Notes Qiskit UF

Quantum States and Qubits

Introduction

The Atoms of Computation

Understanding first classical computation and bits, using the same tools we will use later for quantum.

In [1]:
```
from qiskit import QuantumCircuit, assemble, Aer
from qiskit.visualization import plot_histogram
```

In [2]:
```
from qiskit_textbook.widgets import binary_widget
binary_widget(nbits=5)
```

First quantum circuit

3 jobs:

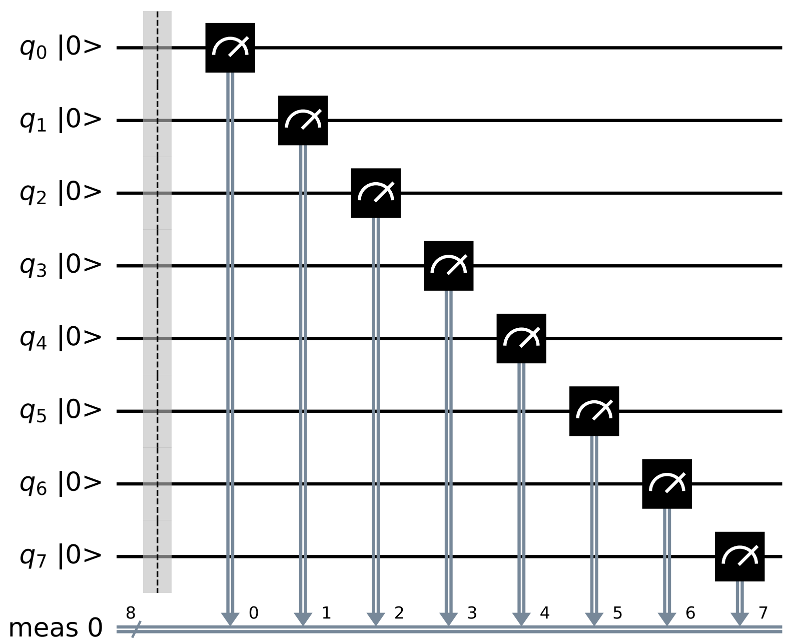
- enconde the input
- actual computation
- extract output

In [3]:
```
# define QC
qc_output = QuantumCircuit(8) # takes n of bits as argument
```

In [4]:
```
# add measurement
qc_output.measure_all() # extraction of outputs
```

In [5]:
```
qc_output.draw(initial_state=True)
```

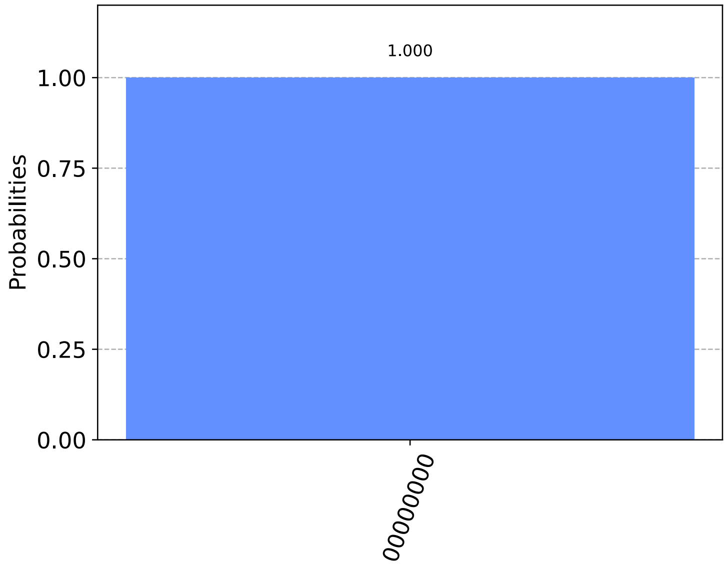
Out[5]:



Notice the qubits are **always** initialized at $|0\rangle$.

In [6]:
```
# simulating the circuit multiple times
sim = Aer.get_backend('aer_simulator')
result = sim.run(qc_output).result()
count = result.get_counts()
plot_histogram(count)
```

Out[6]:



We run many times because quantum computers may have some randomness when measuring.

Notice this is a **simulation**, which can be done only to a small number of qubits ≈ 30. To run in a real device, just need to replace `Aer.get_backend('aer_simulator')` with the backend of the device.

Example: creating an Adder circuit (item 4 on textbook)

Remember, we need to:

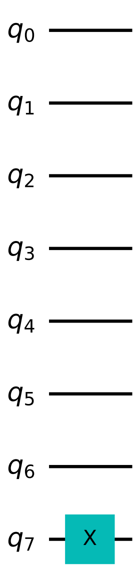
- enconde the input
- actual computation
- extract output

Enconding the input

NOT gate first: flips the qubit -> x

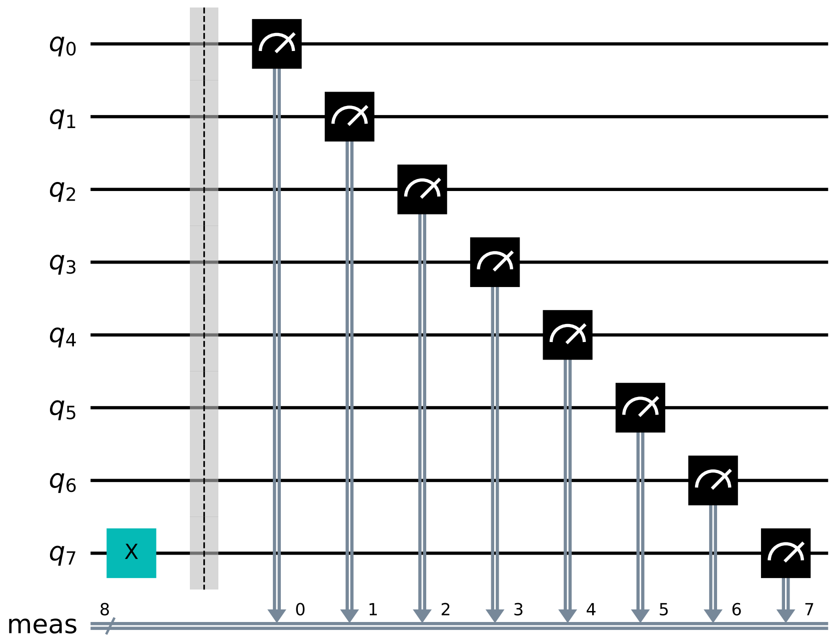
In [7]:
```
qc_encode = QuantumCircuit(8)
qc_encode.x(7) #(the very last qubit)
qc_encode.draw()
```

Out[7]:



In [8]:
```
qc_encode.measure_all()
qc_encode.draw()
```

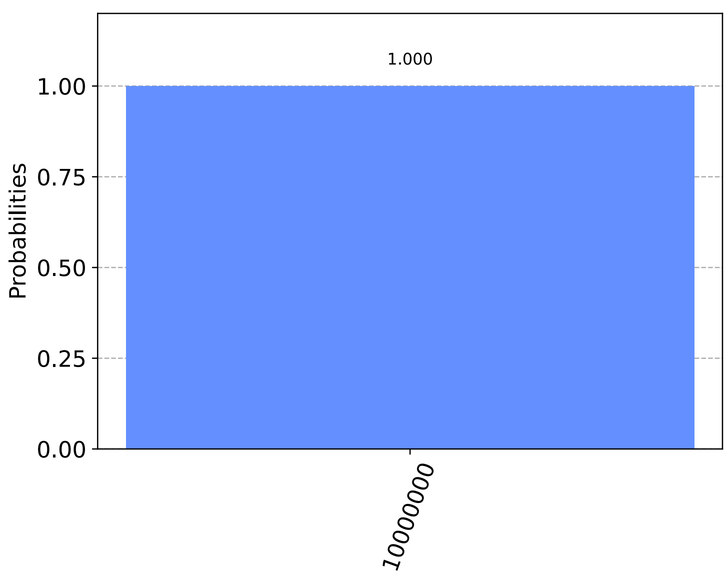
Out[8]:



And similarly to before, we can simulate it:

In [9]:
```
sim = Aer.get_backend('aer_simulator')
result = sim.run(qc_encode).result()
counts = result.get_counts()
plot_histogram(counts)
```

Out[9]:

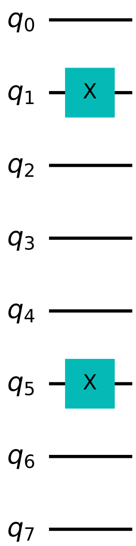


Notice it reads from right to left, to be similar to the representation of numbers in the decimal system

$$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 10000000$$

In [10]:
```
qc_encode = QuantumCircuit(8)
qc_encode.x(1)
qc_encode.x(5)

qc_encode.draw()
```

Out[10]:



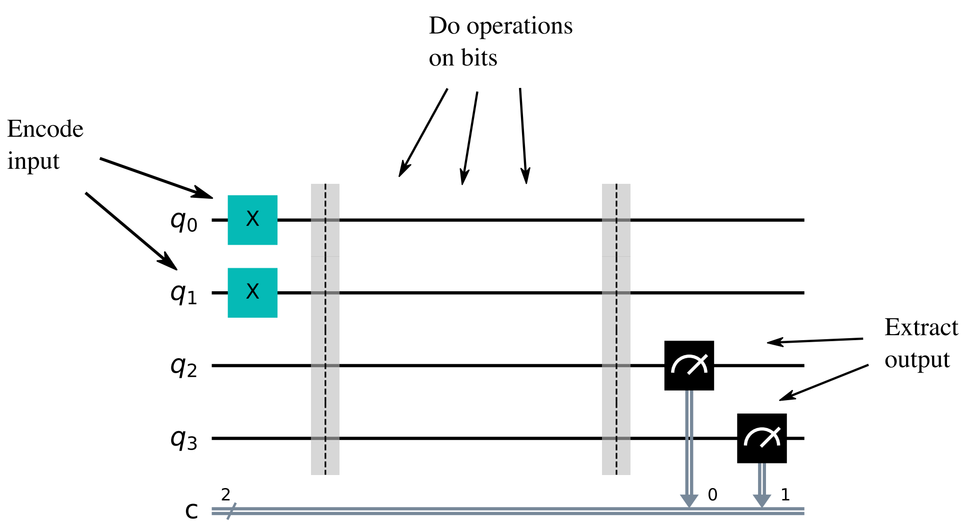
Remember the usual addition, carry one algorithm.

$$
\begin{aligned}
&10 && (1)\\
&+01 && (2)\\
&=11 && (3)
\end{aligned}
$$

The sums in decimal then becomes,

$$
\begin{aligned}
0+0&=00 && (4)\\
0+1&=01 && (5)\\
1+0&=01 && (6)\\
1+1&=10 && (7)
\end{aligned}
$$

which is called a **half adder**.



The two qubits to add are encoded in 0 and 1. In the above circuit, we are looking for the solution of $1+1$. The results will be stored on the qubits 2 and 3 and will store in classical bits 0 and 1, respectively.

Dashed lines are made with the `barrier` *command.*

Remember

$$
\begin{aligned}
0+0&=00 && (8)\\
0+1&=01 && (9)\\
1+0&=01 && (10)\\
1+1&=10 && (11)
\end{aligned}
$$

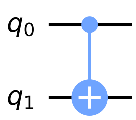
Notice this seems like **XOR gate**:

$$0+0=0 \tag{12}$$
$$0+1=1 \tag{13}$$
$$1+0=1 \tag{14}$$
$$1+1=0 \tag{15}$$

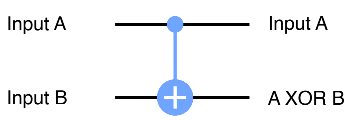
which in quantum computers is done by the **CNOT gate**, `cx` .

In [11]:
```
qc_cnot = QuantumCircuit(2)
qc_cnot.cx(0,1) ## first is the control, second is the target
qc_cnot.draw()
```

Out[11]:



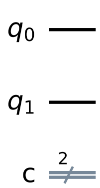
It compares both inputs to see if they are different, and overwrites the target bit with the answer. Or, said in another manner, it flips the target if the control is 1.



Trying it:

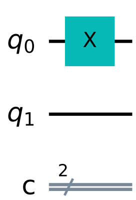
In [12]:
```
qc = QuantumCircuit(2,2) # Two quantum, two classical regimes
qc.draw()
```

Out[12]:



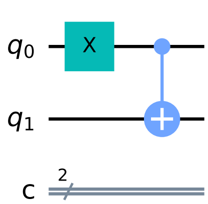
In [13]:
```
qc.x(0)
qc.draw()
```

Out[13]:



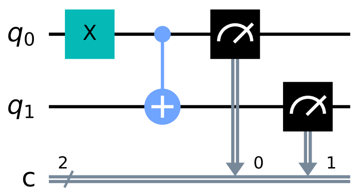
In [14]:
```
qc.cx(0,1)
qc.draw()
```

Out[14]:



In [15]:
```
qc.measure(0,0)
qc.measure(1,1)
qc.draw()
```

Out[15]:

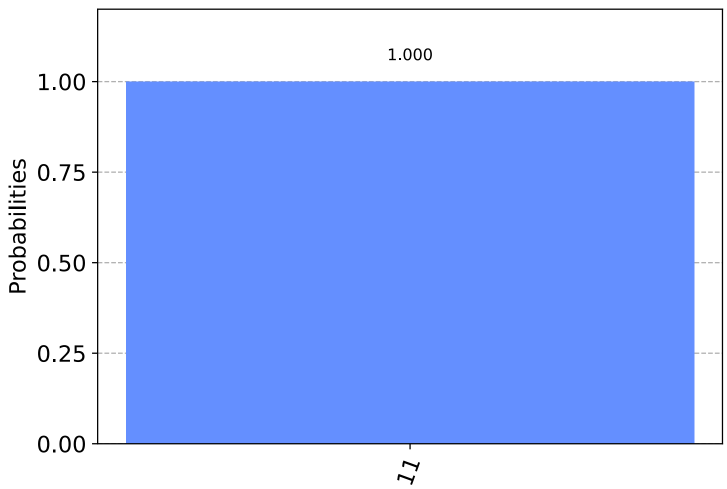


The result should be 11

In [16]:

```
sim = Aer.get_backend('aer_simulator')
result = sim.run(qc).result()
counts = result.get_counts()
plot_histogram(counts)
```

Out[16]:

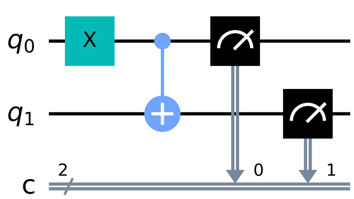


Or, all in one single cell:

In [17]:

```
qc = QuantumCircuit(2,2)
qc.x(0)
qc.cx(0,1)
qc.measure(0,0)
qc.measure(1,1)
qc.draw()
```

Out[17]:



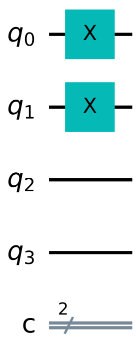
For our half adder, we don't want to overwrite one of our inputs.

Instead, *we want to write the result on a different pair of qubits*. For this, we can use two CNOTs.

In [18]:

```
qc_ha = QuantumCircuit(4,2)
# encode inputs in qubits 0 and 1
qc_ha.x(0) # For a=0, remove this line. For a=1, leave it.
qc_ha.x(1) # For b=0, remove this line. For b=1, l
qc_ha.draw()
```

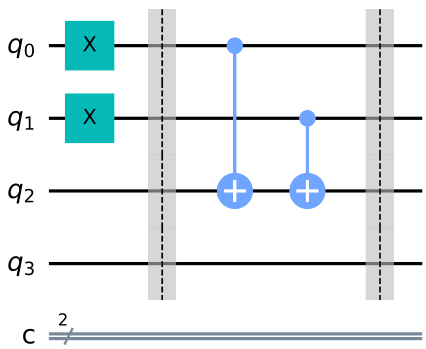
Out[18]:



In [19]:

```
qc_ha.barrier()
# use cnots to write the XOR of the inputs on qubit 2
qc_ha.cx(0,2)
qc_ha.cx(1,2)
qc_ha.barrier()
qc_ha.draw()
```

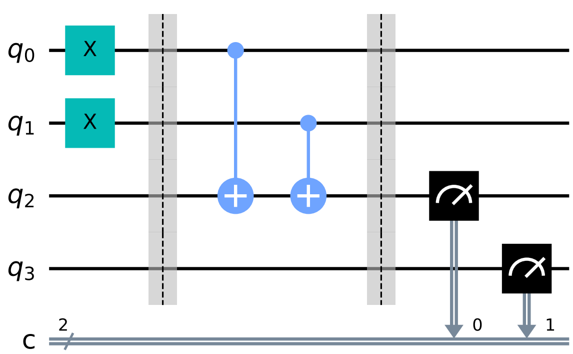
Out[19]:



In [20]:

```
# extract outputs
qc_ha.measure(2,0) # extract XOR value
qc_ha.measure(3,1)

qc_ha.draw()
```

Out[20]:

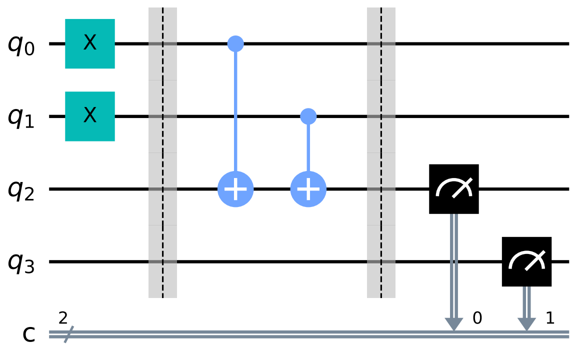


All in one cell:

In [21]:

```
qc_ha = QuantumCircuit(4,2)
# encode inputs in qubits 0 and 1
qc_ha.x(0) # For a=0, remove this line. For a=1, leave it.
qc_ha.x(1) # For b=0, remove this line. For b=1, leave it.
qc_ha.barrier()
# use cnots to write the XOR of the inputs on qubit 2
qc_ha.cx(0,2)
qc_ha.cx(1,2)
qc_ha.barrier()
# extract outputs
qc_ha.measure(2,0) # extract XOR value
qc_ha.measure(3,1)

qc_ha.draw()
```

Out[21]:



So, q_2 has the result of the first bit.

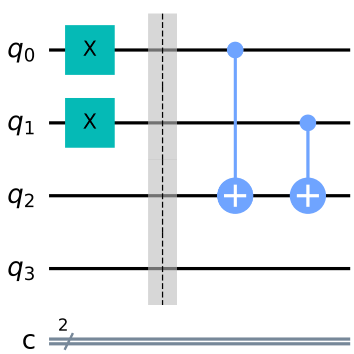
For the second qubit, recorded in $q3$, it will only be 1 when $1 + 1 = 10$. Therefore, we can check when both inputs are 1. If both are, we need a **NOT** gate on $q3$, controlled on both $q1$ and $q2$: **Toffoli gate** (basically an AND gate); `ccx` .

Repeating the circuit above:

In [22]:

```
qc_ha = QuantumCircuit(4,2)
# encode inputs in qubits 0 and 1
qc_ha.x(0) # For a=0, remove the this line. For a=1, leave it.
qc_ha.x(1) # For b=0, remove the this line. For b=1, leave it.
qc_ha.barrier()
# use cnots to write the XOR of the inputs on qubit 2
qc_ha.cx(0,2)
qc_ha.cx(1,2)
qc_ha.draw()
```

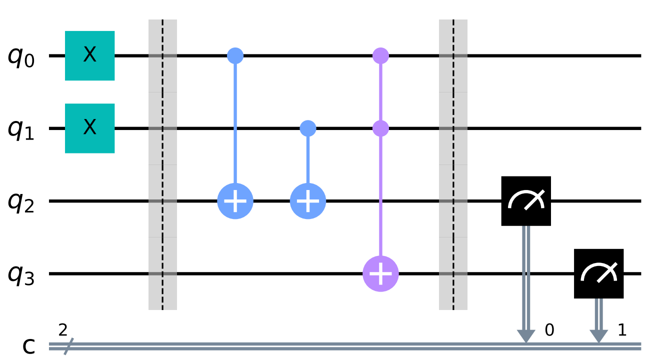
Out[22]:



In [23]:

```
# use ccx to write the AND of the inputs on qubit 3
qc_ha.ccx(0,1,3)
qc_ha.barrier()
# extract outputs
qc_ha.measure(2,0) # extract XOR value
qc_ha.measure(3,1) # extract AND value

qc_ha.draw()
```

Out[23]:



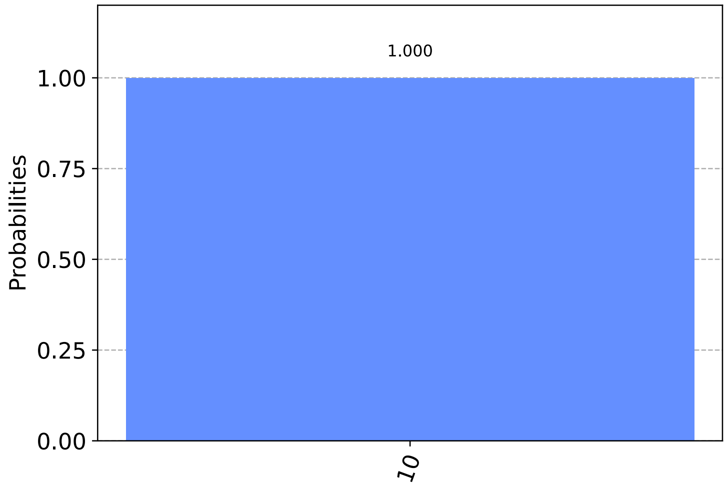
Now let's calculate the outcomes of this circuit. Notice that so far we have only created the circuit. Now let's calculate it:

In [24]:

```
#### Notice that assemble -> qobj is not needed anymore,
#### can run the circuit directly
```

```
#qobj = assemble(qc_ha)
#counts = sim.run(qobj).result().get_counts()

counts = sim.run(qc_ha).result().get_counts()
plot_histogram(counts)
```

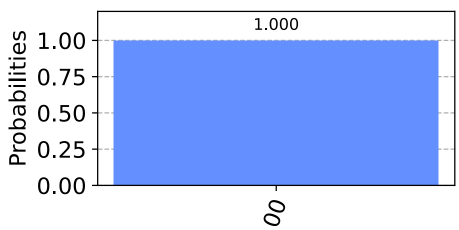
Out[24]:



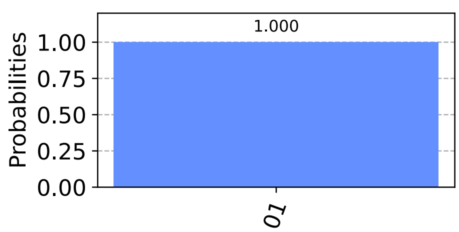
In [25]:

```
## trying all possibilities:
for q0, q1 in [(q0,q1) for q0 in [0,1] for q1 in [0,1]]:
    print(q0,q1)
    qc_ha = QuantumCircuit(4,2)
    # encode inputs in qubits 0 and 1
    if q0 == 1:
        qc_ha.x(0) # For a=0, remove the this line. For a=1, leave it.
    if q1 == 1:
        qc_ha.x(1) # For b=0, remove the this line. For b=1, leave it.
    qc_ha.barrier()
    # use cnots to write the XOR of the inputs on qubit 2
    qc_ha.cx(0,2)
    qc_ha.cx(1,2)
    # use ccx to write the AND of the inputs on qubit 3
    qc_ha.ccx(0,1,3)
    qc_ha.barrier()
    # extract outputs
    qc_ha.measure(2,0) # extract XOR value
    qc_ha.measure(3,1) # extract AND value

    qc_ha.draw()

    counts = sim.run(qc_ha).result().get_counts()
    display(plot_histogram(counts, figsize=(4, 2)))
```

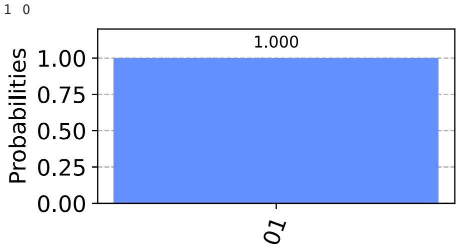
0 0



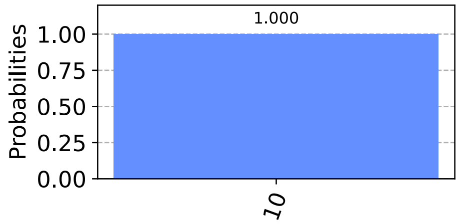
0 1



1 0



1 1



The half-adder contains everything needed for addition!

NOT+CNOT+Toffoli: can add any set of numbers of any size.

In fact, we could even do without the CNOT and NOT (only used to go from $0 \to 1$). **The Toffoli gate is the atom of quantum computation.**

Representing Qubit States

Example of statevector

$$|q_0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{bmatrix}$$

and since $|0\rangle$ and $|1\rangle$ form and orthonormal basis, we can write the statevector on this basis as a **superposition** of $|0\rangle$ and $|1\rangle$:

$$|q_0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle$$

Exploring Qubits with Qiskit

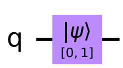
In [26]:
```
from qiskit import QuantumCircuit, Aer
from qiskit.visualization import plot_histogram, plot_bloch_vector
from math import sqrt, pi
```

In [27]:
```
qc = QuantumCircuit(1)# Create a quantum circuit with one qubit
```

Qubits always start on $|0\rangle$, but we can use the `initialize()` method to transform it.

In [28]:
```
qc = QuantumCircuit(1)# Create a quantum circuit with one qubit
initial_state = [0,1] ## notice we give it in the matrix form, as a list
qc.initialize(initial_state, 0) ## what we want the initial state to be, which qubit
qc.draw()
```

Out[28]:



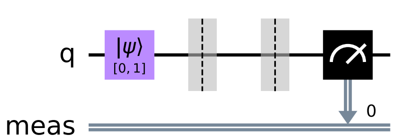
We can them use one of Qiksit simulators to view the resulting state of the qubit:

In [29]:
```
sim = Aer.get_backend('aer_simulator')

qc = QuantumCircuit(1)# Create a quantum circuit with one qubit
initial_state = [0,1] ## notice we give it in the matrix form, as a list
qc.initialize(initial_state, 0) ## what we want the initial state to be, which qubit
qc.save_statevector() ## tell sim to save statevector, but only possible with sim, obvioulsy
result = sim.run(qc).result()  ##.result gets the result of measurement

out_state = result.get_statevector()
print(out_state) ## which should be the state we gave initially
```

```
[0.+0.j 1.+0.j]
```

In [30]:
```
qc.measure_all()
qc.draw()
```

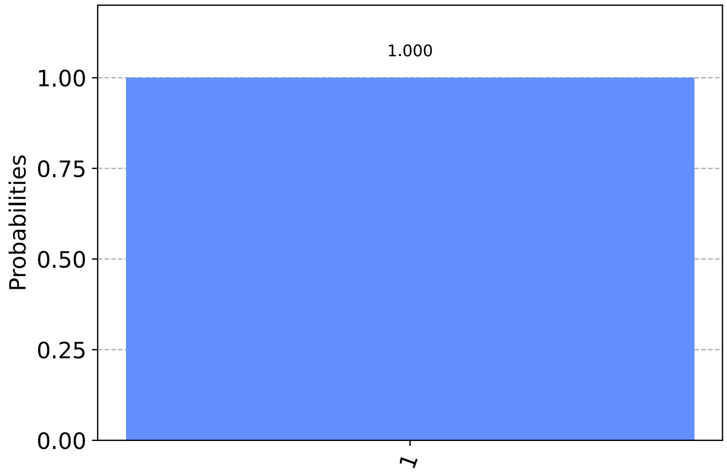
Out[30]:



To see which state we measured, run simulation and `get_counts()`

In [31]:
```
counts = result.get_counts()
plot_histogram(counts)
```

Out[31]:



What if instead we tried the same thing with $|q_o\rangle$?

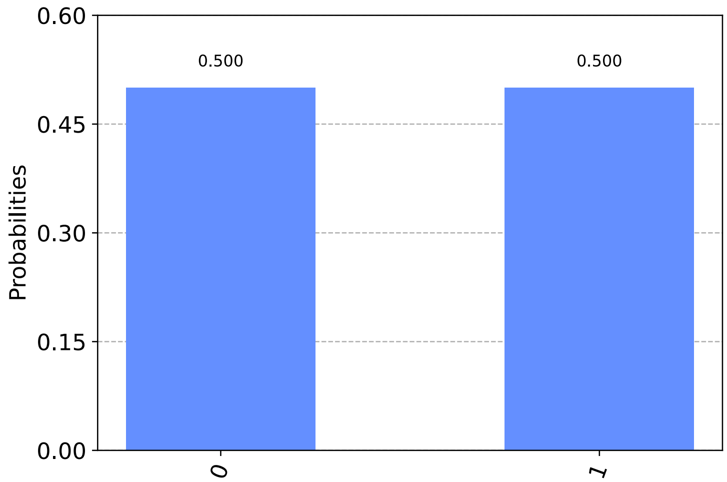
In [32]:
```
qc = QuantumCircuit(1)
initial_state = [1/sqrt(2), 1j/sqrt(2)]
qc.initialize(initial_state, 0)
qc.save_statevector()
state = sim.run(qc).result().get_statevector() ## which is the final statevector?
print(state)
```

```
[0.70710678+0.j         0.        +0.70710678j]
```

But what if we make a measurement?

In [33]:
```
results = sim.run(qc).result().get_counts()
plot_histogram(results)
```

Out[33]:



The Rules of Measurement

Probability of measuring state $|\psi\rangle$ into state $|x\rangle$

$$p(|x\rangle) = |\langle x|\psi\rangle|^2$$

where the inner product of

$$\langle a| = \begin{bmatrix} a_1^* & a_2^* & \cdots & a_n^* \end{bmatrix}, \quad |b\rangle = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix}$$

is given by

$$\langle a|b\rangle = a_1^* b_1 + a_2^* b_2 + \cdots + a_n^* b_n$$

In Qiskit, due to normalization, if one tries to initialize a vector that is not orthonormal, it will give us an error

In [34]:
```
vector = [1,1]
qc.initialize(vector, 0)
```

```
---------------------------------------------------------------------------
QiskitError                               Traceback (most recent call last)
<ipython-input-34-ddc73828b990> in <module>
      1 vector = [1,1]
----> 2 qc.initialize(vector, 0)
```

```
~/opt/anaconda3/lib/python3.8/site-packages/qiskit/extensions/quantum_initializer/initializer.py in initialize(self, params, qubits)
    457
    458     num_qubits = None if not isinstance(params, int) else len(qubits)
--> 459     return self.append(Initialize(params, num_qubits), qubits)
    460
    461

~/opt/anaconda3/lib/python3.8/site-packages/qiskit/extensions/quantum_initializer/initializer.py in __init__(self, params, num_qubits)
     90             # Check if probabilities (amplitudes squared) sum to 1
     91             if not math.isclose(sum(np.absolute(params) ** 2), 1.0, abs_tol=_EPS):
---> 92                 raise QiskitError("Sum of amplitudes-squared does not equal one.")
     93
     94             num_qubits = int(num_qubits)

QiskitError: 'Sum of amplitudes-squared does not equal one.'
```

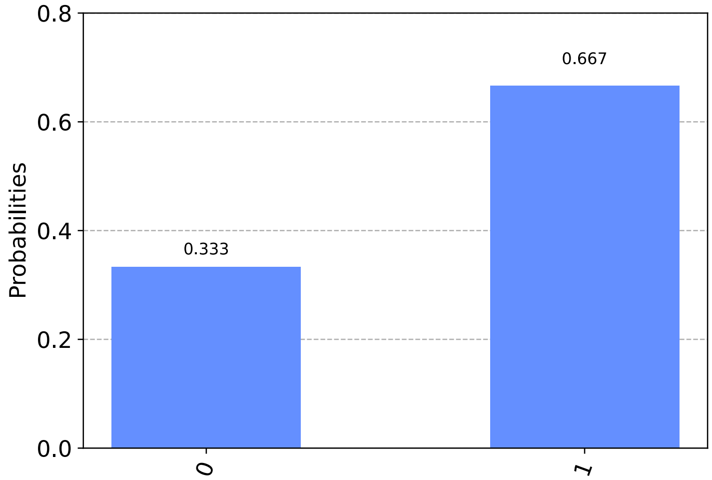
Quick exercises:

1. Create a state vector that will give a $1/3$ probability of measuring $|0\rangle$
2. Create a different state vector that will give the same measurement probabilities.
3. Verify that the probability of measuring $|1\rangle$ for these states is $2/3$.

In [35]:

```
qc = QuantumCircuit(1)
initial_vector_1 = [1/sqrt(3), sqrt(2)/sqrt(3)]
qc.initialize(initial_vector_1, 0)
qc.save_statevector() ## We need to save statevector after initializing it, apparently
results = sim.run(qc).result().get_counts()
plot_histogram(results)
```

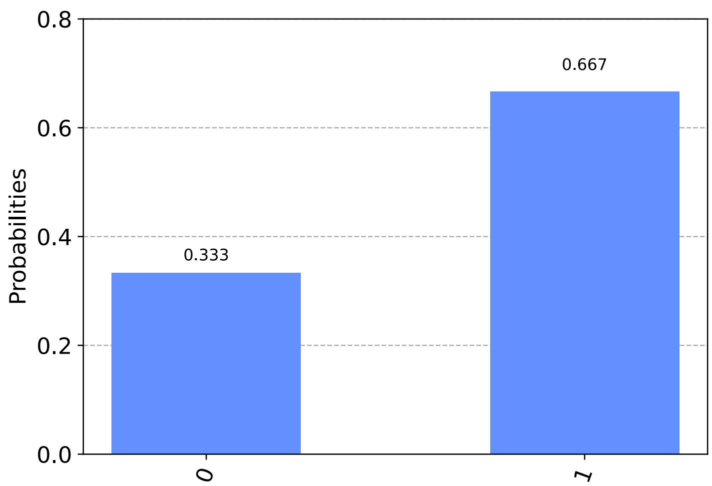
Out[35]:



In [36]:

```
qc = QuantumCircuit(1)
initial_vector_2 = [1j/sqrt(3), -sqrt(2)/sqrt(3)]
qc.initialize(initial_vector_2, 0)
#qc.measure_all()
qc.save_statevector() ## We need to save statevector after initializing it, apparently
results = sim.run(qc).result().get_counts()
plot_histogram(results)
```

Out[36]:



2 And obviously, notice we can also measure in a different basis

3 *Global phases*, typically not physically relevant:

γ such that $|\gamma| = 1$

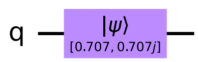
which are different from *relative phases*.

4 Once measured, we know for certain which state the qubit is

In [37]:

```
qc = QuantumCircuit(1)
initial_state = [1/sqrt(2), 1j/sqrt(2)]
qc.initialize(initial_state, 0)
qc.draw()
```

Out[37]:

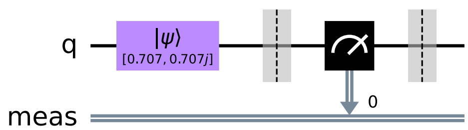


But after the measurement

In [38]:

```
qc = QuantumCircuit(1)
initial_state = [1/sqrt(2), 1j/sqrt(2)]
qc.initialize(initial_state, 0)
qc.measure_all()
qc.save_statevector()
qc.draw()
```

Out[38]:



In [39]:

```
state = sim.run(qc).result().get_statevector()
print(state)
```

```
[1.+0.j 0.+0.j]
```

We can see that writing down a qubit's state requires keeping track of two complex numbers, but when using a real quantum computer we will only ever receive a yes-or-no (0 or 1) answer for each qubit. The output of a 10-qubit quantum computer will look like this:

0110111110

Just 10 bits, no superposition or complex amplitudes. When using a real quantum computer, we cannot see the states of our qubits mid-computation, as this would destroy them! This behaviour is not ideal for learning, so Qiskit provides different quantum simulators: By default, the aer_simulator mimics the execution of a real quantum computer, but will also allow you to peek at quantum states before measurement if we include certain instructions in our circuit. For example, here we have included the instruction `.save_statevector()` , which means we can use `.get_statevector()` on the result of the simulation.

The Bloch Sphere

The Qubit is described by

$$|q\rangle = \alpha|0\rangle + \beta|1\rangle, \ \ \alpha, \beta \text{ in } \mathbb{C}$$

but since we cannot observe global phases, we can remove one of the degrees of freedom and rewrite this as

$$|q\rangle = \alpha|0\rangle + e^{i\phi}\beta|1\rangle, \ \ \alpha, \beta, \phi \text{ in } \mathbb{R}$$

and given the normalization $|\alpha|^2 + |\beta|^2 = 1$, we can use the trigonometric identity $\sin^2 x + \cos^2 x = 1$ to describe α, β in $\mathbb{R}$ in terms of a single angle θ,

$$\alpha = \cos\frac{\theta}{2}, \ \ \beta = \sin\frac{\theta}{2},$$

and describe the qubit using two angles ϕ and θ,

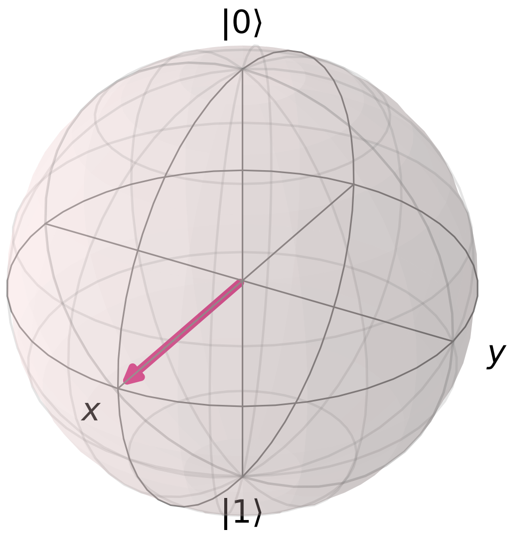
$$|q\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle, \ \ \theta, \phi \text{ in } \mathbb{R}$$

This means we can represent these states in a Bloch sphere:

In [40]:

```
from qiskit_textbook.widgets import plot_bloch_vector_spherical
coords = [pi/2, 0, 1] ## theta, phi, radius
plot_bloch_vector_spherical(coords)
```

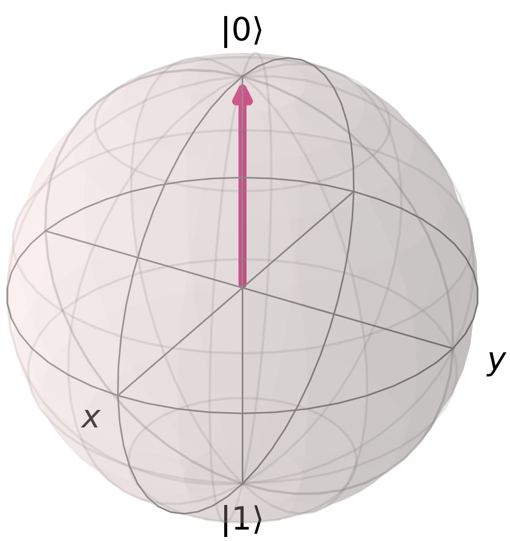
Out[40]:



Do not confuse the *Bloch vector* with the *statevector*. The Bloch vector is a visualisation tool that maps the 2D, complex statevector onto real, 3D space.

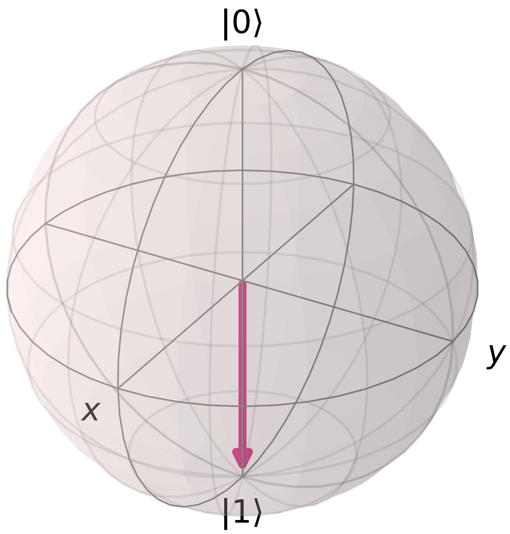
In [41]:
```
# 1: state 0 -> theta = 0, phi = 0

coords = [0, 0, 1]
plot_bloch_vector_spherical(coords)
```

Out[41]:



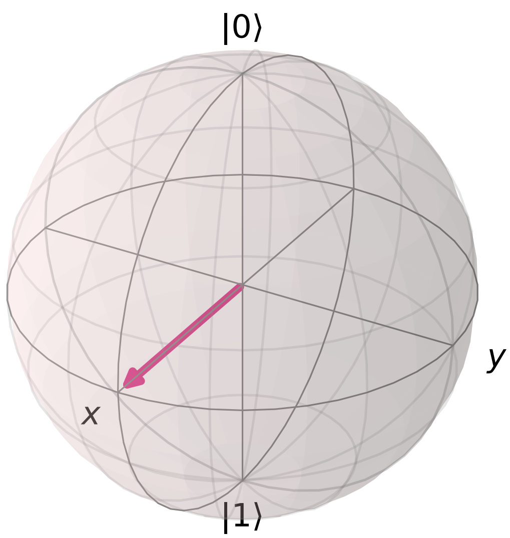
In [42]:
```
# 2: state 1 -> theta = pi, phi = 0
coords = [pi, 0, 1]
plot_bloch_vector_spherical(coords)
```

Out[42]:



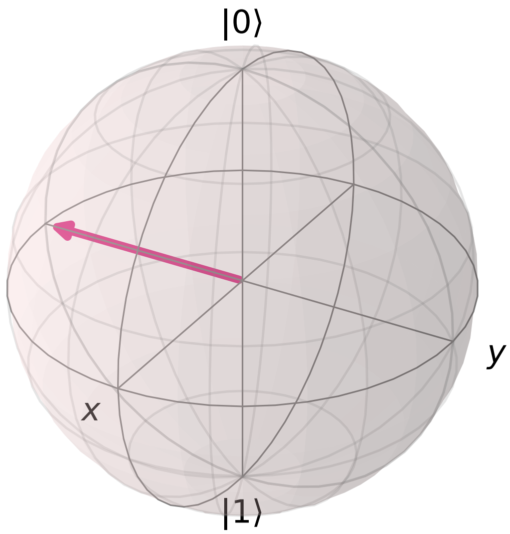
In [43]:
```
# 3: state (1/sqrt(2))(0+1) -> theta = pi/2, phi = 0
coords = [pi/2, 0, 1]
plot_bloch_vector_spherical(coords)
```

Out[43]:



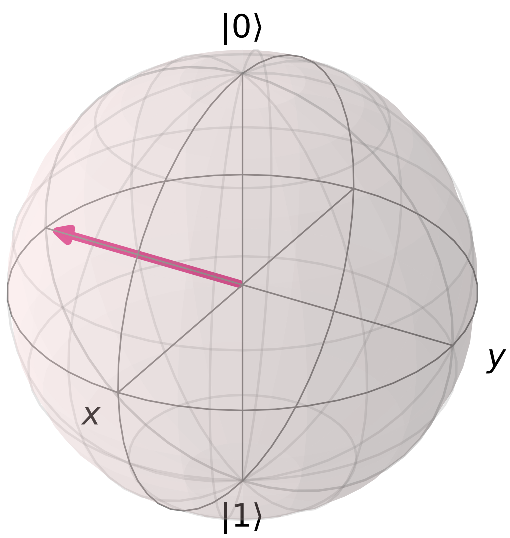
In [44]:
```
# 4: tate (1/sqrt(2))(0-i*1) -> theta = pi/2, phi = 3pi/2
coords = [pi/2, 3*pi/2, 1]
plot_bloch_vector_spherical(coords)
```

Out[44]:



In [45]:
```
# 5: tate (1/sqrt(2))(0-i*1) -> theta = pi/2, phi = 3pi/2
coords = [pi/2, 3*pi/2, 1]
plot_bloch_vector_spherical(coords)
```

Out[45]:



In [ ]:

In [ ]:

In [46]:
```
from qiskit_textbook.widgets import bloch_calc
bloch_calc()
```

In [ ]:

Single Qubit Gates

In this section we will cover gates, the operations that change a qubit between these states.

In [47]:
```
from qiskit import QuantumCircuit, Aer
from math import pi, sqrt
from qiskit.visualization import plot_bloch_multivector, plot_histogram
sim = Aer.get_backend('aer_simulator')
```

The Pauli Gates

The X-Gate

Pauli-X matrix

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

To see the effect of the gate, let's apply to $|0\rangle$ & $|1\rangle$.

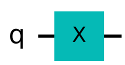
$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

π rotation around the X-axis

In [48]:

```
qc = QuantumCircuit(1)
qc.x(0)
qc.draw()
```

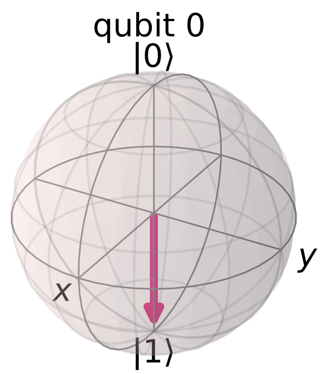
Out[48]:



In [49]:

```
qc.save_statevector()
state = sim.run(qc).result().get_statevector()
plot_bloch_multivector(state)
```

Out[49]:



The Y & Z-Gates

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -i|0\rangle\langle 1| + i|1\rangle\langle 0|$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1|$$

π rotation around y- and z-axis, respectively.

In [50]:

```
from qiskit_textbook.widgets import gate_demo
gate_demo(gates='pauli')
```

In [51]:

```
qc.y(0)
qc.z(0)
qc.draw()
```

Out[51]:

In [ ]:

```
```

Digression: the X, Y & Z Bases

$|0\rangle$ and $|1\rangle$ are the two eigenstatesof the Z-gate. In fact, the *computational basis* ($|0\rangle$ and $|1\rangle$) is often called *Z-basis*. Another popular basis is the *X-basis*, the eigenstates of the X-gate:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

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

The less common are the eigenstates of the Y-gate, $|\circlearrowleft\rangle$ and $|\circlearrowright\rangle$

Quick exercises

1 Verify that $|+\rangle$ and $|-\rangle$ are in fact eigenstates of the X-gate.

2 What eigenvalues do they have?

$$X|+\rangle = |+\rangle \quad (16)$$
$$X|-\rangle = -|-\rangle \quad (17)$$

3 Find the eigenstates of the Y-gate, and their co-ordinates on the Bloch sphere.

Eigenvalues:

$$\begin{bmatrix} -\lambda & -i \\ i & -\lambda \end{bmatrix} \rightarrow \lambda^2 = 1 \rightarrow \lambda = \pm 1$$

Eigenvectors: $$ \lambda = +1, \alpha=1 \rightarrow \begin{bmatrix} -i\beta \ i

\end{bmatrix}

$$\begin{bmatrix} 1 \\ \beta \end{bmatrix}$$

\rightarrow \beta = i \rightarrow |\lambda\rangle_+ = \frac{1}{\sqrt{2}}

$$\begin{bmatrix} 1 \\ i \end{bmatrix}$$

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

$$\lambda = -1, \alpha = 1 \rightarrow \begin{bmatrix} -i\beta \\ i \end{bmatrix} = \begin{bmatrix} -1 \\ -\beta \end{bmatrix} \rightarrow \beta = -i \rightarrow |\lambda\rangle_- = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -i \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$$

In [52]:

```
import numpy as np
from numpy.linalg import eig
a = np.array([[0, 0-1j],
              [0+1j, 0]])
print(a)
w,v=eig(a)

print('E-value:', w, '\n')
print('E-vector', v)
```

```
[[0.+0.j 0.-1.j]
 [0.+1.j 0.+0.j]]
E-value: [ 1.+0.j -1.+0.j]

E-vector [[-0.        -0.70710678j  0.70710678+0.j        ]
 [ 0.70710678+0.j          0.        -0.70710678j]]
```

Using only Paulis we cannot move the qubit away from $|0\rangle$, $|1\rangle$ and we cannot achieve superpositions.

In [ ]:

The Hadamard Gate

H-Gate

$$H = \frac{1}{\sqrt(2)} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Which allows the transformations

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = |+\rangle$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = |-\rangle$$

Which can be thought as a rotation of $\pi/2$ around the Bloch vector $[1, 0, 1]$ (or around the y-axis), transforming the state of the qubit between the X and Z bases.

In [53]:

```
from qiskit_textbook.widgets import gate_demo
gate_demo(gates='pauli+h')
```

Quick exercises:

- #1 Write the H-gate as the outer products of vectors $|0\rangle, |1\rangle, |+\rangle, |-\rangle$

We have that:

$$|0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{18}$$

$$|0\rangle\langle 1| = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{19}$$

$$|1\rangle\langle 0| = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{20}$$

$$|1\rangle\langle 1| = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{21}$$

and

$$|+\rangle\langle +| \quad = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{22}$$

$$|+\rangle\langle -| \quad = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \tag{23}$$

$$|-\rangle\langle +| = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \tag{24}$$

$$|-\rangle\langle -| = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{25}$$

and therefore,

$$H = \frac{1}{\sqrt{2}}[|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1|] = \frac{1}{\sqrt{8}}[|+\rangle\langle +| + |+\rangle\langle -| + |-\rangle\langle +| - |-\rangle\langle -|]$$

- #2 Show that applying the sequence of gates: HZH, to any qubit state is equivalent to applying an X-gate.

$$HZH = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} = X$$

- #3 Find a combination of X, Z and H-gates that is equivalent to a Y-gate (ignoring global phase).

From the properties of Pauli matrices, $ZX = iY$

In [ ]:

```

```

Digression: Measuring in Different Bases

Since Qiskit only allows measuring in the Z-basis, if we want to make measurements in a different basis we must create it using Hadamard gates. To measure on the X-basis, for instance:

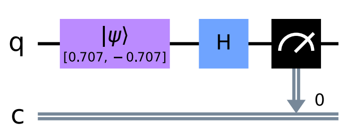
In [54]:

```python
## Create the X-measurememnt function
def x_measurement(qc, qubit, cbit):
    '''
    Measures ''qubit in the X-basis and store the result in 'cbit'
    '''
    qc.h(qubit)  ## transforms from the Z-basis to the X-basis
    qc.measure(qubit, cbit)
    return qc
```

In [55]:

```python
initial_state =[1/sqrt(2), -1/sqrt(2)]
qc = QuantumCircuit(1,1) ##notice the addition of cbit for the measurements
qc.initialize(initial_state, 0)
x_measurement(qc, 0, 0) #measure qubit 0 to classical bit 0
qc.draw()
```

Out[55]:



Remember that $X = HZH$. What this is doing is:

H-gate switches the qubit to X-basis,

$$H|0\rangle = |+\rangle \tag{26}$$

$$H|1\rangle = |-\rangle \tag{27}$$

Z-gate performs a NOT in the X-basis,

$$Z|+\rangle = |-\rangle \tag{28}$$

$$Z|-\rangle = |+\rangle \tag{29}$$

and the final H-gate returns the qubit to the Z-basis.

$$H|+\rangle = |0\rangle \tag{30}$$

$$H|-\rangle = |1\rangle \tag{31}$$

Quick exercises:

1 If we initialize our qubit in the state $|+\rangle$, what is the probability of measuring it in state $|-\rangle$?

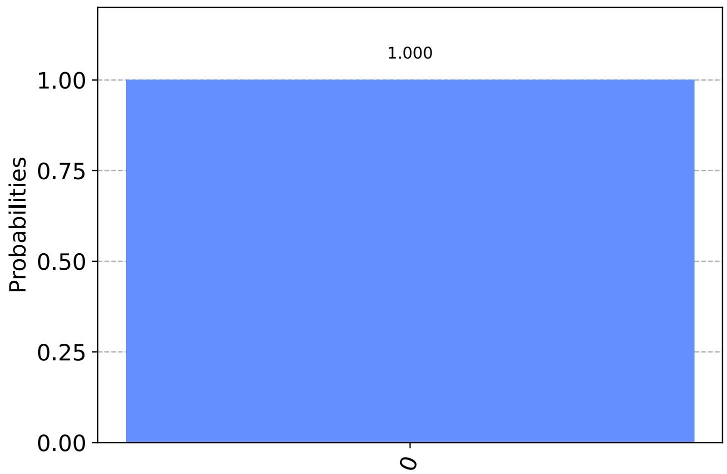
2 Use Qiskit to display the probability of measuring a $|0\rangle$ qubit in the states $|+\rangle$ and $|-\rangle$ (Hint: you might want to use .get_counts() and plot_histogram()).

3 Try to create a function that measures in the Y-basis.

In [56]:

```
# Probability is zero
initial_state =[1/sqrt(2), 1/sqrt(2)]
qc = QuantumCircuit(1,1) ##notice the addition of cbit for the measurements
qc.initialize(initial_state, 0)
x_measurement(qc, 0, 0) #measure qubit 0 to classical bit 0
counts = sim.run(qc).result().get_counts()
plot_histogram(counts)
#qc.draw()
```

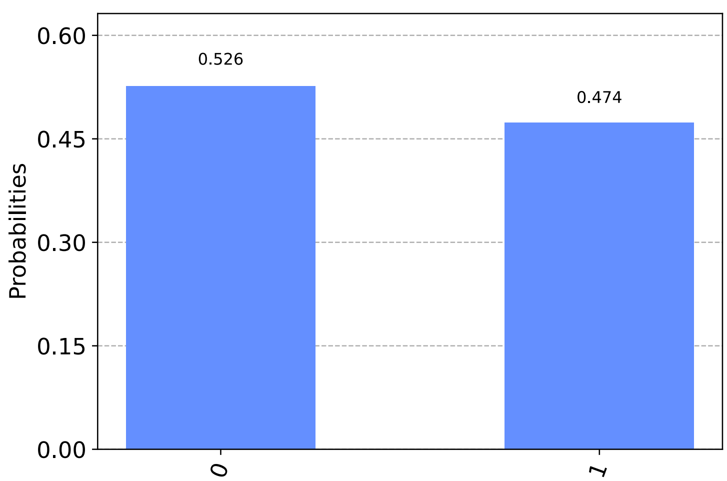
Out[56]:



In [57]:

```
qc = QuantumCircuit(1,1)
x_measurement(qc, 0, 0)
counts = sim.run(qc).result().get_counts()
plot_histogram(counts)

## Probability is 1/2 and 1/2
```

Out[57]:

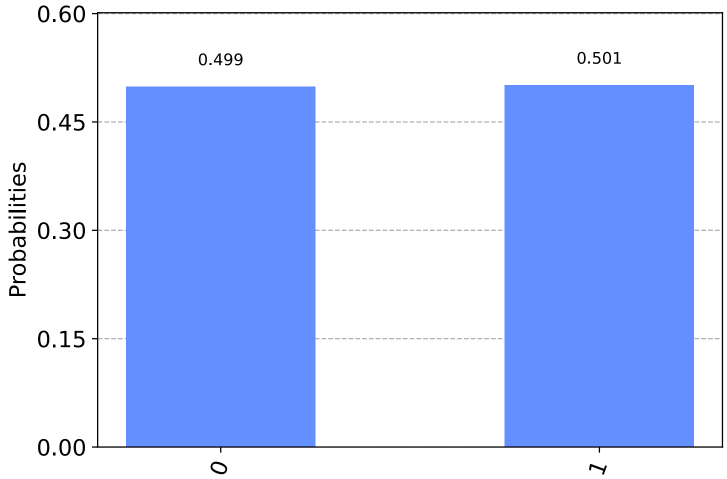


In [58]:

```
def y_measurement(qc,qubit,cbit):
    qc.sdg(qubit)
    qc.h(qubit)
    qc.measure(qubit,cbit)
    return qc

circuit = QuantumCircuit(1,1)
circuit.h(0)
circuit.barrier()


y_measurement(circuit, 0, 0)
```

```
counts = sim.run(qc).result().get_counts()
plot_histogram(counts)
```

Out[58]:



Whatever state our quantum system is in, there is always a measurement that has a deterministic outcome.

In [ ]:

```

```

The P-Gate

P-gate (phase gate) is parameterised by a rotation around the Z-axis.

$$P(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}, \phi \text{ in } \mathbb{R}$$

In [59]:

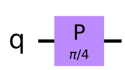
```
from qiskit_textbook.widgets import gate_demo
gate_demo(gates='pauli+h+p')
```

It is defined in Qiskit as

In [60]:

```
qc = QuantumCircuit(1)
qc.p(pi/4, 0)
qc.draw()
```

Out[60]:



Notice the Z-gate is a special case of the P-gate, for $\phi = \pi$. There are also 3 other important special cases: the I, S, and T-Gates

In [ ]:

```

```

The I, S, and T-Gates

The I-Gate

Identity gate

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The reaspm to consider this a gate (which does not do anything on a qubit) is its use in calculations, such as $I = XX$, etc.

Quick exercise: Which are the eigenstates of the I-gate?

All statevectors should be eigenstates of the I-gate, with eigenvalue $= 1$.

The S-Gate

S-gate, also known as $\sqrt{Z}$-gate $\rightarrow \phi = \pi/2$. Importantly, **it is not its own inverse**, i.e.,

$$SS \neq I$$

.

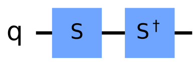
$S^\dagger$-gate, also known as $\sqrt{Z}^\dagger$-gate $\rightarrow \phi = -\pi/2$

$$SS|q\rangle = Z|q\rangle$$

the reason for the $\sqrt{Z}$-gate.

In [61]:
```
qc = QuantumCircuit(1)
qc.s(0)
qc.sdg(0)
qc.draw()
```

Out[61]:



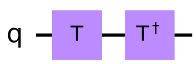
The T-Gate

T-gate, sometimes known as $\sqrt[4]{Z}$-gate, $\rightarrow \phi = \pi/4$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{bmatrix}, \quad T\dagger = \begin{bmatrix} 1 & 0 \\ 0 & \exp(-i\pi/4) \end{bmatrix}$$

In [62]:
```
qc = QuantumCircuit(1)
qc.t(0)
qc.tdg(0)
qc.draw()
```

Out[62]:



In [63]:
```
from qiskit_textbook.widgets import gate_demo
gate_demo()
```

The General U-Gate

U-gate, most general of the single qubit gates can be parameterised by 3-parameters:

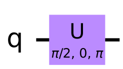
$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & e^{-i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{bmatrix}$$

where the specific cases become

$$U(\frac{\pi}{2}, 0, \pi) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H, \quad U(0, 0, \lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} = P$$

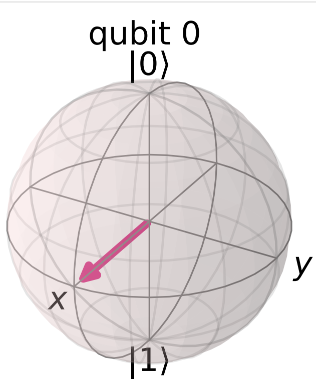
In [64]:
```
qc = QuantumCircuit(1)
qc.u(pi/2, 0, pi, 0)## 3 parameters and qubit to be applied to
qc.draw()
```

Out[64]:



In [65]:
```
qc.save_statevector()
state = sim.run(qc).result().get_statevector()
plot_bloch_multivector(state)
```

Out[65]:



Qiskit also provides the X equivalent of S gates, SX-gate and SXdg-gate, which do a quarter turn around the X-axis.

Before running on real IBM quantum hardware, all single qubit ops are compiled down to I, X, SX, and R_z, whihc are called *physical gates*.

The Case for Quantum

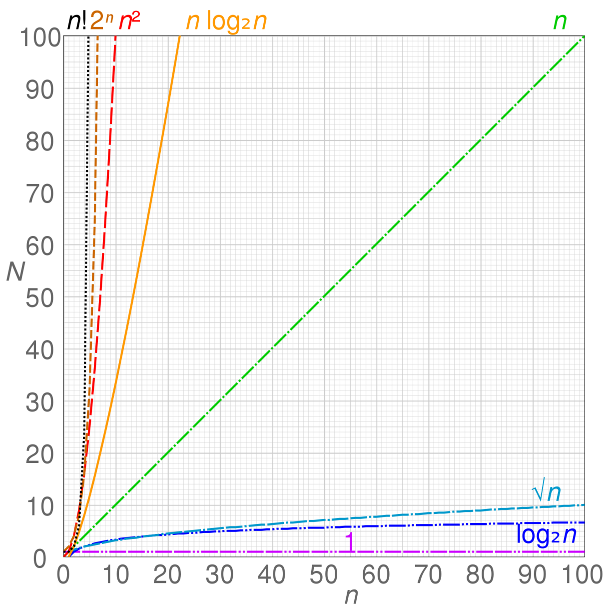
Complexity of adding

$$n \leq c(n) \leq 2n \longrightarrow c(n) = O(n)$$

Big O notation

For functions $f(x)$ and $g(x)$ and parameter x, the statement $f(x) = O(g(x))$ means that $\exists$ some finite $M > 0$ and x_0 such that

$$f(x) \leq Mg(x), \forall x > x_0$$



Sclaing as functions of the input size n

Complexity Theory

Multiplication: $O(n^2)$

Factorization: $O(e^{n^{1/3}})$

Searching database:$O(n)$

Formally, defining the complexity of an algorithm depends on the exact theoretical model for computation we are using. Each model has a set of basic operations, known as primitive operations, with which any algorithm can be expressed.

For **Boolean circuits**, as we considered in the first section, the primitive operations are the logic gates.

For **Turing machines**, a hypothetical form of computer proposed by Alan Turing, we imagine a device stepping through and manipulating information stored on a tape.

The **RAM model** has a more complex set of primitive operations and acts as an idealized form of the computers we use every day.

All these are models of digital computation, based on discretized manipulations of discrete values. Different as they may seem from each other, *it turns out that it is very easy for each of them to simulate the others*.

Beyond digital computers

Digital computers: discrete values (0-1, e.g.). Can detect and correct errors relatively easy.

Analog computers: precise manipulation of continuoulsy varying parameters. Issue: arbitrary precision.

Ideally: robusteness of digital with subtle manipulations of analog.

Quantum computing is the only known technology exponentially fatster than classical computers for certain tasks.

When to use QC

Novel algorithms: One way in which this can be done is when we have some function for which we want to determine a global property. For example, if we want to find the value of some parameter x for which some function $f(x)$ is a minimum, or the period of the function if $f(x)$ is periodic.

Superposition of states to induce quantum interference and reveal global property.

Ex.

Grover's search algorithm: $O(n)$ to $O(n^{1/2})$.

Shor's factorization algorithm: $O(e^{n^{1/3}})$ to $O(n^3)$

Solve quantum problems: dimensional need.

Particularly promising are those problems for which classical algorithms face inherent scaling limits and which do not require a large classical dataset to be loaded.

See https://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf

In [ ]:

In [ ]:

In [ ]:

In [66]:
```
import qiskit.tools.jupyter
%qiskit_version_table
```

```
/Users/ufranca/opt/anaconda3/lib/python3.8/site-packages/qiskit/aqua/__init__.py:86: DeprecationWarning: The package qiskit.aqua is deprecate
d. It was moved/refactored to qiskit-terra For more information see <https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide
>
  warn_package('aqua', 'qiskit-terra')
```

Version Information

Qiskit Software	Version
qiskit-terra	0.18.3
qiskit-aer	0.9.0
qiskit-ignis	0.6.0
qiskit-ibmq-provider	0.16.0
qiskit-aqua	0.9.5
qiskit	0.30.1
qiskit-nature	0.2.2
qiskit-finance	0.2.1
qiskit-optimization	0.2.3
qiskit-machine-learning	0.2.1
System information	
Python	3.8.8 (default, Apr 13 2021, 12:59:45) [Clang 10.0.0]
OS	Darwin
CPUs	8
Memory (Gb)	16.0
	Sat Dec 18 12:32:31 2021 EST

In [ ]: