\rangle + (0.12-0.02i)|18\rangle + (0.12-0.08i)|22\rangle + (0.12-0.19i)|26\rangle + (0.12-0.63i)|30\rangle$

The QFT is so efficient because the number of quantum gates it requires for n qubits scales like $O(n^2)$, whereas the dimension of the Hilbert space scales like $N = 2^n$. This can be compared to best DFT algorithm, the Fast Fourier Transform (FFT), which has complexity $O(N \log N) = O(n2^n)$ — that is, the QFT has an exponential advantage over the classical DFT.

We could also check whether we obtain the same result via classical computing methods (bearing in mind that the convention for Fourier / inverse Fourier transform in signal processing is opposite that of quantum computing).

[72]:
```
phihat_c = np.empty((4, 32), dtype=np.cdouble)
ket_c = []
for k in range(4):
    qc = phi(k)
    # Classical Fourier transform
    phihat_c[k] = np.fft.ifft(Statevector(qc), norm='ortho')
    # Format ket as a linear combination of basis vectors (string)
    idx = np.where(np.abs(phihat_c[k]) > 10**-prec)
```

```
    ket_c.append('+'.join([f'({phihat_c[k][j]:.{prec}f})|{j}>' for j in␣
 ↪idx[0]]))

# Print out Fourier transformed kets
ket_c
```

[72]:
```
['(1.00+0.00j)|16>',
 '(0.50+0.50j)|8>+(0.50-0.50j)|24>',
 '(0.25+0.60j)|4>+(0.25+0.10j)|12>+(0.25-0.10j)|20>+(0.25-0.60j)|28>',
 '(0.12+0.63j)|2>+(0.12+0.19j)|6>+(0.12+0.08j)|10>+(0.12+0.02j)|14>+(0.12-
0.02j)|18>+(0.12-0.08j)|22>+(0.12-0.19j)|26>+(0.12-0.63j)|30>']
```

Looks promising! Just so our eyes don't deceive us, let's compare both calculations programatically.

[73]:
```
np.max(np.abs(phihat_q - phihat_c))
```

[73]:
```
3.3306690738754696e-16
```

As we can see, the discrepancy between both calculations is at most on the order of 10^{-16}, which could be attributed to machine error.

Session 3: Execution of real quantum circuits on IBM's computers

[2]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime import QiskitRuntimeService, SamplerV2, EstimatorV2
from qiskit.quantum_info import SparsePauliOp, Operator

# Other imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In the lab session we used premium accounts to fast forward over the long queue list of IBM's quantum machines. However at home we can use our free accounts to schedule calculations.

[3]:
```
# Recover previous session information
df = pd.read_csv('session.csv')

# Loading account
service = QiskitRuntimeService(channel='ibm_quantum', token=df.token[0])

# Simulation machine
backend_S = AerSimulator()
# In the lab we used backend_S = service.backend('ibmq_qasm_simulator'),
# however cloud simulators have been deprecated and will be removed on 15 May␣
↪2024
```

[ ]:
```
# Quantum machine
backend_Q = service.least_busy(operational=True, simulator=False,␣
↪min_num_qubits=20)

# Save backend
df.backend_Q = backend_Q.name
df.to_csv('session.csv', index=False)
```

[3]:
```
backend_Q = service.get_backend(df.backend_Q[0])
backend_Q.status()
```

[3]:
```
<qiskit.providers.models.backendstatus.BackendStatus at 0x2109434b090>
```

[4]:
```
# Maximum execution time in seconds
my_options = {'max_execution_time': 30}

# Classical and quantum estimators
estimator_S = EstimatorV2(backend=backend_S)
```

```
estimator_Q = EstimatorV2(backend=backend_Q, options=my_options)

# Classical and quantum samplers
sampler_S = SamplerV2(backend=backend_S)
sampler_Q = SamplerV2(backend=backend_Q, options=my_options)
```

Random numbers

The generation of random numbers using a classical simulator of a quantum machine was conducted during the second session. Now, let's generate "true random numbers"! To achieve this, we define a 3-qubit circuit that prepares each of them in the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and measures them in the computational basis $|0\rangle, |1\rangle$. This setup ensures that the measurement outcomes will be 0 or 1 with equal probabilities of 0.5.

Question 1: Study the definition of the circuit with care.

Answer: It creates a barrier of Hadamard gates, which prepares each qubit in an equal superposition of $|0\rangle$ and $|1\rangle$. The circuit assigns three qubits to each random number to be generated, meaning the random numbers range between 0 and 7. This is an example of the *Hadamard transform*, which in this case performs the following operation:

$$H^{\otimes 90}|0\rangle^{\otimes 90} = |+\rangle^{\otimes 90}$$

More generally, if we have a register initialised to a state $|j\rangle$ of the computational basis, the action of a Hadamard transform is the following:

$$H^{\otimes n}|j\rangle = \bigotimes_{i=1}^{n} H|j_i\rangle = \frac{1}{2^{n/2}}\bigotimes_{i=1}^{n}(|0\rangle + (-1)^{j_i}|1\rangle) = \frac{1}{2^{n/2}}\bigotimes_{i=1}^{n}\sum_{k_i=0}^{1}(-1)^{k_i j_i}|k_i\rangle =$$

$$= \frac{1}{2^{n/2}}\sum_{k_1=0}^{1}(-1)^{k_1 j_1}|k_1\rangle \sum_{k_2=0}^{1}(-1)^{k_2 j_2}|k_2\rangle ... \sum_{k_n=0}^{1}(-1)^{k_n j_n}|k_n\rangle =$$

$$= \frac{1}{2^{n/2}}\sum_{k_1=0}^{1}\sum_{k_2=0}^{1}...\sum_{k_n=0}^{1}(-1)^{\sum_{i=1}^{n} k_i j_i}|k_1 k_2 ... k_n\rangle \equiv$$

$$\equiv \frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}(-1)^{\sum_{i=1}^{n} k_i j_i}|k\rangle$$

[100]:
```
# How many random numbers will be produced in a single shot:
Nnumbers = 30  # Must be smaller than [127 qubits/3]=42
# Prepare the input circuit
QRNG = QuantumCircuit(3*Nnumbers) # Three qubits per number
QRNG.h(range(3*Nnumbers)) # Apply Hadamard gate to each of the qubits
QRNG.measure_all() # Measure all qubits
```

```
QRNG = transpile(QRNG, backend=backend_Q) # Adapt circuit to architecture of␣
↪quantum machine
```

[34]:
```
# Execute the circuit directly on a quantum computer
job_qrng = sampler_Q.run([QRNG], shots=1)

# Save job identification code for later
df.qrng_id = job_qrng.job_id()
df.to_csv('session.csv', index=False)
```

```
c:\Users\zapat\Escritorio\CODE\PIE_Compu_Cuantica\venv\Lib\site-
packages\qiskit_ibm_runtime\qiskit_runtime_service.py:879: UserWarning: Your
current pending jobs are estimated to consume 649.5556236231932 quantum seconds,
but you only have 600 quantum seconds left in your monthly quota; therefore, it
is likely this job will be canceled
  warnings.warn(warning_message)
```

[81]:
```
# The following cell retrieves information about the job
job_qrng = service.job(df.qrng_id[0])
job_qrng.status()
```

[81]:
```
<JobStatus.DONE: 'job has successfully run'>
```

[82]:
```
# The next statement can put your session on hold until the job runs and returns
result_qrng = job_qrng.result()
```

Question 2: Comment on your results.

Answer: We obtained the following result from the IBM Quantum computer run in the lab with UCM tokens:

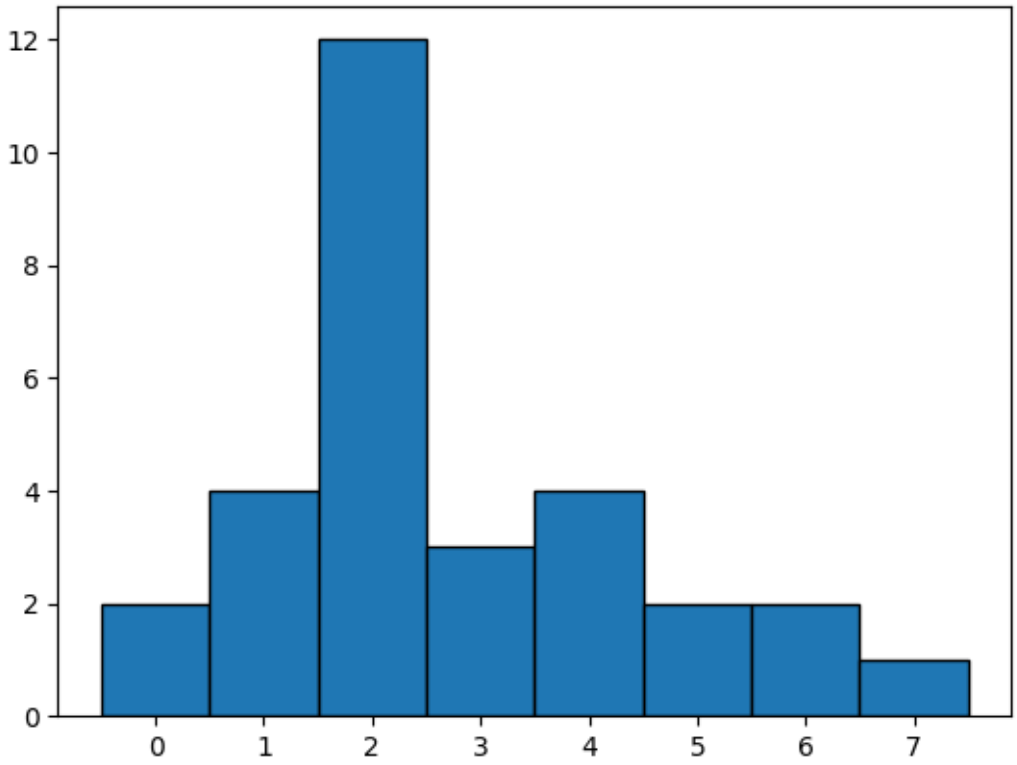
```
[{236732400872330227231122054: 1.0}]
```

After depuration we obtain the following list of numbers

```
[6, 0, 2, 1, 6, 4, 4, 5, 2, 2, 3, 3, 4, 2, 2, 3, 2, 1, 2, 5, 2, 2, 2, 0, 2, 2, 7,
1, 4, 1]
```

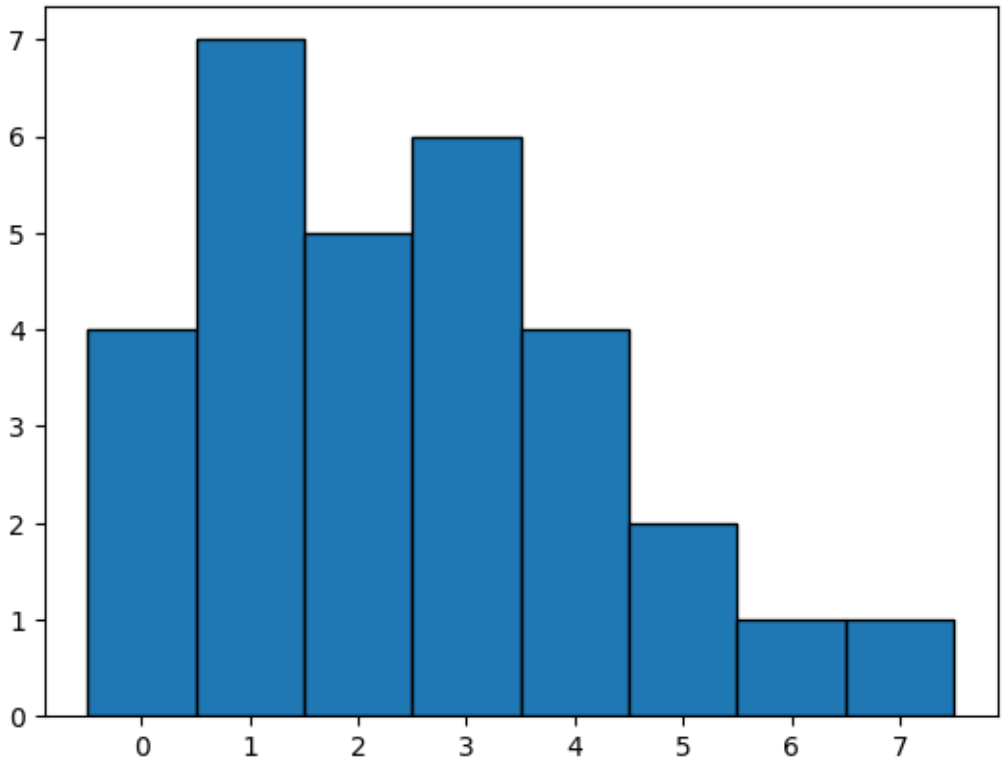
that we can plot in a histogram.

[7]:
```
RandomNumbers = [6, 0, 2, 1, 6, 4, 4, 5, 2, 2, 3, 3, 4, 2, 2, 3, 2, 1, 2, 5, 2,␣
↪2, 2, 0, 2, 2, 7, 1, 4, 1]
plt.hist(RandomNumbers, bins=range(9), align='left', edgecolor='black',␣
↪linewidth=1)
plt.show()
```



We can likewise retrieve our own results run at home.

[134]:
```
bits, = result_qrng[0].data.meas.get_counts()
numbers = [bits[i:i+3] for i in range(0, 3*Nnumbers, 3)]
decimal_numbers = [int(n, 2) for n in numbers]
plt.hist(decimal_numbers, bins=range(9), align='left', edgecolor='black',␣
↪linewidth=1)
plt.show()
```



The distribution is not "particularly uniform", it rather looks more log-normal. However, we generated very few random numbers, to draw any statistical conclusions we would likely have to generate many more.

Deutsch's algorithm

To encode the action of a Boolean fuction, Deutsch's algorithm uses the following operation over two qubits

$$|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$$

where $\oplus$ denotes the binary addition (i.e. the addition mod. 2).

Homework 1 (to do later after the lab on pen and paper): Prove that this is a unitary operation.

Answer: It is clear that the quantum oracle preserves the norm, therefore it is unitary. If we take $(x, y) \in \{0, 1\}^2, f(x) \in \{0, 1\}$, we have that $y \oplus f(x) \in \{0, 1\}$ — in other words, $|x\rangle|y \oplus f(x)\rangle$ is a ket from the computational basis of the Hilbert space. The operation might be non-invertible if it mapped two different kets to a single one, but this is impossible, since $0 \oplus f(x) \neq 1 \oplus f(x)$:

$$(\langle x|\langle 0|)U_f^\dagger U_f(|x\rangle|1\rangle) = (\langle x|\langle 0 \oplus f(x)|)(|x\rangle|1 \oplus f(x)\rangle) = \langle x|x\rangle\langle 0 \oplus f(x)|1 \oplus f(x)\rangle = 0$$

Alternatively, given there exist a finite number of Boolean functions, we can calculate the oracle matrix for each f to better understand what U_f does. If we place the control qubit $|x\rangle$ first, as we'll later do in the circuits (so $|y, x\rangle \rightarrow |y \oplus f(x), x\rangle$), the representations of U_f are

$$f(x) = 0: \quad U_f = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad f(x) = 1: \quad U_f = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$f(x) = x: \quad U_f = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad f(x) = \neg x: \quad U_f = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

These matrices all represent permutations, which are unitary operations.

Question 2: Complete the following cells to create quantum circuits for Oracle_f that decides whether f is (or not) constant using the classical method with two evaluations.

Answer: To evaluate f using the oracle we can measure the leftmost qubit in the following two states:

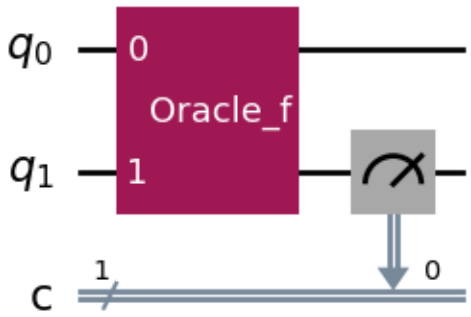
$$U_f|0\rangle|0\rangle = |f(0)\rangle|0\rangle$$

$$U_f|0\rangle|1\rangle = |f(1)\rangle|1\rangle$$

N.B.: Qubit significance and indexing in Qiskit goes from right to left, top to bottom, 0 to $n-1$.

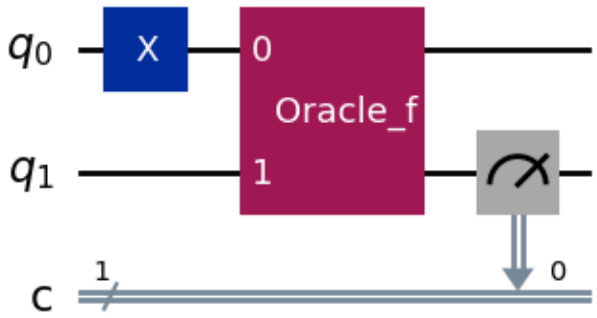
[9]:
```
Oracle_f = Operator([[0,0,1,0],[0,1,0,0],[1,0,0,0],[0,0,0,1]])
Oracle_g = Operator([[0,0,1,0],[0,0,0,1],[1,0,0,0],[0,1,0,0]])

# First evaluation
Check_f0 = QuantumCircuit(2,1)
Check_f0.unitary(Oracle_f, [0, 1], label='Oracle_f')
Check_f0.measure(1, 0)
Check_f0.draw(output='mpl')
```

[9]:



[123]:
```
# Second evaluation
Check_f1 = QuantumCircuit(2,1)
Check_f1.x(0)
Check_f1.unitary(Oracle_f, [0, 1], label='Oracle_f')
Check_f1.measure(1,0)
Check_f1.draw(output='mpl')
```

[123]:



[183]:
```
# Run both evaluations
job_clf = sampler_S.run([Check_f0, Check_f1], shots=1)
result_clf = job_clf.result()
```

Question 3: By printing "result_clf.quasi_dists", explain why the following cell gives the calculated result for $f(0)$ and $f(1)$.

Answer: The function $f(x)$ appears to be $\neg x$, and therefore a balanced function. Indeed, if we compare the `Oracle_f` with the matrix we calculated above for $f(x) = \neg x$, they coincide.

N.B.: For my calculations at home I used the `SamplerV2` class which has slightly different methods for retrieving results as compared to `Sampler`.

[184]:
```
# Print the measurement results
print(f'f(0) = {int(*result_clf[0].data.c.get_counts())}')
print(f'f(1) = {int(*result_clf[1].data.c.get_counts())}')
```

```
f(0) = 1
f(1) = 0
```

Question 4: Complete the following cells to create quantum circuits for Oracle_g that solve the problem of whether g is constant using the classical method of two evaluations.

Answer: We may repeat the previous process. We find that $g(x) = 1$, therefore g is a constant function.

[186]:
```
# First evaluation
Check_g0 = QuantumCircuit(2,1)
Check_g0.unitary(Oracle_g, [0, 1], label='Oracle_g')
Check_g0.measure(1, 0)

# Second evaluation
Check_g1 = QuantumCircuit(2,1)
Check_g1.x(0)
Check_g1.unitary(Oracle_g, [0, 1], label='Oracle_g')
Check_g1.measure(1,0)

# Run both evaluations
job_clg = sampler_S.run([Check_g0, Check_g1], shots=1)
result_clg = job_clg.result()

# Print the measurement results
print(f'g(0) = {int(*result_clg[0].data.c.get_counts())}')
print(f'g(1) = {int(*result_clg[1].data.c.get_counts())}')
```

```
g(0) = 1
g(1) = 1
```

The idea of Deutsch's algorithm is based on the use of quantum superposition to attempt a simultaneous evaluation and comparison of $f(0)$ and $f(1)$ with a single action of the quantum oracle.

If $f(0) = f(1)$,

$$U_f|+\rangle|-\rangle = \frac{1}{2}[|0\rangle|f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|f(0)\rangle - |1\rangle|1 \oplus f(0)\rangle]$$

$$= \frac{1}{2}[(|0\rangle + |1\rangle)|f(0)\rangle - (|0\rangle + |1\rangle)|1 \oplus f(0)\rangle] = |+\rangle \left[\frac{|f(0)\rangle - |1 \oplus f(0)\rangle}{\sqrt{2}}\right]$$

Homework question 5: Prove (after the lab, on pen and paper): the alternative case, that if $f(0) \neq f(1)$, the final state becomes:

$$|-\rangle \left[\frac{|f(0)\rangle - |f(1)\rangle}{\sqrt{2}}\right].$$

Answer: If $f(0) \neq f(1)$, then $f(0) \oplus 1 = f(1)$ and $f(1) \oplus 1 = f(0)$, therefore

$$|+\rangle|-\rangle = \frac{1}{2}(|0\rangle|0\rangle - |0\rangle|1\rangle + |1\rangle|0\rangle - |1\rangle|1\rangle) \rightarrow$$

$$\frac{1}{2}[|0\rangle|f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle] =$$

$$= \frac{1}{2}[|0\rangle|f(0)\rangle - |0\rangle|f(1)\rangle + |1\rangle|f(1)\rangle - |1\rangle|f(0)\rangle] =$$

$$= \frac{1}{2}[|0\rangle - |1\rangle][|f(0)\rangle - |f(1)\rangle] = |-\rangle \left[\frac{|f(0)\rangle - |f(1)\rangle}{\sqrt{2}}\right]$$

After running $|+\rangle|-\rangle$ through the oracle, we may apply a Hadamard gate to the leftmost qubit and then measure it. This way we obtain $H|+\rangle = |0\rangle$ if f is constant and $H|-\rangle = |1\rangle$ if f is balanced.

A better way to understand the solution to Deutsch's problem is through *phase kickback*, the fact that controlled operations apply a phase to their control qubits. By considering it, we can simultaneously solve both cases in the problem statement:

$$U_f|+\rangle|-\rangle = \frac{1}{2}\Big(|0\rangle(|f(0)\oplus 0\rangle - |f(0)\oplus 1\rangle) + |1\rangle(|f(1)\oplus 0\rangle - |f(1)\oplus 1\rangle)\Big) =$$

$$= \frac{1}{2}\Big((-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle)\Big) =$$

$$= \frac{1}{\sqrt{2}}\Big((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\Big)|-\rangle =$$

$$= \frac{(-1)^{f(0)}}{\sqrt{2}}\Big(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle\Big)|-\rangle$$

The modest quantum advantage in the Deutsch algortihm can be improved by keeping phase kickback in mind. In the Deutsch-Jozsa algorithm [1], the oracle now implements $f : \{0,1\}^n \to \{0,1\}$, which we are promised is either constant or balanced, and we are asked to classify it. We start with state $|0\rangle^{\otimes n}|1\rangle$, to which we apply a Hadamard transform:

$$H^{\otimes(n+1)}(|0\rangle^{\otimes n}|1\rangle) = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}|x\rangle|-\rangle$$

If we now run this state through the oracle, as before, we get

$$\frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle|-\rangle$$

Next, we take the n control qubits and apply a Hadamard transform to them:

$$H^{\otimes n}\left[\frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle\right] = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\left[\frac{1}{2^{n/2}}\sum_{y=0}^{2^n-1}(-1)^{x\cdot y}|y\rangle\right] = \sum_{y=0}^{2^n-1}\left[\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\right]|y\rangle$$

Finally, we conclude that the probability to measure $|z\rangle$ is $\left|\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot z}\right|^2$, so

$$\text{Prob}(|0\rangle^{\otimes n}) = \left|\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\right|^2 = \begin{cases}1 \text{ if } f(x) \text{ is constant}\\ \\ 0 \text{ if } f(x) \text{ is balanced}\end{cases}$$

thus solving the problem in a single evaluation of f. For a classical solution, we would have to check just over half of all possible bit strings ($2^{n-1}+1$), meaning the Deutsch-Jozsa algorithm has an exponential advantage over classical algorithms.

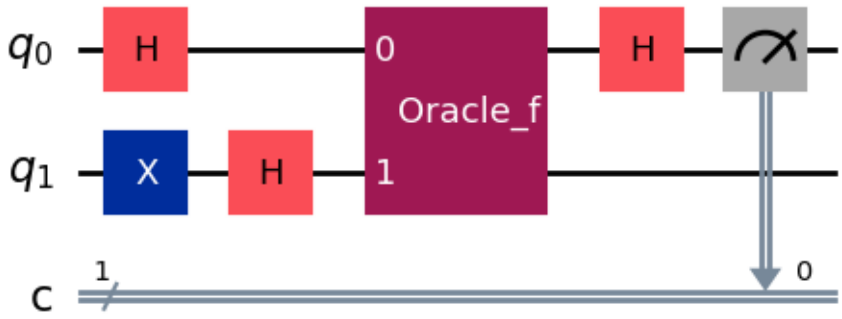
Question 6: Define a quantum circuit to implement Deutsch's algorithm with the function f and another with the function g.

Answer:

For the function f:

[10]:
```
deutsch_f = QuantumCircuit(2,1)
deutsch_f.x(1)
deutsch_f.h([0, 1])
deutsch_f.unitary(Oracle_f, [0, 1], label='Oracle_f')
deutsch_f.h(0)
deutsch_f.measure(0,0)
deutsch_f.draw(output='mpl')
```

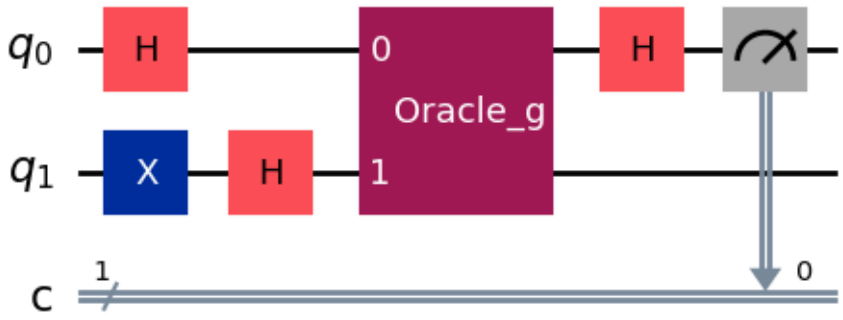
[10]:



And for the function g we just change the oracle:

[11]:
```
deutsch_g = QuantumCircuit(2,1)
deutsch_g.x(1)
deutsch_g.h([0, 1])
deutsch_g.unitary(Oracle_g, [0, 1], label='Oracle_g')
deutsch_g.h(0)
deutsch_g.measure(0,0)
deutsch_g.draw(output='mpl')
```

[11]:



[26]:
```
# Execute the circuit
job_Deutsch_S = sampler_S.run([deutsch_f, deutsch_g], shots=1)
result_Deutsch_S = job_Deutsch_S.result()

# Print the measurement results
if int(*result_Deutsch_S[0].data.c.get_counts()) == 0:
    print('f is constant')
else:
    print('f is balanced')
if int(*result_Deutsch_S[1].data.c.get_counts()) == 0:
    print('g is constant')
else:
    print('g is balanced')
```

```
f is balanced
g is constant
```

Question 7: Does the result of your quantum computation using Deutsch's algorithm agree with the previous classical result?

Answer: The classical result is that f is balanced and g is constant. After applying Deutsch's algorithm to both functions, we obtained the same result — that is, that f is balanced and g is constant. Our run in the lab with UCM tokens yielded the same result.

[13]:
```
# Transpile circuits to backend architecture
deutsch_f_t = transpile(deutsch_f, backend=backend_Q)
deutsch_g_t = transpile(deutsch_g, backend=backend_Q)

# Execute the circuit
job_Deutsch_Q = sampler_Q.run([deutsch_f_t, deutsch_g_t], shots=3)

# Save job identification code for later
df.deutsch_id = job_Deutsch_Q.job_id()
```

```
df.to_csv('session.csv', index=False)
```

```
c:\Users\zapat\Escritorio\CODE\PIE_Compu_Cuantica\venv\Lib\site-
packages\qiskit_ibm_runtime\qiskit_runtime_service.py:879: UserWarning: Your
current pending jobs are estimated to consume 649.5556236231932 quantum seconds,
but you only have 588 quantum seconds left in your monthly quota; therefore, it
is likely this job will be canceled
  warnings.warn(warning_message)
```

[5]:
```
# The following cell retrieves information about the job
job_Deutsch_Q = service.job(df.deutsch_id[0])
job_Deutsch_Q.status()
```

[5]:
```
<JobStatus.DONE: 'job has successfully run'>
```

[6]:
```
# The next statement can put your session on hold until the job runs and returns
result_Deutsch_Q = job_Deutsch_Q.result()
ans_f = result_Deutsch_Q[0].data.c.get_counts()
ans_g = result_Deutsch_Q[1].data.c.get_counts()

# Print the measurement results
if int(max(ans_f, key=ans_f.get)) == 0:
    print('f is constant')
else:
    print('f is balanced')
if int(max(ans_g, key=ans_g.get)) == 0:
    print('g is constant')
else:
    print('g is balanced')
```

```
f is balanced
g is constant
```

Experimental measurement of a Bell inequality (CHSH combination of correlators)

If $\{A_1, A_2\}$ and $\{B_1, B_2\}$ are two pairs of observables (with dichotomic/binary outcome) of two spatially separated systems, the expected values of their products $\langle A_i B_j \rangle$ according to any local hidden variable model (that is, an attempt at trying to explain away quantum features with additional classical mechanics variables) satisfy the classical CHSH inequality

$$|\langle A_1 B_1 \rangle + \langle A_1 B_2 \rangle + \langle A_2 B_1 \rangle - \langle A_2 B_2 \rangle| \leq 2.$$

Quantum theory, on the contrary, predicts that this inequality is violated for a suitable choice of observables, obtaining the maximum violation, in the case of two qubits, when

$$A_1 = X, \quad A_2 = Y, \quad B_1 = \frac{-(X+Y)}{\sqrt{2}}, \quad B_2 = \frac{-(X-Y)}{\sqrt{2}}$$

or rotationally equivalent configurations. Here X and Y denote the σ_x and σ_y Pauli matrices.

Question 2: Using this command, define the four product observables that appear in the CHSH inequality: A_1B_1, A_2B_1, A_1B_2, and A_2B_2:

Answer:

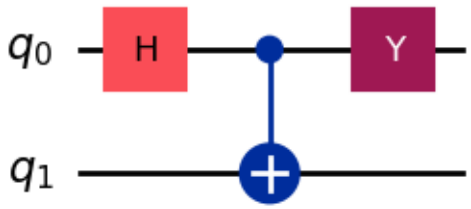
[4]:
```
# Define pairs of observables for maximum violation of the CHSH innequality
coef = 1 / np.sqrt(2)
A1B1aux = SparsePauliOp.from_list([('XX', - coef), ('XY', - coef)])
A1B2aux = SparsePauliOp.from_list([('XX', - coef), ('XY', + coef)])
A2B1aux = SparsePauliOp.from_list([('YX', - coef), ('YY', - coef)])
A2B2aux = SparsePauliOp.from_list([('YX', - coef), ('YY', + coef)])
Obsaux = [A1B1aux, A1B2aux, A2B1aux, A2B2aux]

# We need to fill the rest of the qubits with the identity (do nothing)
# this is thought for the current 127-qubit machines
n_remaining_qubits = 125
I125 = SparsePauliOp('I' * n_remaining_qubits)
Obs = [I125.tensor(op) for op in Obsaux]
```

Question 1: Define a circuit that prepares the "singlet state" of two qubits $|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$.

Answer: The following circuit specifically prepares the state $\frac{i}{\sqrt{2}}(|01\rangle - |10\rangle)$, which is physically equivalent to the requested state.

[5]:
```
# Prepare the input circuit:
chsh_circuit = QuantumCircuit(2)
chsh_circuit.h(0) # Hadamard on the first qubit
chsh_circuit.cx(0, 1) # NOT controlled to the first qubit
chsh_circuit.y([0]) # Y-Pauli gate to the first qubit
chsh_circuit.draw(output='mpl')
```

[5]:



Question 3: Calculate the simulated value obtained for the CHSH inequality jointly with its error.

Answer: The code we ran in the lab with UCM tokens yielded:

```
The simulated result is 2.81824 + 0.04:

does it exceed 2 with sufficient statistical certainty?
```

We can perform the calculations again ourselves.

[8]:
```
# Execute the circuit
job_CHSH_S = estimator_S.run([(chsh_circuit, Obsaux)])
result_S = job_CHSH_S.result()

# Statistical results
values_S = result_S[0].data.evs
Standard_errors_S = result_S[0].data.stds

# Print simulated expectation values
print('Simulated expectation values for the four correlators:')
print([f'{v:.3f} ± {s:.3f}' for v, s in zip(values_S, Standard_errors_S)])
```

```
Simulated expectation values for the four correlators:
['0.711 ± 0.011', '0.703 ± 0.011', '0.702 ± 0.011', '-0.713 ± 0.011']
```

[9]:
```
# CHSH value
CHSH_mean_S = abs(values_S[0] + values_S[1] + values_S[2] - values_S[3])
CHSH_uncertainty_S = sum(Standard_errors_S)
print(f'''
The simulated result is {CHSH_mean_S:.3f} ± {CHSH_uncertainty_S:.3f}:
does it exceed 2 with sufficient statistical certainty?
''')
```

```
The simulated result is 2.828 ± 0.044:
does it exceed 2 with sufficient statistical certainty?
```

Question 4: Compute, with a hand calculator or simple python commands, the experimental value obtained for the CHSH inequality jointly with its error from the quantum data obtained.

Answer: The code we ran in the lab with UCM tokens yielded:

```
The simulated result is 2.60215 + 0.41467:

does it exceed 2 with sufficient statistical certainty?
```

Let's perform the calculations again ourselves.

[65]:
```
# Transpile circuit to backend architecture
chsh_circuit_t = transpile(chsh_circuit, backend=backend_Q)

# Execute the circuit
job_CHSH_Q = estimator_Q.run([(chsh_circuit_t, Obs)])
```

```
# Save job identification code for later
df.chsh_id = job_CHSH_Q.job_id()
df.to_csv('session.csv', index=False)
```

```
c:\Users\zapat\Escritorio\CODE\PIE_Compu_Cuantica\venv\Lib\site-
packages\qiskit_ibm_runtime\qiskit_runtime_service.py:879: UserWarning: Your
current pending jobs are estimated to consume 623.1034379365619 quantum seconds,
but you only have 586 quantum seconds left in your monthly quota; therefore, it
is likely this job will be canceled
  warnings.warn(warning_message)
```

[7]:
```
# The following cell retrieves information about the job
job_CHSH_Q = service.job(df.chsh_id[0])
job_CHSH_Q.status()
```

[7]:
```
<JobStatus.DONE: 'job has successfully run'>
```

[8]:
```
# The next statement can put your session on hold until the job runs and returns
result_Q = job_CHSH_Q.result()

# Statistical results
values_Q = result_Q[0].data.evs
Standard_errors_Q = result_Q[0].data.stds

# Print simulated expectation values
print('Simulated expectation values for the four correlators:')
print([f'{v:.3f} ± {s:.3f}' for v, s in zip(values_Q, Standard_errors_Q)])
```

```
Simulated expectation values for the four correlators:
['0.689 ± 0.036', '0.801 ± 0.036', '0.744 ± 0.034', '-0.665 ± 0.034']
```

[9]:
```
# CHSH value
values_Q = result_Q[0].data.evs
Standard_errors_Q = result_Q[0].data.stds
CHSH_mean_Q = abs(values_Q[0] + values_Q[1] + values_Q[2] - values_Q[3])
CHSH_uncertainty_Q = sum(Standard_errors_Q)
print(f'''
The simulated result is {CHSH_mean_Q:.3f} ± {CHSH_uncertainty_Q:.3f}:
does it exceed 2 with sufficient statistical certainty?
''')
```

```
The simulated result is 2.899 ± 0.139:
does it exceed 2 with sufficient statistical certainty?
```

The qubits we used for our computation exhibit quantum statistics, and thus are properly entangled. A computation such as this serves just as much as a proof of CHSH violation as it does as a

calibration of a quantum computer.

References

[1] Cleve, R., Ekert, A., Macchiavello, C., & Mosca, M. (1998). Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences, 454(1969), 339–354.* Retrieved from https://arxiv.org/abs/quant-ph/9708016.