Hands On: I Primi Circuiti con Qiskit e Python

Giuseppe Bisicchia

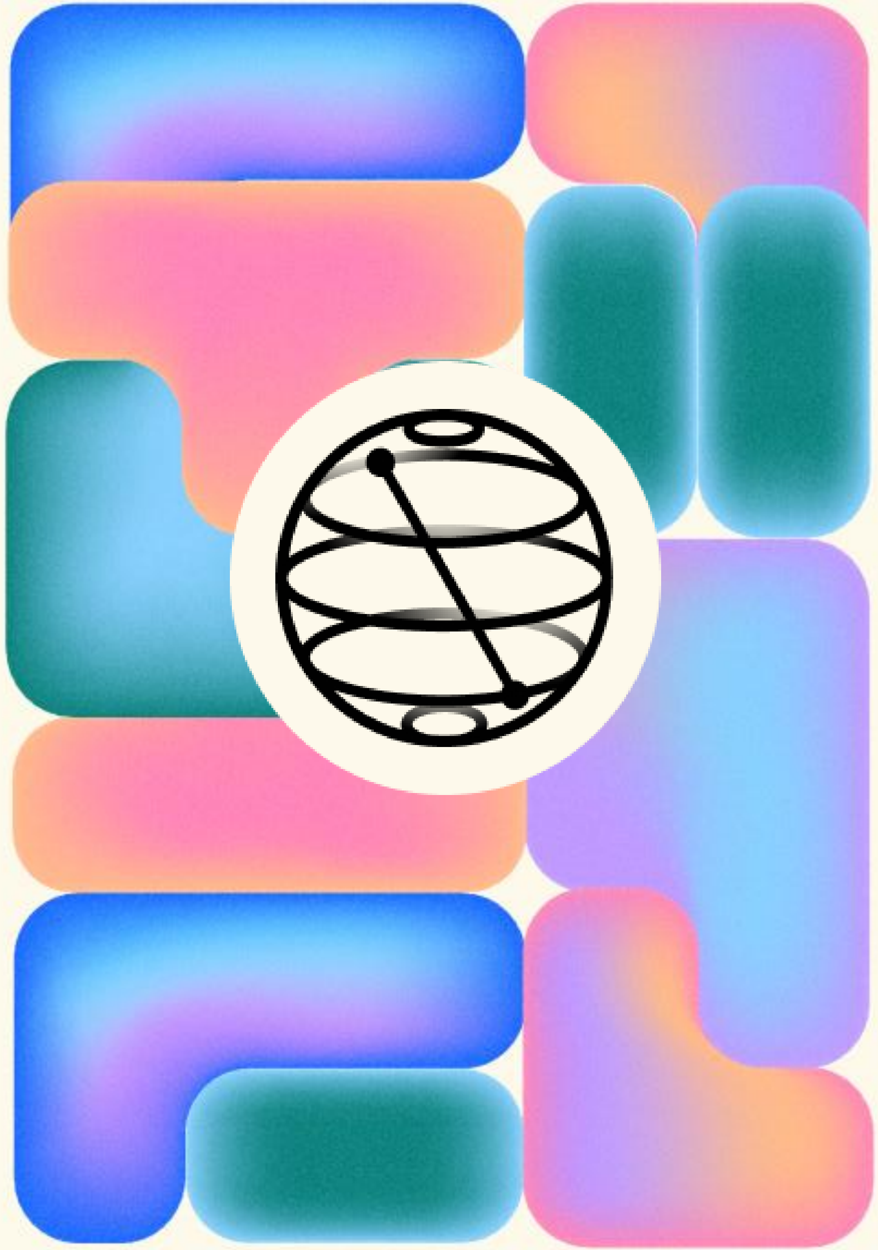
Dottorando in Quantum Software Engineering

Università di Pisa

E-mail: giuseppe.bisicchia@phd.unipi.it

Website: gbisi.github.io

Qiskit | Fall Fest 2023



Cosa faremo?

Impareremo a realizzare, manipolare ed eseguire circuiti quantistici in ***Python***, tramite il Software Development Kit (SDK) ***Qiskit***, su macchine e simulatori *IBM*.

Nel mentre, risponderemo alle seguenti domande:

- *Come si programma un circuito quantistico?*
- *Come posso eseguire un circuito?*
- *Come si interpretano i risultati?*

Per questa lezione lavoreremo su: https://quantum-computing.ibm.com/lab

A (qu)bit of theory

- Il Circuito Quantistico (*QuantumCircuit*) è l'unità fondamentale di Qiskit.
- In Qiskit un programma è diviso in (almeno) due fasi:
  - **Build**: il Quantum Circuit viene creato;
  - **Execute**: il Quantum Circuit viene eseguito.
    - Un circuito può essere eseguito su un simulatore locale, nel Cloud o direttamente su un Computer Quantistico.

Creiamo il nostro primo Quantum Circuit!

```
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```

Importiamo la libreria Qiskit e in particolare la classe QuantumCircuit

Creiamo il nostro primo Quantum Circuit!

```
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```

Creiamo un QuantumCircuit chiamato *qc* avente 3 qubit e 3 bit classici

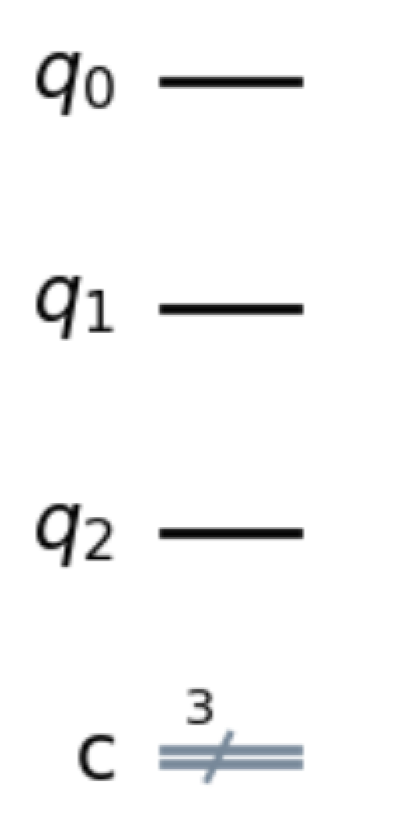
Creiamo il nostro primo Quantum Circuit!

```
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```

Visualizziamo graficamente il circuito

Creiamo il nostro primo Quantum Circuit!

```python
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```



Question Time!

```
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```

Perchè ci servono i bit classici?

Perchè ci servono i bit classici?

```
from qiskit import QuantumCircuit

# Create quantum circuit with 3 qubits and 3 classical bits
# (we'll explain why we need the classical bits later)
qc = QuantumCircuit(3,3)

# return a drawing of the circuit
qc.draw()
```

Il risultato di una misura su un qubit deve essere memorizzato in un bit classico!

Come si misurano i qubit?

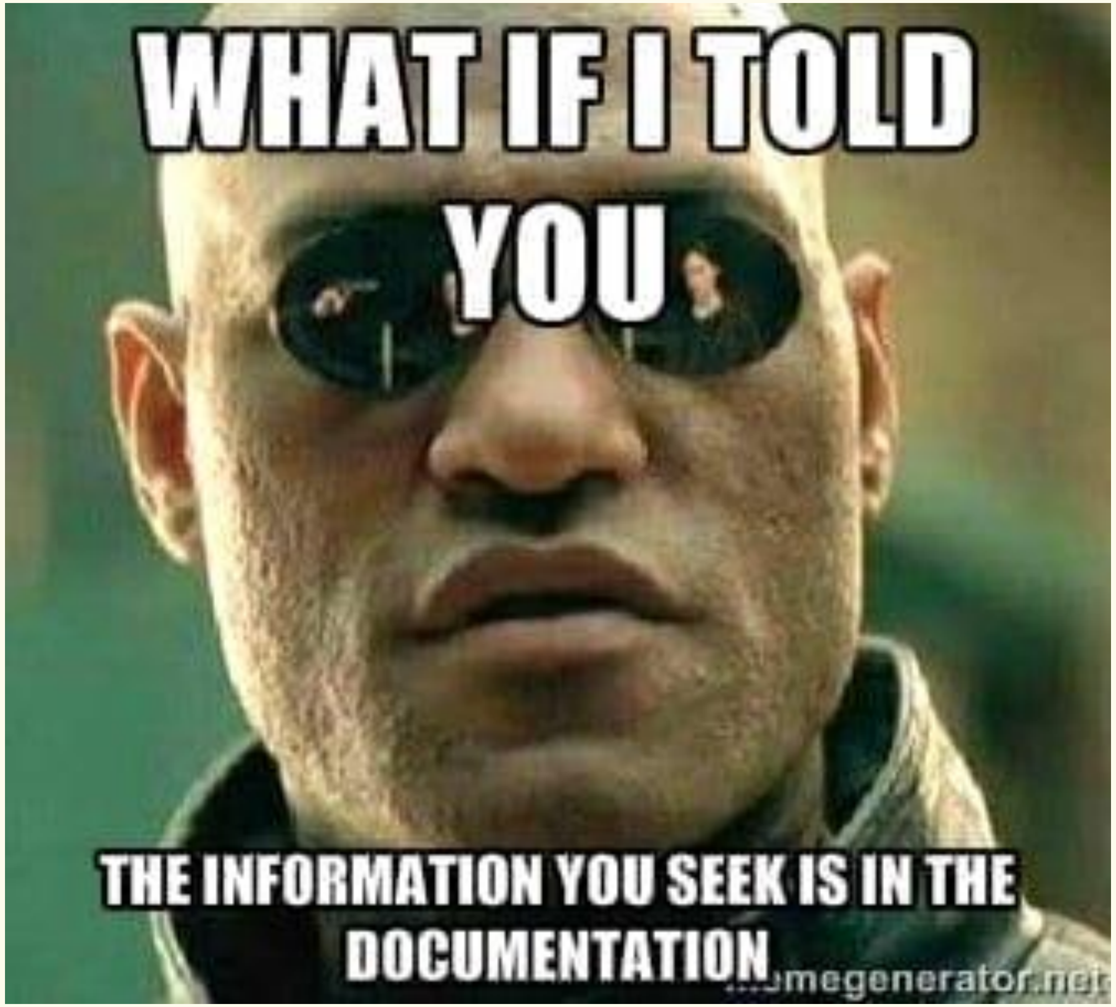
```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")
```

Question Time!

```python
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")
```

Che cosa indicano le due liste di numeri?

Che cosa indicano le due liste di numeri?



Che cosa indicano le due liste di numeri?

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")
```

`measure(qubit, cbit)` [source]

Measure a quantum bit (`qubit`) in the Z basis into a classical bit (`cbit`).

When a quantum state is measured, a qubit is projected in the computational (Pauli Z) basis to either $|0\rangle$ or $|1\rangle$. The classical bit `cbit` indicates the result of that projection as a `0` or a `1` respectively. This operation is non-reversible.

Parameters:

- **qubit** (*Qubit* | *QuantumRegister* | *int* | *slice* | *Sequence[Qubit* | *int]*) – qubit(s) to measure.
- **cbit** (*Clbit* | *ClassicalRegister* | *int* | *slice* | *Sequence[Clbit* | *int]*) – classical bit(s) to place the measurement result(s) in.

Returns:

handle to the added instructions.

Return type:

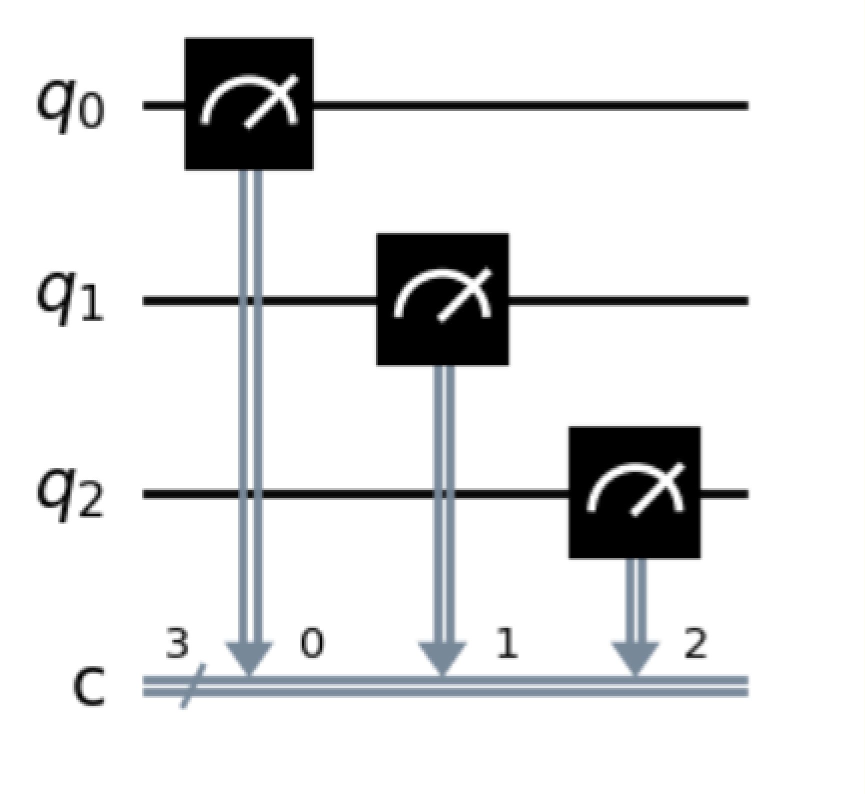
qiskit.circuit.InstructionSet

Raises:

CircuitError – if arguments have bad format.

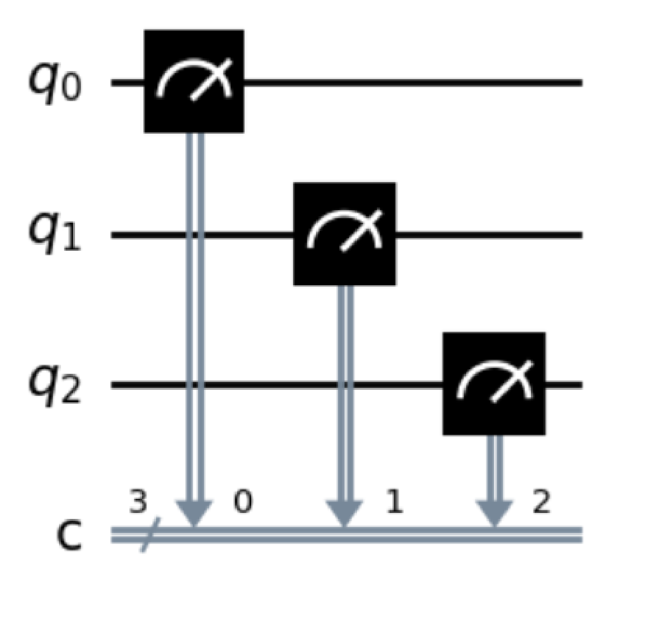
Come si misurano i qubit?

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")
```



Come si misurano i qubit? – Esercizio!

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")
```

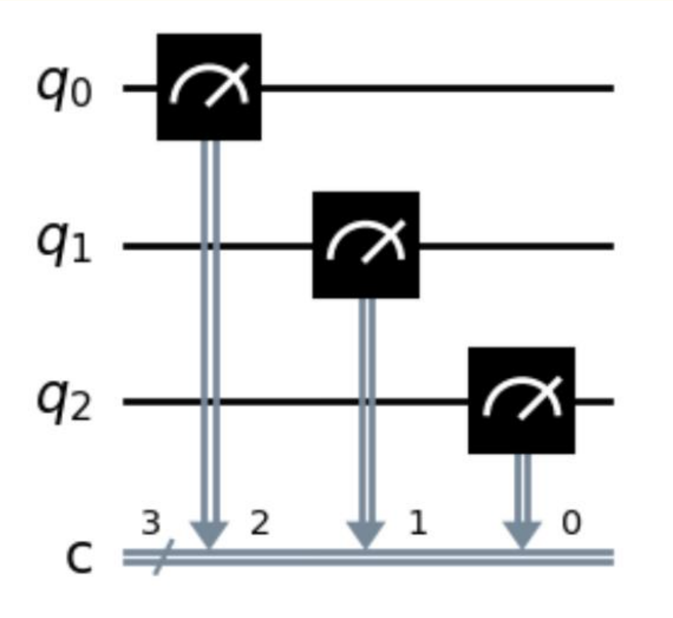


Provate a cambiare la corrispondenza tra qubit e bit classici!

Esempio: il qubit 2 viene memorizzato nel bit 0 e il qubit 0 nel bit 2

Come si misurano i qubit? – Esercizio!

```python
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [2,1,0])

qc.draw(output="mpl")
```



Provate a cambiare la corrispondenza tra qubit e bit classici!

Esempio: il qubit 2 viene memorizzato nel bit 0 e il qubit 0 nel bit 2

Eseguiamo il circuito!

```python
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()  # get the results

result.get_counts()    # interpret the results as a "counts" dictionary
```

Eseguiamo il circuito!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```

Importiamo il simulatore *Aer*

Eseguiamo il circuito!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)      # run the experiment
result = job.result()  # get the results

result.get_counts()    # interpret the results as a "counts" dictionary
```

Creiamo un simulatore *Aer* chiamato *sim*

Eseguiamo il circuito!

```python
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)          # run the experiment
result = job.result()  # get the results

result.get_counts()      # interpret the results as a "counts" dictionary
```

Eseguiamo il circuito *qc* nel simulatore

Eseguiamo il circuito!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)        # run the experiment
result = job.result()    # get the results

result.get_counts()      # interpret the results as a "counts" dictionary
```

qc **può essere qualsiasi circuito! Il programma è indipendente dal particolare circuito che vogliamo eseguire!!**

Eseguiamo il circuito!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)         # run the experiment
result = job.result()     # get the results

result.get_counts()       # interpret the results as a "counts" dictionary
```

Memorizziamo i risultati dell'esecuzione in una variabile chiamata *result*

Eseguiamo il circuito!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```

Visualizziamo i risultati

Question Time!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)         # run the experiment
result = job.result()     # get the results

result.get_counts()       # interpret the results as a "counts" dictionary
```

```
{'000': 1024}
```

Qual è il significato dell’output?

Qual è il significato dell'output?

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```

```
{'000': 1024}
```

L'output rappresenta una serie di **coppie *chiave-valore***. Ogni **chiave** (a sinistra) rappresenta un possible **stato dei bit classici** al termine dell'esecuzione del circuito. Il **valore** (a destra) rappresenta **quante volte è stato osservato quello stato**. Nell'esempio lo stato 000 è stato misurato 1024 volte

Question Time!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```

```
{'000': 1024}
```

Perchè misuriamo solo 000?

Perchè misuriamo solo 000?

```python
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```

```
{'000': 1024}
```

All’inizio dell’esecuzione di ogni circuito i qubit vengono **inizializzati** nello stato **0**

Question Time!

```
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)        # run the experiment
result = job.result()    # get the results

result.get_counts()      # interpret the results as a "counts" dictionary
```

Cosa accade se proviamo a stampare solo *result?*

Cosa accade se proviamo a stampare solo *result*?

```python
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()     # interpret the results as a "counts" dictionary
```


```
Result(backend_name='aer_simulator', backend_version='0.12.2', qobj_id='', job_id='b307155a-6f88-4f3e-9663-f2e481c1113e', success=True, results=[ExperimentResult(shots=1024, success=True, meas_level=2, data=ExperimentResultData(counts={'0x0': 1024}), header=QobjExperimentHeader(creg_sizes=[['c', 3]], global_phase=0.0, memory_slots=3, n_qubits=3, name='circuit-121', qreg_sizes=[['q', 3]], metadata={}), status=DONE, seed_simulator=4243787058, metadata={'noise': 'ideal', 'batched_shots_optimization': False, 'measure_sampling': True, 'parallel_shots': 1, 'remapped_qubits': False, 'active_input_qubits': [0, 1, 2], 'num_clbits': 3, 'parallel_state_update': 8, 'sample_measure_time': 0.002469919, 'num_qubits': 3, 'device': 'CPU', 'input_qubit_map': [[2, 2], [1, 1], [0, 0]], 'method': 'stabilizer', 'fusion': {'enabled': False}}, time_taken=0.007558453)], date=2023-11-04T13:05:24.112465, status=COMPLETED, header=None, metadata={'time_taken_execute': 0.007652782, 'mpi_rank': 0, 'num_mpi_processes': 1, 'max_gpu_memory_mb': 0, 'max_memory_mb': 31890, 'parallel_experiments': 1, 'num_processes_per_experiments': 1, 'omp_enabled': True}, time_taken=0.009244203567504883)
```

Tutto insieme!

```
# -------------
# --- BUILD ---
# -------------
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)

# measure all the qubits
qc.measure([0,1,2], [0,1,2])

qc.draw(output="mpl")

# ---------------
# --- EXECUTE ---
# ---------------
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)      # run the experiment
result = job.result()  # get the results

result.get_counts()
```

Tutto insieme!

```
# -------------
# --- BUILD ---
# -------------
from qiskit import QuantumCircuit

qc = ...

# ---------------
# --- EXECUTE ---
# ---------------
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()   # get the results

result.get_counts()
```

Tutto insieme!

```
# ------------
# --- BUILD ---
# ------------
from qiskit import QuantumCircuit

qc = ...

# --------------
# --- EXECUTE ---
# --------------
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc)       # run the experiment
result = job.result()  # get the results

result.get_counts()
```

Esercizio!

Create un circuito con 6 qubit e 3 bit classici.

Ogni *qubit* dispari deve essere misurato e memorizzato in un bit classico.

Eseguite il circuito sul simulatore Aer per 512 shots.

Hint1: Consultate la documentazione ufficiale!
https://qiskit.org/documentation/apidoc/index.html

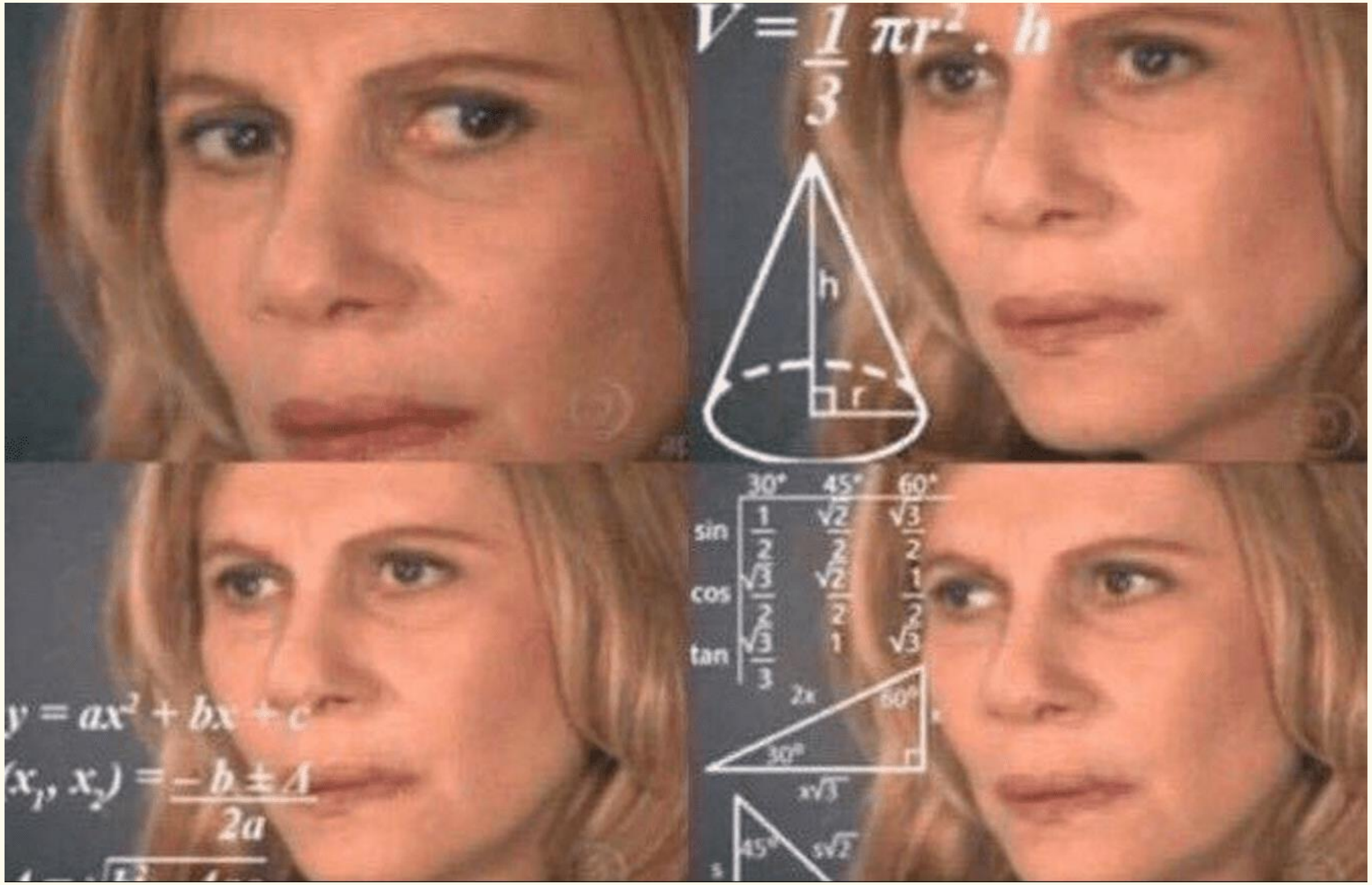
Hint2: Utilizzate il *template* della slide precedente!

Esercizio!

```
# -------------
# --- BUILD ---
# -------------
from qiskit import QuantumCircuit

qc = QuantumCircuit(6,3)

# measure the odd qubits
qc.measure([1,3,5], [0,1,2])

qc.draw(output="mpl")

# ---------------
# --- EXECUTE ---
# ---------------
from qiskit.providers.aer import AerSimulator

# make a new simulator object
sim = AerSimulator()

job = sim.run(qc, shots=512)  # run the experiment
result = job.result()  # get the results

result.get_counts()
```

```
{'000': 512}
```

Come si esegue un circuito in un vero Quantum Computer?



Come si esegue un circuito in un vero Quantum Computer?

```python
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

service.backends()
```

```
[<IBMBackend('ibmq_qasm_simulator')>,
 <IBMBackend('simulator_mps')>,
 <IBMBackend('ibm_perth')>,
 <IBMBackend('ibm_lagos')>,
 <IBMBackend('ibm_brisbane')>,
 <IBMBackend('simulator_extended_stabilizer')>,
 <IBMBackend('simulator_statevector')>,
 <IBMBackend('simulator_stabilizer')>,
 <IBMBackend('ibm_nairobi')>]
```

Come si esegue un circuito in un vero Quantum Computer?

```
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options
```

Importiamo tutto il necessario

```
# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

service.backends()
```

```
[<IBMBackend('ibmq_qasm_simulator')>,
 <IBMBackend('simulator_mps')>,
 <IBMBackend('ibm_perth')>,
 <IBMBackend('ibm_lagos')>,
 <IBMBackend('ibm_brisbane')>,
 <IBMBackend('simulator_extended_stabilizer')>,
 <IBMBackend('simulator_statevector')>,
 <IBMBackend('simulator_stabilizer')>,
 <IBMBackend('ibm_nairobi')>]
```

Come si esegue un circuito in un vero Quantum Computer?

```
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

service.backends()
```

```
[<IBMBackend('ibmq_qasm_simulator')>,
 <IBMBackend('simulator_mps')>,
 <IBMBackend('ibm_perth')>,
 <IBMBackend('ibm_lagos')>,
 <IBMBackend('ibm_brisbane')>,
 <IBMBackend('simulator_extended_stabilizer')>,
 <IBMBackend('simulator_statevector')>,
 <IBMBackend('simulator_stabilizer')>,
 <IBMBackend('ibm_nairobi')>]
```

Colleghiamoci al nostro account *IBM Quantum*

Come si esegue un circuito in un vero Quantum Computer?

```
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

service.backends()
```

```
[<IBMBackend('ibmq_qasm_simulator')>,
 <IBMBackend('simulator_mps')>,
 <IBMBackend('ibm_perth')>,
 <IBMBackend('ibm_lagos')>,
 <IBMBackend('ibm_brisbane')>,
 <IBMBackend('simulator_extended_stabilizer')>,
 <IBMBackend('simulator_statevector')>,
 <IBMBackend('simulator_stabilizer')>,
 <IBMBackend('ibm_nairobi')>]
```

Visualizziamo i *backend* disponibili

Come si esegue un circuito in un vero Quantum Computer?

```
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)  # run the experiment
result = job.result()  # get the results
result
```

Come si esegue un circuito in un vero Quantum Computer?

```python
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)  # run the experiment
result = job.result()  # get the results
result.quasi_dists
```

```python
service.backends()
```

```
[<IBMBackend('ibmq_qasm_simulator')>,
 <IBMBackend('simulator_mps')>,
 <IBMBackend('ibm_perth')>,
 <IBMBackend('ibm_lagos')>,
 <IBMBackend('ibm_brisbane')>,
 <IBMBackend('simulator_extended_stabilizer')>,
 <IBMBackend('simulator_statevector')>,
 <IBMBackend('simulator_stabilizer')>,
 <IBMBackend('ibm_nairobi')>]
```

Come si esegue un circuito in un vero Quantum Computer?

```python
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)   # run the experiment
result = job.result()   # get the results
result.quasi_dists
```

```python
sim = AerSimulator()

job = sim.run(qc)
result = job.result()

result.get_counts()
```

Come si esegue un circuito in un vero Quantum Computer?

```
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)  # run the experiment
result = job.result()  # get the results
result.quasi_dists
```

```
[{0: 1.0}]
```

Come si esegue un circuito in un vero Quantum Computer?

```
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)  # run the experiment
result = job.result()  # get the results
result.quasi_dists
```

```
[{0: 1.0}]
```

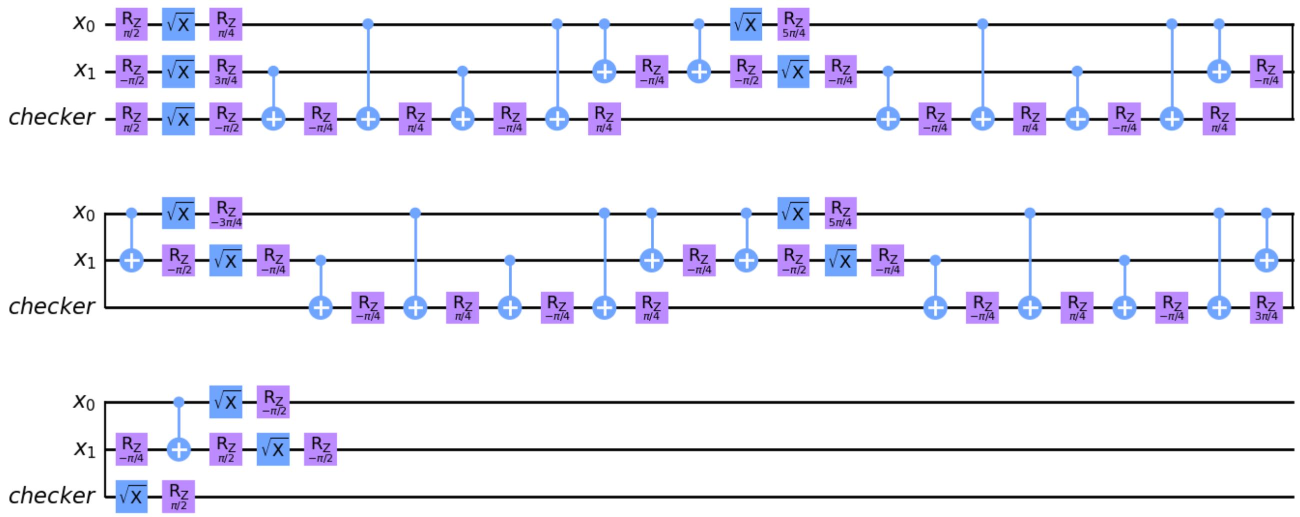
L'output è sempre una serie di coppie chiave-valore. Ma in questo caso la chiave non è direttamente la stringa in binario ma in decimale!

Come si esegue un circuito in un vero Quantum Computer?

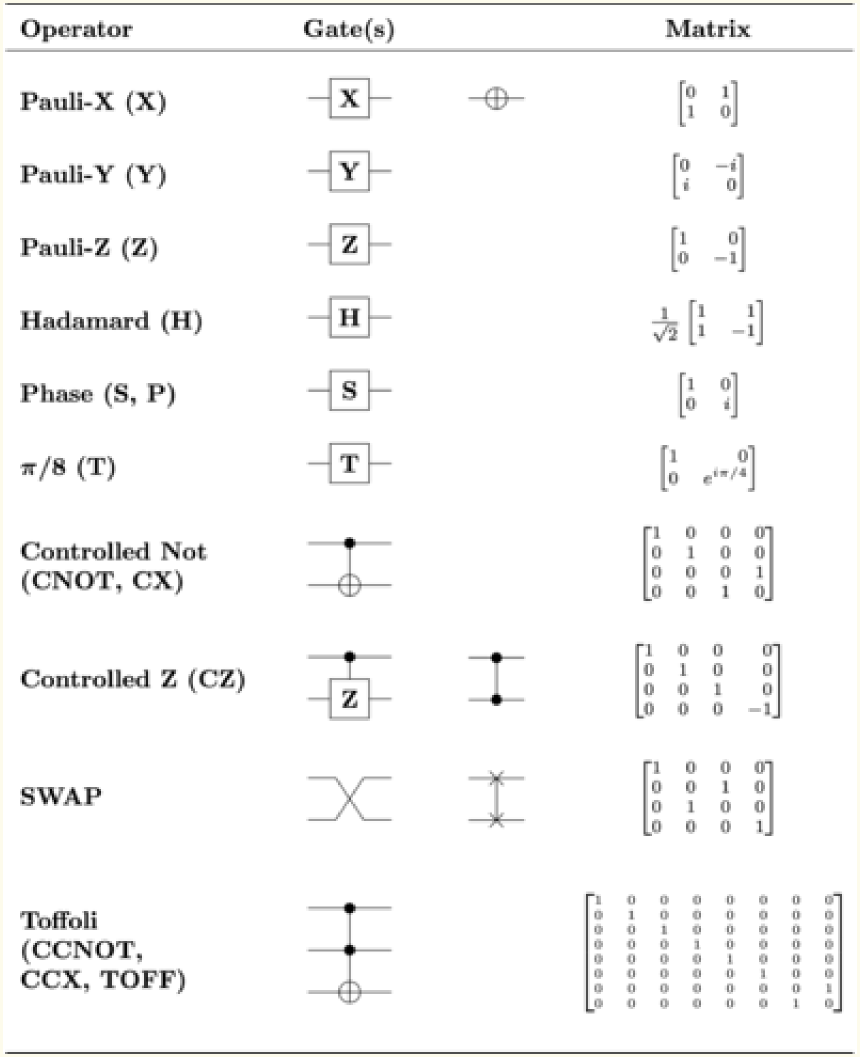
```
# Retrieving a Quantum Backend
backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc)  # run the experiment
result = job.result()  # get the results
result
```

```
SamplerResult(quasi_dists=[{0: 1.0}], metadata=[{'shots': 4000, 'circuit_metadata': {}}])
```

Quantum Gates!



Quantum Gates!



Operator	Gate(s)		Matrix
Pauli-X (X)	$\boxed{X}$	$\oplus$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)	$\boxed{Y}$		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)	$\boxed{Z}$		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)	$\boxed{H}$		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)	$\boxed{S}$		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)	$\boxed{T}$		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)			$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	$\boxed{Z}$		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP			$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)			$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Wikipedia.org

Quantum Gates!

Quantum Gates disponibili su Qiskit e relativa sintassi:

https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html

Quantum Gates!

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

# We start by flipping the first qubit, which is qubit 0, using an X gate
qc.x(0)

# Next we add an H gate on qubit 0, putting this qubit in superposition.
qc.h(0)

# Finally we add a CX (CNOT) gate on qubit 0 and qubit 1
# This entangles the two qubits together
qc.cx(0, 1)
```

Quantum Gates!

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

# We start by flipping the first qubit, which is qubit 0, using an X gate
qc.x(0)

# Next we add an H gate on qubit 0, putting this qubit in superposition.
qc.h(0)

# Finally we add a CX (CNOT) gate on qubit 0 and qubit 1
# This entangles the two qubits together
qc.cx(0, 1)
```

Quantum Gates!

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

# We start by flipping the first qubit, which is qubit 0, using an X gate
qc.x(0)

# Next we add an H gate on qubit 0, putting this qubit in superposition.
qc.h(0)

# Finally we add a CX (CNOT) gate on qubit 0 and qubit 1
# This entangles the two qubits together
qc.cx(0, 1)
```

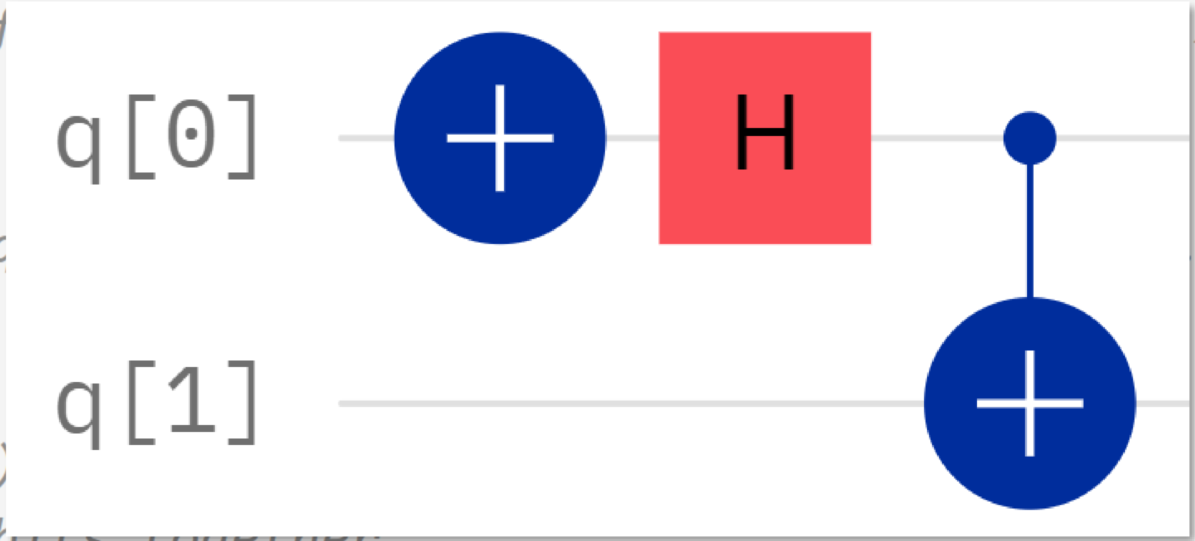
Non abbiamo bit classici questa volta! (Non possiamo misurare)

Quantum Gates!

```python
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

# We start by flipping the first qubit, which is qubit 0, using an X gate
qc.x(0)

# Next we add an H gate on qubit 0, putting this qubit in superposition.
qc.h(0)

# Finally we add a CX (CNOT) gate on qubit 0 and qubit 1
# This entangles the two qubits together
qc.cx(0, 1)
```

Quantum Gates!

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

# We start by flipping the f                                 X gate
qc.x(0)

# Next we add an H gate on q                                 tion.
qc.h(0)

# Finally we add a CX (CNOT)
# This entangles the two qubits together
qc.cx(0, 1)
```

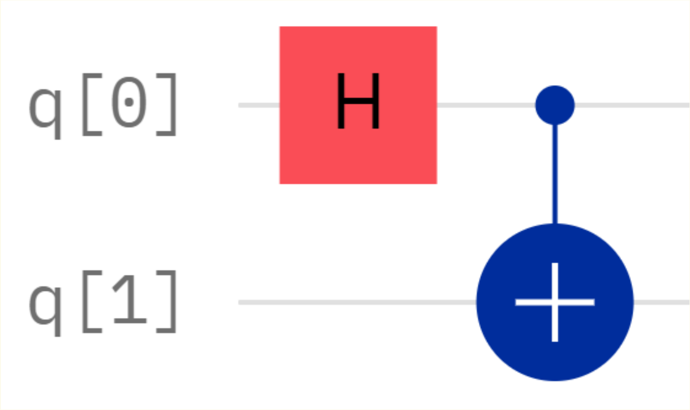


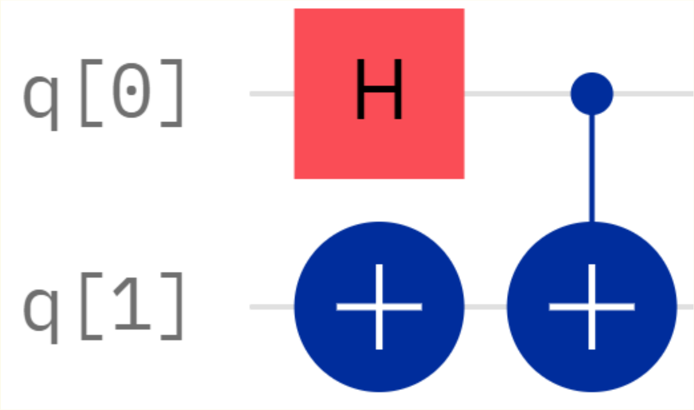
Esercizio!

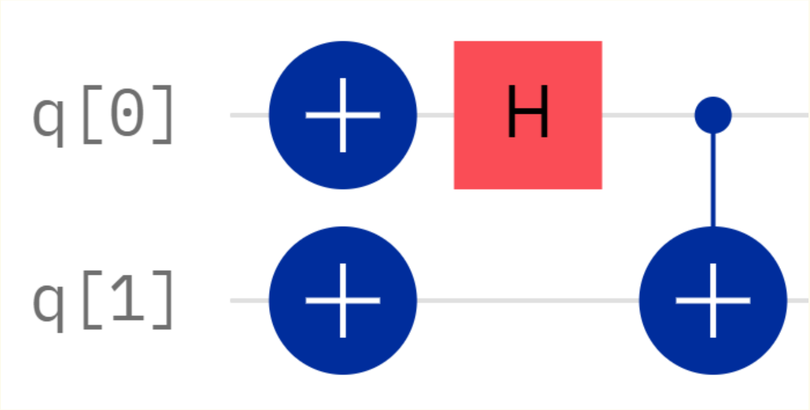
Create un circuito per ogni *stato di Bell*.

Aggiungete anche i bit classici e le operazioni di misura.

Eseguite ogni circuito su un simulatore online per 1024 shots.



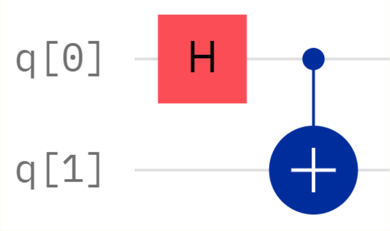




Esercizio!

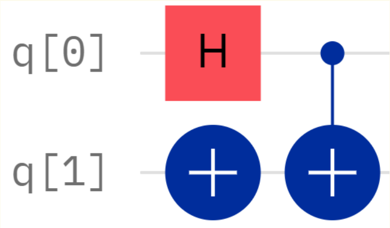
```
qc = QuantumCircuit(2,2)

qc.h(0)
qc.cx(0, 1)
qc.measure([0,1], [0,1])

service = QiskitRuntimeService(channel="ibm_quantum")

backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc, shots=1024)
result = job.result()
result
```

```
SamplerResult(quasi_dists=[{3: 0.515625, 0: 0.484375}], metadata=[{'shots': 1024, 'circuit_metadata': {}}])
```



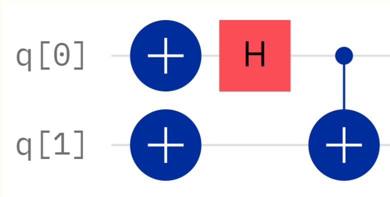

```
qc = QuantumCircuit(2,2)

qc.h(0)
qc.x(1)
qc.cx(0, 1)
qc.measure([0,1], [0,1])

service = QiskitRuntimeService(channel="ibm_quantum")

backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc, shots=1024)
result = job.result()
result
```

```
SamplerResult(quasi_dists=[{2: 0.5029296875, 1: 0.4970703125}], metadata=[{'shots': 1024, 'circuit_metadata': {}}])
```




```
qc = QuantumCircuit(2,2)

qc.x(1)
qc.h(0)
qc.x(1)
qc.cx(0, 1)
qc.measure([0,1], [0,1])

service = QiskitRuntimeService(channel="ibm_quantum")

backend = Sampler("ibmq_qasm_simulator")

job = backend.run(qc, shots=1024)
result = job.result()
result
```

```
SamplerResult(quasi_dists=[{3: 0.5029296875, 0: 0.4970703125}], metadata=[{'shots': 1024, 'circuit_metadata': {}}])
```



Esercizio!

Create un circuito con 2 qubit che prende in input un numero in formato binario e aggiunge 1:

00 -> 01

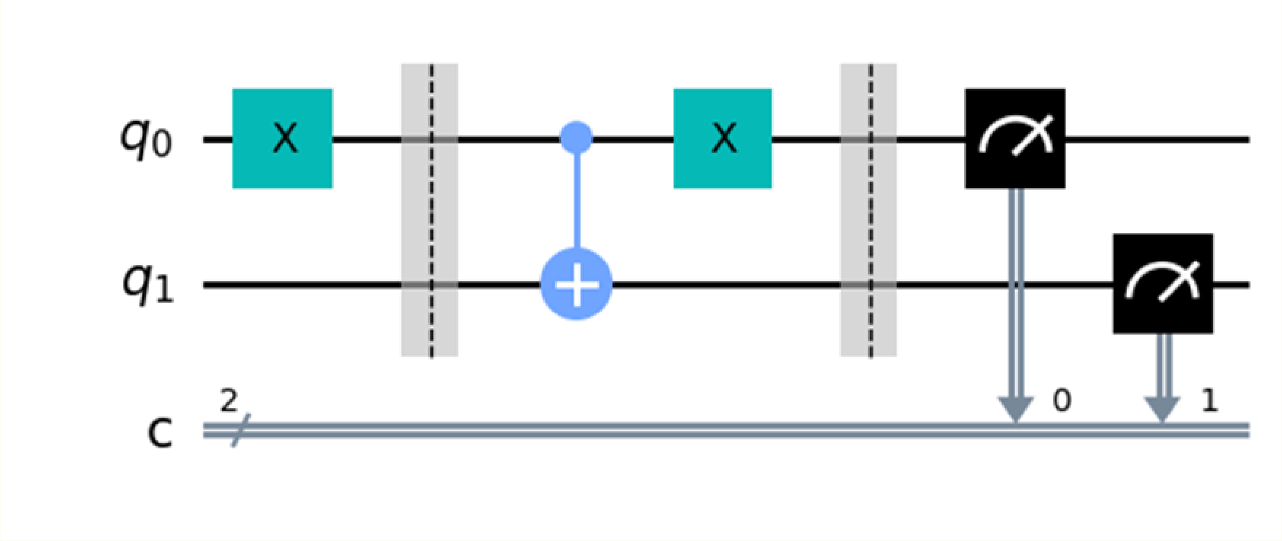
01 -> 10

10 -> 11

Cosa accade se l'input è 11?

Esercizio!

```python
from qiskit import QuantumCircuit

qc = QuantumCircuit(2, 2)

qc.x(0)
qc.barrier(0, 1)
qc.cx(0, 1)
qc.x(0)
qc.barrier(0, 1)
qc.measure(0, 0)
qc.measure(1, 1)
```



Esercizio!

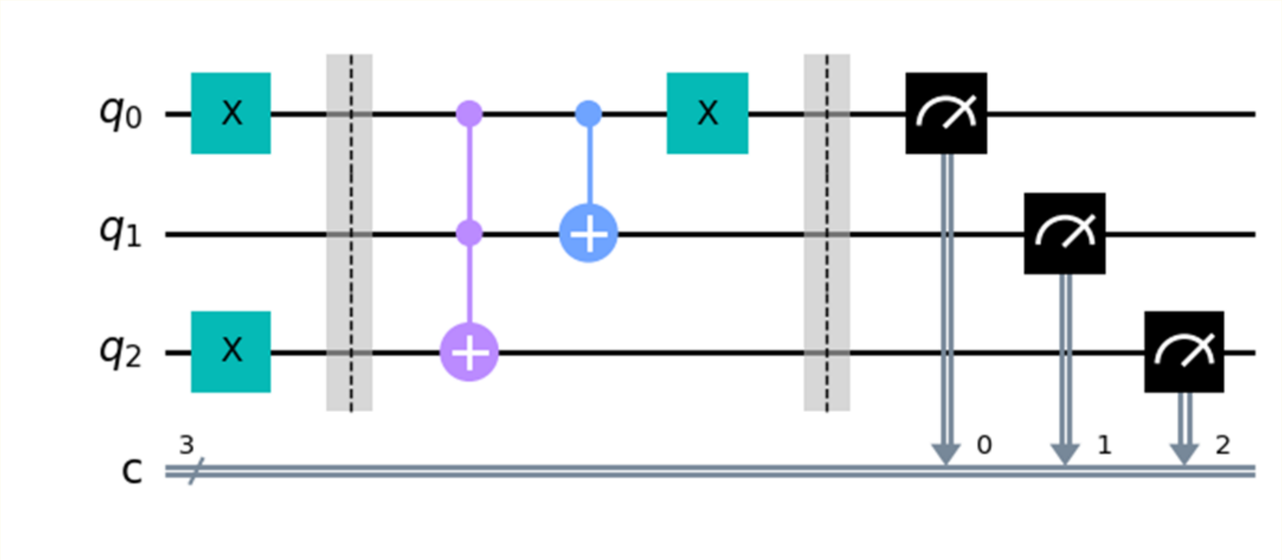
Come cambia il circuito precedente se invece di avere 2 qubit ne abbiamo 3?

Testate il circuito sul simulatore locale, un simulatore sul cloud e un quantum computer.

Che differenze notate?

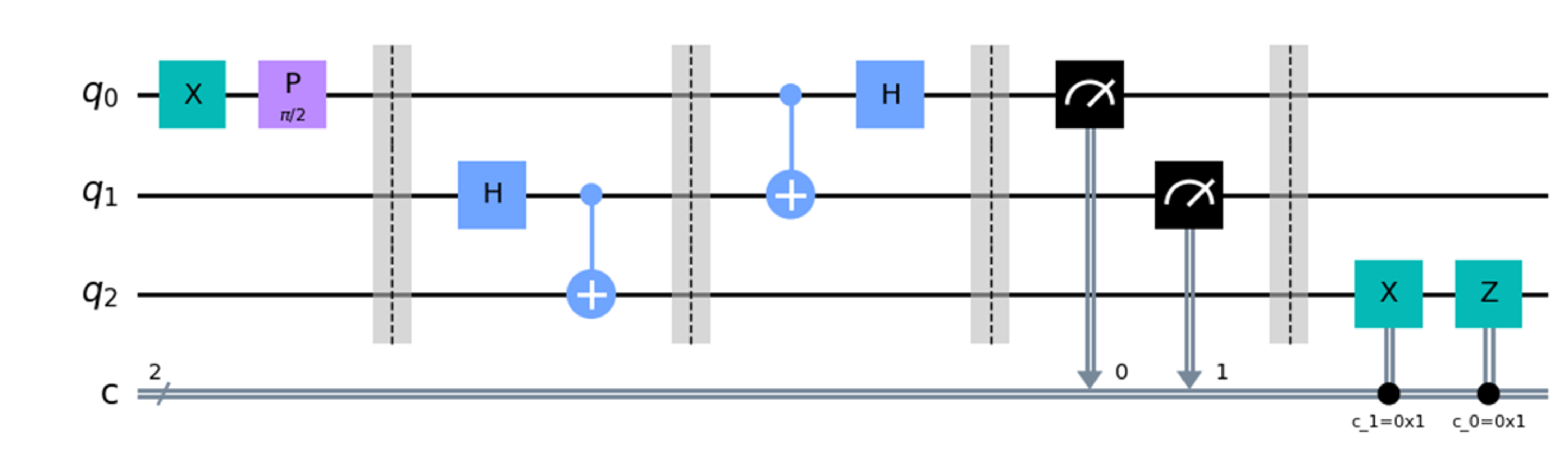
Esercizio!

```
qc = QuantumCircuit(3, 3)

qc.x(2)
qc.x(0)
qc.barrier(0, 1, 2)
qc.ccx(0, 1, 2)
qc.cx(0, 1)
qc.x(0)
qc.barrier(0, 1, 2)
qc.measure(0, 0)
qc.measure(1, 1)
qc.measure(2, 2)
```



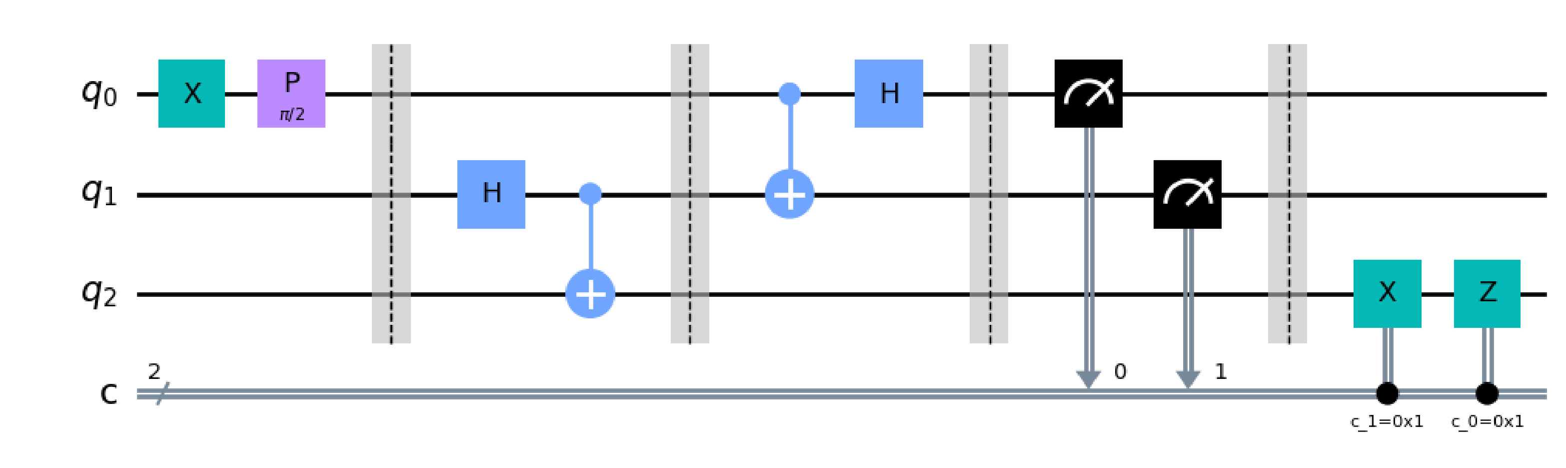
Esercizio!

Realizzate in Qiskit il circuito per il *teletrasporto quantistico*!



Esercizio!

```python
from qiskit import QuantumCircuit
from numpy import pi

qc = QuantumCircuit(3,2)

qc.x(0)
qc.p(pi / 2, 0)
qc.barrier(0,1,2)
qc.h(1)
qc.cx(1, 2)
qc.barrier(0,1,2)
qc.cx(0, 1)
qc.h(0)
qc.barrier(0,1,2)
qc.measure(0, 0)
qc.measure(1, 1)
qc.barrier(0,1,2)
qc.x(2).c_if(1, 1)
qc.z(2).c_if(0, 1)
```



Wrapping Up…

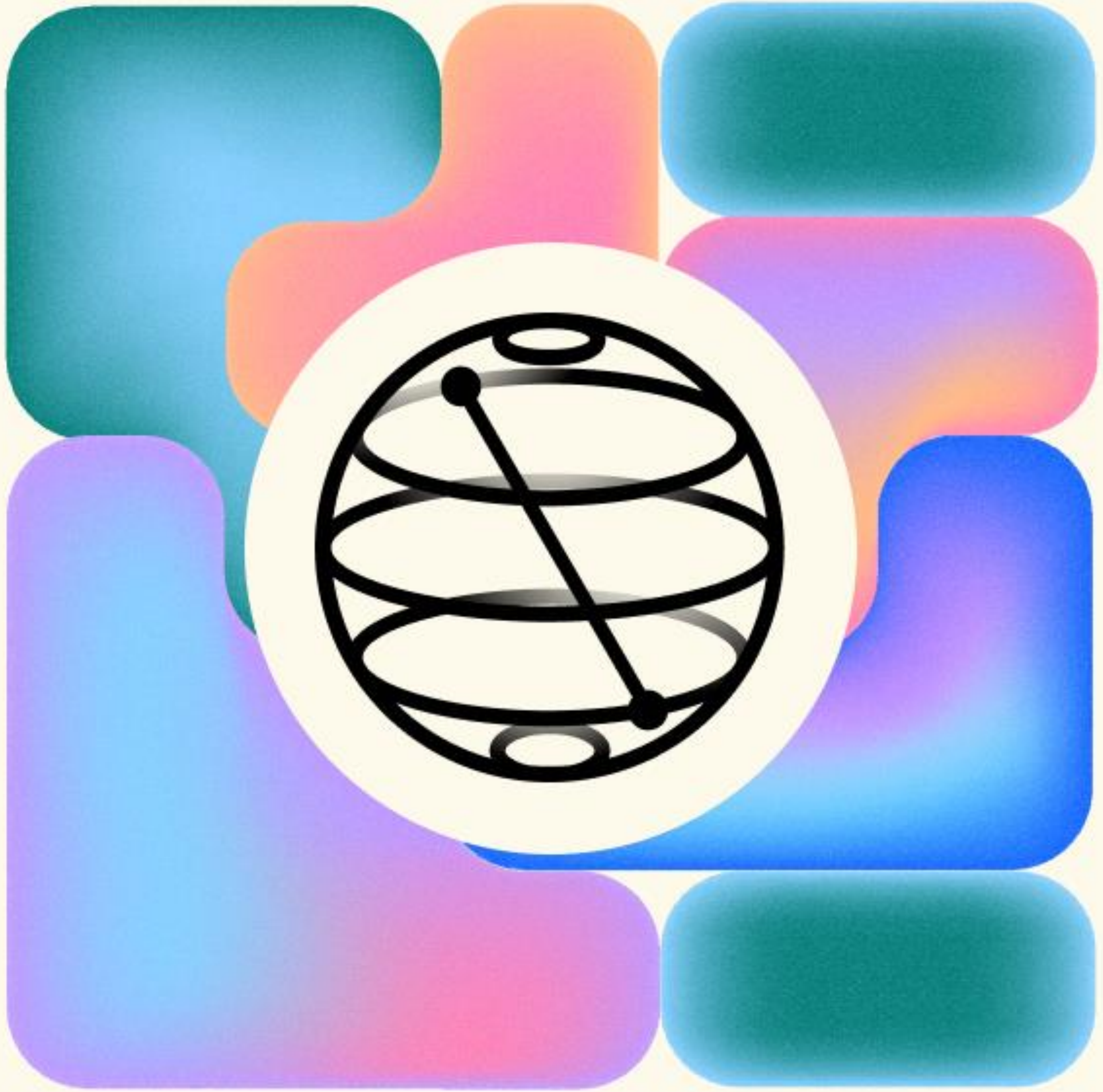
Provate ad eseguire i circuiti precedenti su diversi computer quantistici e simulatori

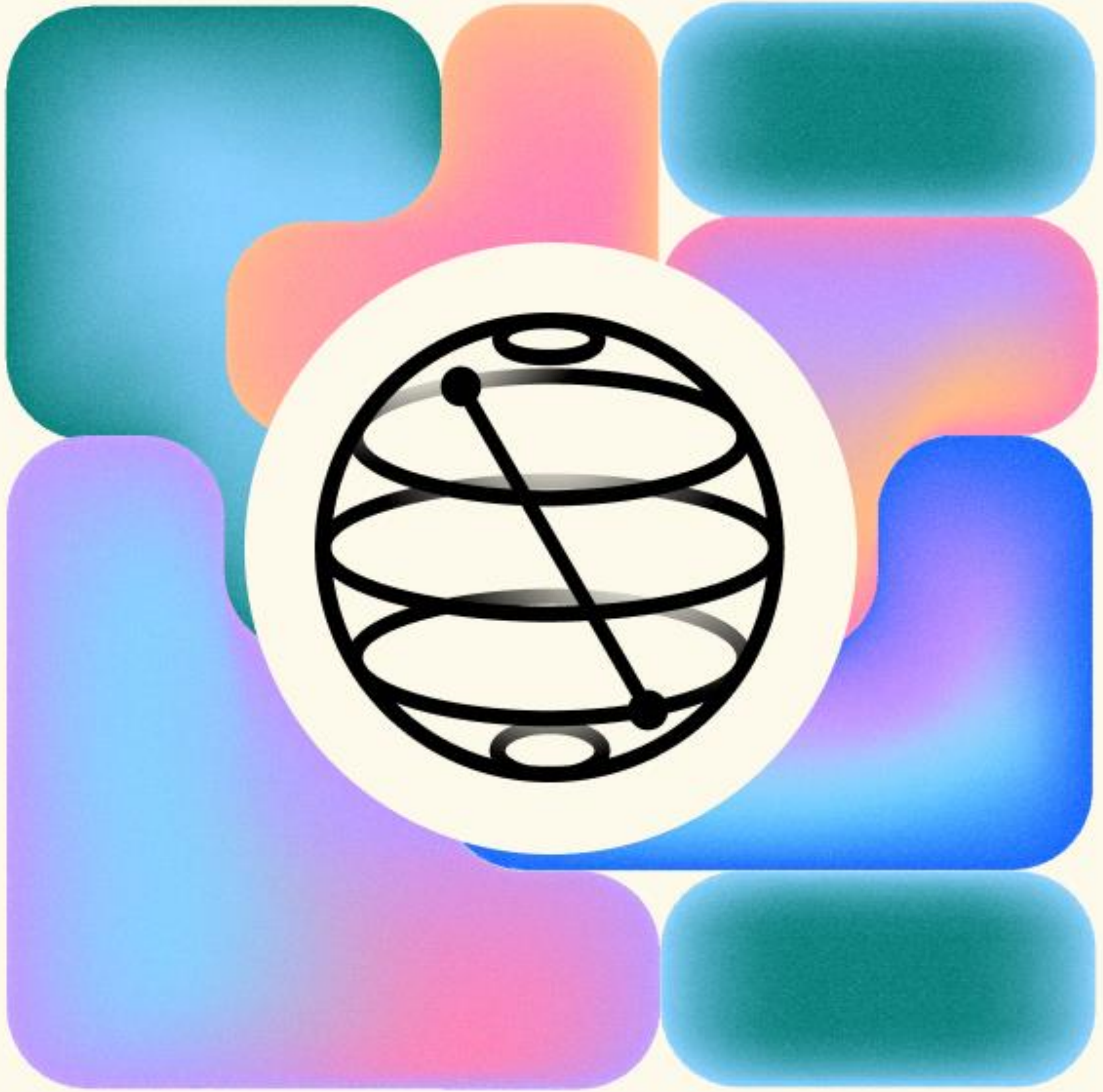
Ci sono differenze nei tempi di esecuzione e di attesa? Perché?

Come cambia il rumore e l’affidabilità dei risultati?

Cosa abbiamo imparato

- Come si programma un circuito quantistico con Python e Qiskit
- Come simulare in locale l'esecuzione di un circuito quantistico
- Come eseguire un circuito quantistico nel cloud di IBM Quantum su un vero computer quantistico
- Come interpretare i risultati di un'esecuzione di un circuito quantistico





Take-Home Message

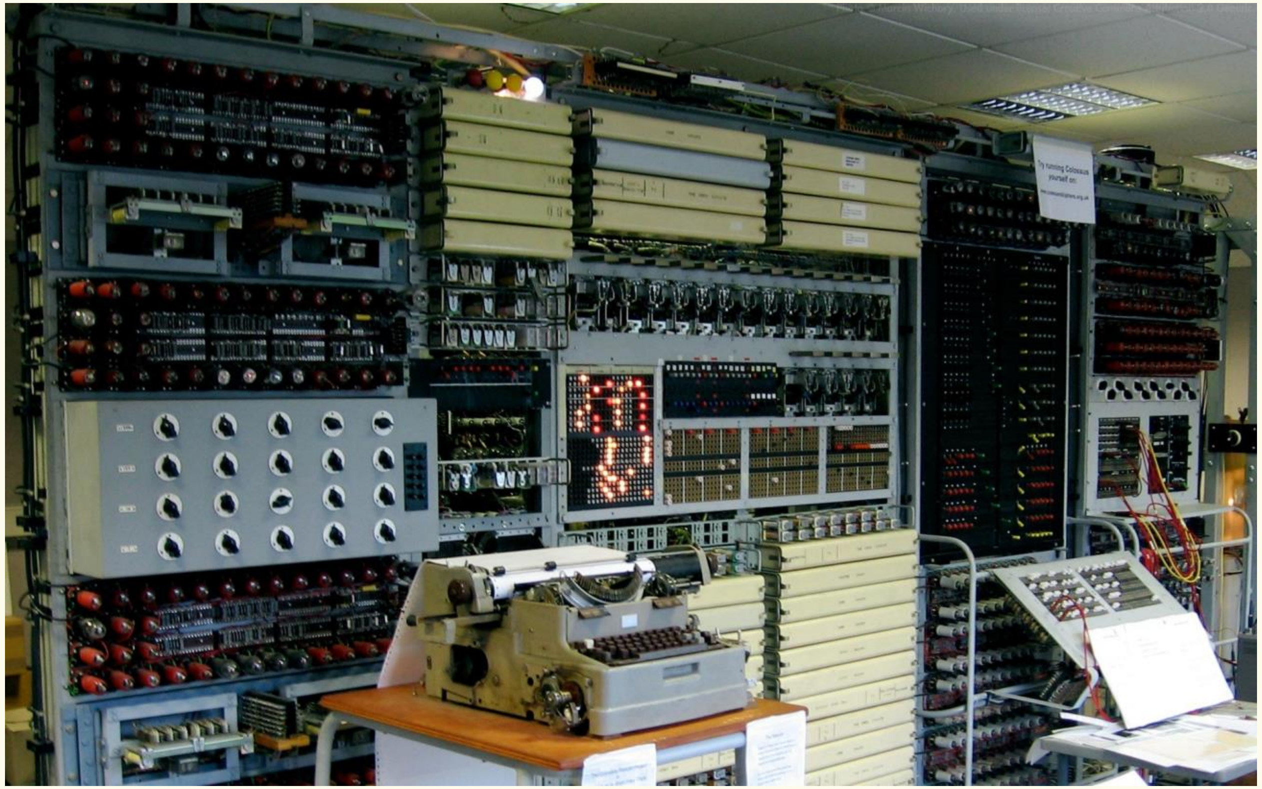
Sviluppare circuiti quantistici è un'arte che necessita fondamenta teoriche e molta pratica!

Strumenti e software come Qiskit possono aiutare e guidare gli sviluppatori nella programmazione di algoritmi quantistici.

È necessaria ancora molta ricerca affinché vengano sviluppate le giuste tecniche e metodologie di *Quantum Software Engineering*.

L'Ingegneria del Software ci consentirà di sviluppare e adoperare software quantistico in maniera più intuitiva ed efficiente.

In passato serviva una laurea per poter operare un computer, in futuro sarà necessario avere una laurea in fisica quantistica per programmare un computer quantistico?



Thank you

Hands On: I Primi Circuiti con Qiskit e Python

Giuseppe Bisicchia

Dottorando in Quantum Software Engineering

Università di Pisa

E-mail: giuseppe.bisicchia@phd.unipi.it

Website: gbisi.github.io

Qiskit | Fall Fest 2023