

# KOMPUTERY KWANTOWE

różnice względem klasycznych komputerów  
na przykładzie



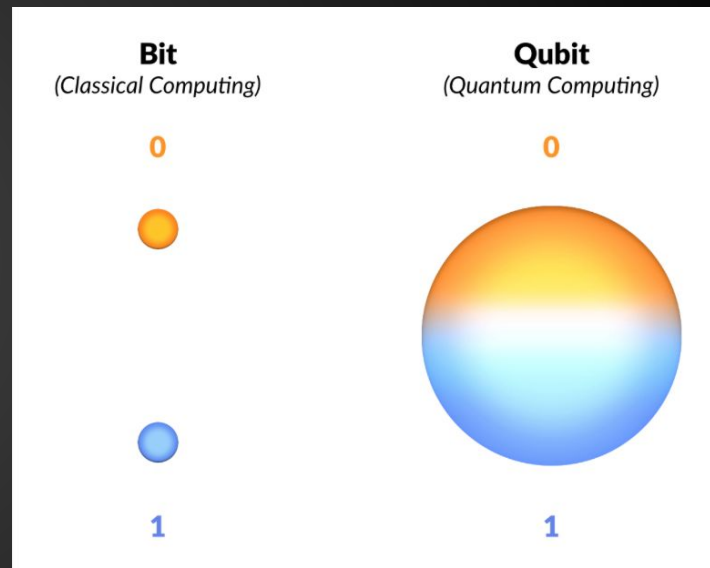


# 01. Podstawy działania

qubity, superpozycja

# + PODSTAWOWA JEDNOSTKA INFORMACJI - QUBIT

Qubit jest kwantową jednostką informacji. W odróżnieniu od klasycznego bitu oprócz wartości 0 i 1 może przyjmować pełne spektrum stanów pośrednich. Dzięki temu jest zdolny do przechowywania większej liczby informacji niż bit.



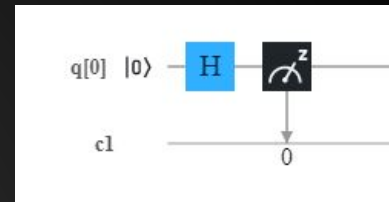
# SUPERPOZYCJA

Qubit jest kwantową **superpozycją** 0 i 1. Oznacza to, że jest jednocześnie 0 i 1 do momentu obserwacji, w którym przyjmuje wartość w zależności od prawdopodobieństwa bycia 0 lub 1.

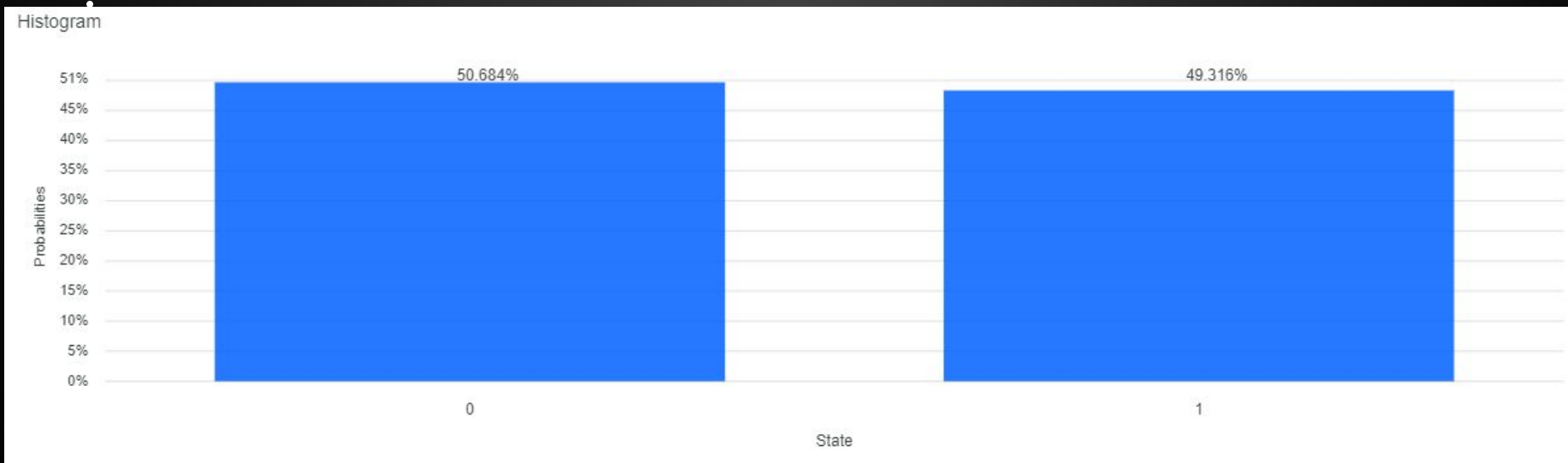
- Wprowadza to do kwantowych obliczeń **niepewność** i sprawia, że obliczenia należy powtarzać wiele razy, aby otrzymać prawidłowy wynik.

# PRZYKŁAD

Qubit w stanie superpozycji “zmierzony” 1024 razy okazał się być w ok. 50% przypadków 0 i ok. 50% przypadków 1.



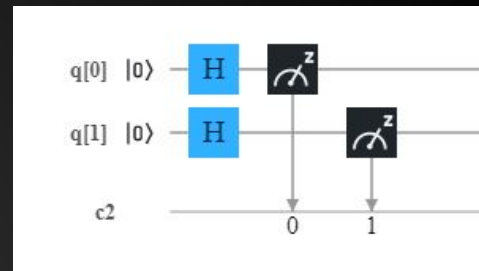
>>>>



+

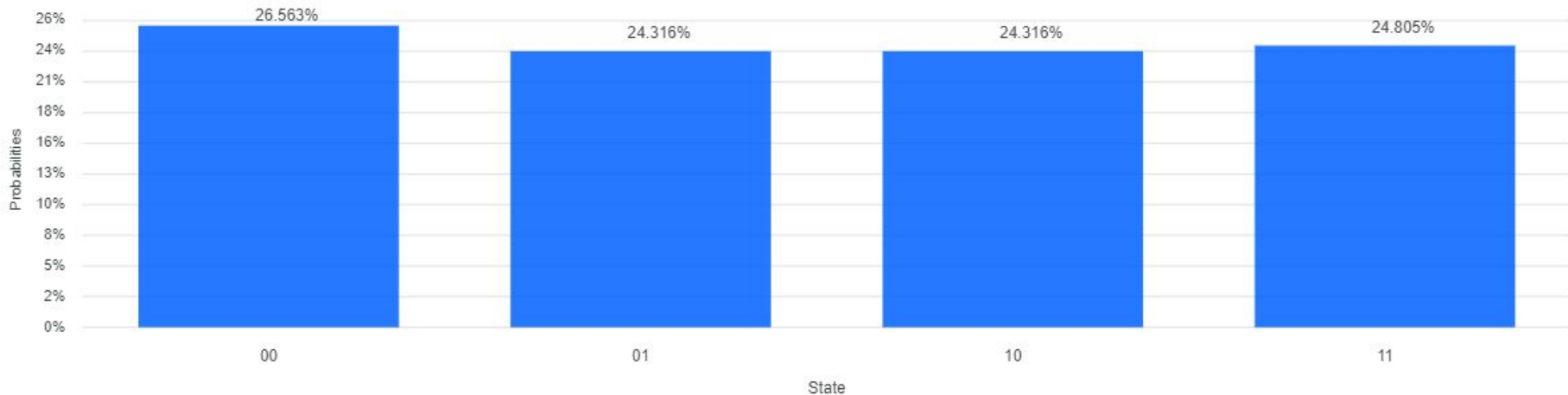
# PRZYKŁAD DLA 2 QUBITÓW

2 qubity zmierzone 1024 razy okazały się być w ok. 25% przypadków 00, 01, 10 i 11.



&gt;&gt;&gt;&gt;

Histogram





# 02. Bramki logiczne

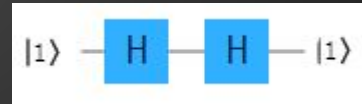
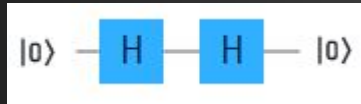
wybrane bramki potrzebne do  
zrozumienia dalszego przykładu

# Bramka Hamaarda (H)



Wprowadza qubit w superpozycję. Zastosowanie jej ponownie wprowadza qubit w stan w jakim był przedtem.

>>>>





+

+

## Bramka Haramarda (H)

W tej bramce występuje "phase effect". Zastosowana na qubit w stanie 0 sprawia, że w superpozycji jest on z 50% prawdopodobieństwem 0 i 1.

Natomiast zastosowana na qubit w stanie 1 sprawia, że w superpozycji jest on w 50% 0 i tak jakby w -50% w stanie 1 (oczywiście prawdopodobieństwo nie może być ujemne).

Z perspektywy obserwatora oba qubity w superpozycji mają po 50% szans na bycie 0 lub 1. **Nie są to jednak te same qubity.**

$$|0\rangle \xrightarrow{H} |0\rangle + |1\rangle$$

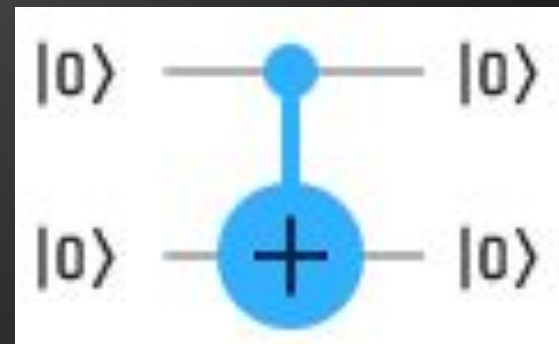
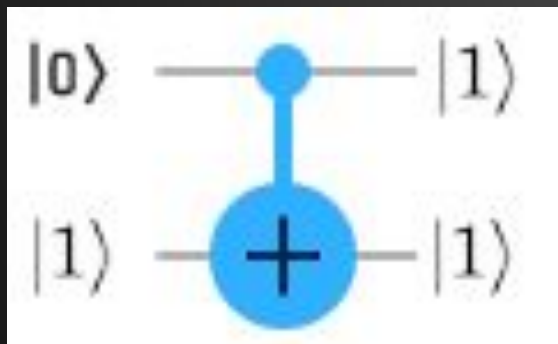
$$|1\rangle \xrightarrow{H} |0\rangle - |1\rangle$$

# Bramka Controlled NOT (cX)



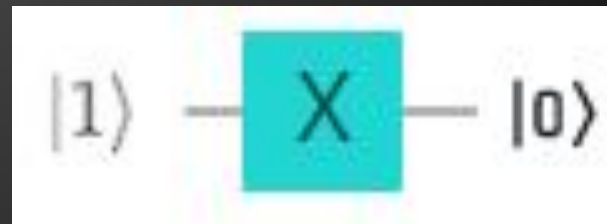
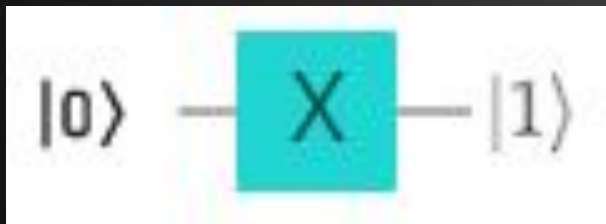
Operuje na parach qubitów, jeśli qubit kontrolny jest w stanie 1, to na drugim qubicie z pary wykonuje operację NOT.

>>>>



# Bramka Pauli-X (X)

Jest odpowiednikiem klasycznej bramki NOT.





# 03. Algorytm Bernsteina- Vaziraniego

opis problemu i implementacja



# Opis problemu

+

>>>>

+

Wyobraźmy sobie, że mamy czarne pudełko, które skrywa sekretną liczbę (ciąg  $n$  bitowy). Możemy ją odgadnąć wysyłając do pudełka własny ciąg  $n$  bitów. Pudełko wykona na sekretnej liczbie i naszym ciągu iloczyn skalarny i zwróci zawsze 0 lub 1.

>>>>

$S_{n-1} \dots S_1 S_0$   
Sekretny ciąg  
 $n$ -bitowy

← zapytanie  $X_{n-1} \dots X_1 X_0$

$X_{n-1} * S_{n-1} + \dots + X_0 * S_0$   
→ odpowiedź

Klasyczny komputer będzie potrzebował  $n$  prób, aby zgadnąć sekretną liczbę.

>>>>

**1101**  
Sekretny ciąg  
n-bitowy

0001

0010

0100

1000

zapytanie

odpowiedź

$$1*0+1*0+0*0+1*1=1$$

$$1*0+1*0+0*1+1*0=0$$

$$1*0+1*1+0*0+1*0=1$$

$$1*1+1*0+0*0+1*0=1$$

} n prób

+

>>>>

+



Gdyby nasz ciąg miał długość 1000 bitów, klasyczny komputer potrzebowałby 1000 prób aby go odgadnąć. Komputer potrzebuje tyle prób, ile cyfr ma liczba.




Komputer kwantowy natomiast może zgadnąć sekretną liczbę zawsze za jedną próbą używając algorytmu Bernsteina-Vaziraniego.

Pokazuje to, że komputery kwantowe w specyficznych sytuacjach mają znaczącą przewagę nad klasycznymi.







# Implementacja



Dostęp do wykonywania programów na komputerze kwantowym oferuje platforma **IBM Q Experience**.




Możemy tutaj stworzyć obwód kwantowy i wykonać go na symulatorze, lub prawdziwym komputerze kwantowym IBM.



Na IBM Q Experience używa się języka **OpenQASM**.



Programy kwantowe można także pisać w **Pythonie** używając frameworka **Qiskit**.



Zaczynamy od zadeklarowania **qubitów** i klasycznych **bitów** potrzebnych do realizacji algorytmu.

Bity potrzebne nam będą do sczytania wyniku.

W tym przykładzie sekretną liczbą będzie **1101**.

Potrzebujemy więc 5 qubitów (4 jako reprezentację każdej cyfry z liczby oraz qubit kontrolny dla bramki cX) i 4 bitów do zapisania wyniku.

Instrukcja **qreg q[5];** deklaruje qubity.

Instrukcja **creg c[4];** deklaruje bity.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[4];
```

+

Qubity domyślnie inicjowane są z wartością 0.  
Potrzebujemy aby qubit kontrolny (z indeksem 4) miał wartość 1, więc negujemy go używając bramki X.

Następnie wszystkie qubity wprowadzamy w superpozycję. Stosujemy do tego bramkę H.

Instrukcja `h q[indeks qubitu];` wprowadza qubit w superpozycję.

x q[4];  
h q[0];  
h q[1];  
h q[2];  
h q[3];  
h q[4];

+

Teraz zaimplementujemy “czarne pudełko”. 1101 zakodujemy za pomocą bramek cX na odpowiednich qubitach. Do bramek cX potrzebny będzie qubit kontrolny q[4].

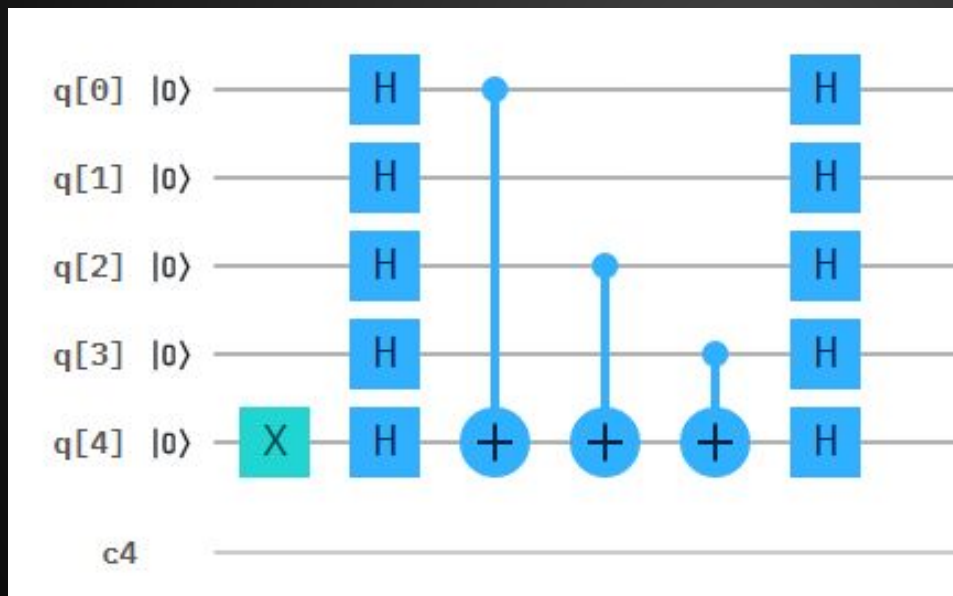
Instrukcja **cx q[ indeks ], q[ indeks ]**; tworzy bramkę cX. Pierwszy qubit w instrukcji jest docelowym, drugi jest qubitem kontrolnym.

```
cx q[0],q[4];  
cx q[2],q[4];  
cx q[3],q[4];
```

+

W kolejnym kroku znów stosujemy bramkę H na wszystkich qubitach.

Dotychczas stworzony obwód kwantowy wygląda następująco:



```
h q[0];
```

```
h q[1];
```

```
h q[2];
```

```
h q[3];
```

```
h q[4];
```

P

Pozostało zaimplementować sczytywanie wyniku.

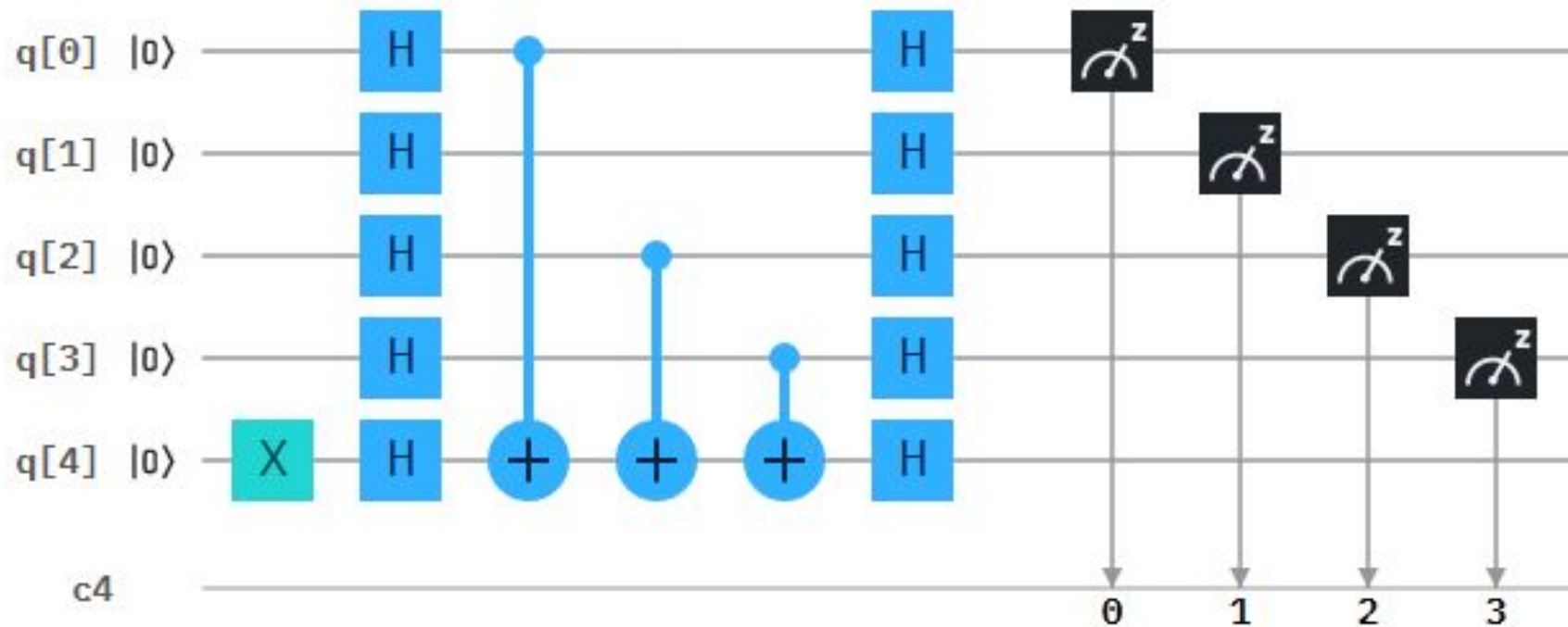
- W tym celu posłużymy się instrukcją

**measure** **q[x]** -> **c[x]**;

gdzie **q[x]** reprezentuje qubit, a **c[x]** klasyczny bit.

```
measure q[0]->c[0];  
measure q[1]->c[1];  
measure q[2]->c[2];  
measure q[3]->c[3];
```

Gotowy obwód kwantowy ma postać:





Pozostało uruchomić program. Najpierw wykonamy go na symulatorze. Kwantowe obliczenia obarczone są niepewnością, więc powtórzymy je 1024 razy, aby upewnić się, że otrzymamy prawidłowy wynik.

Run your circuit

1. Select an available backend

Backends availability and functionality can vary depending on the provider.

ibmq\_qasm\_simulator in ibm-q/open/main ▼

2. Select number of shots

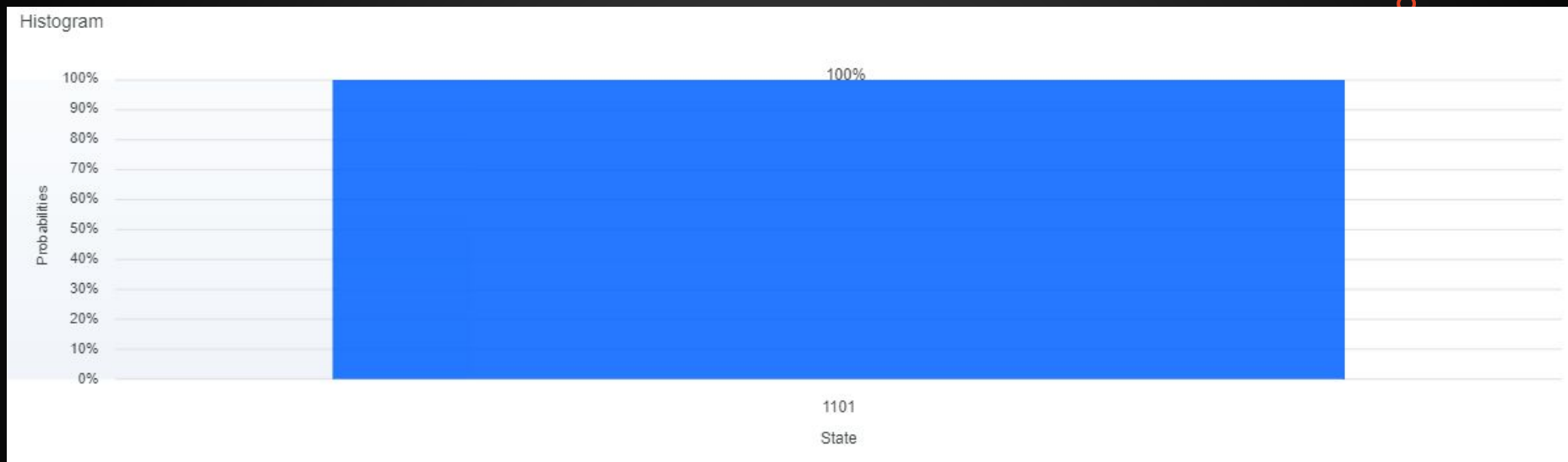
Increase the number of shots to improve statistical accuracy.

1024 ▼

Cancel

Run →

Otrzymałiśmy następujące wyniki:



Wszystkie 1024 próby zwróciły wynik 1101, który jest poprawny. Ponieważ to symulator, do rezultatów nie wkradły się błędne wyniki.

Teraz uruchomimy nasz program na prawdziwym komputerze  
kwantowym.

Run your circuit

1. Select an available backend

Backends availability and functionality can vary depending on the provider.

ibmq\_london in ibm-q/open/main

2. Select number of shots

Increase the number of shots to improve statistical accuracy.

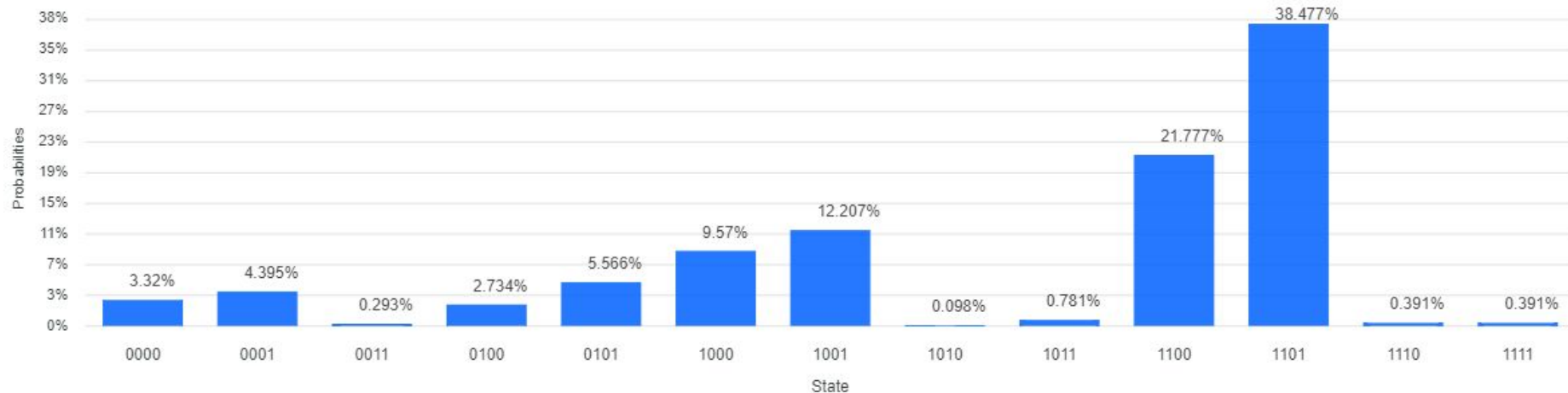
1024

Cancel

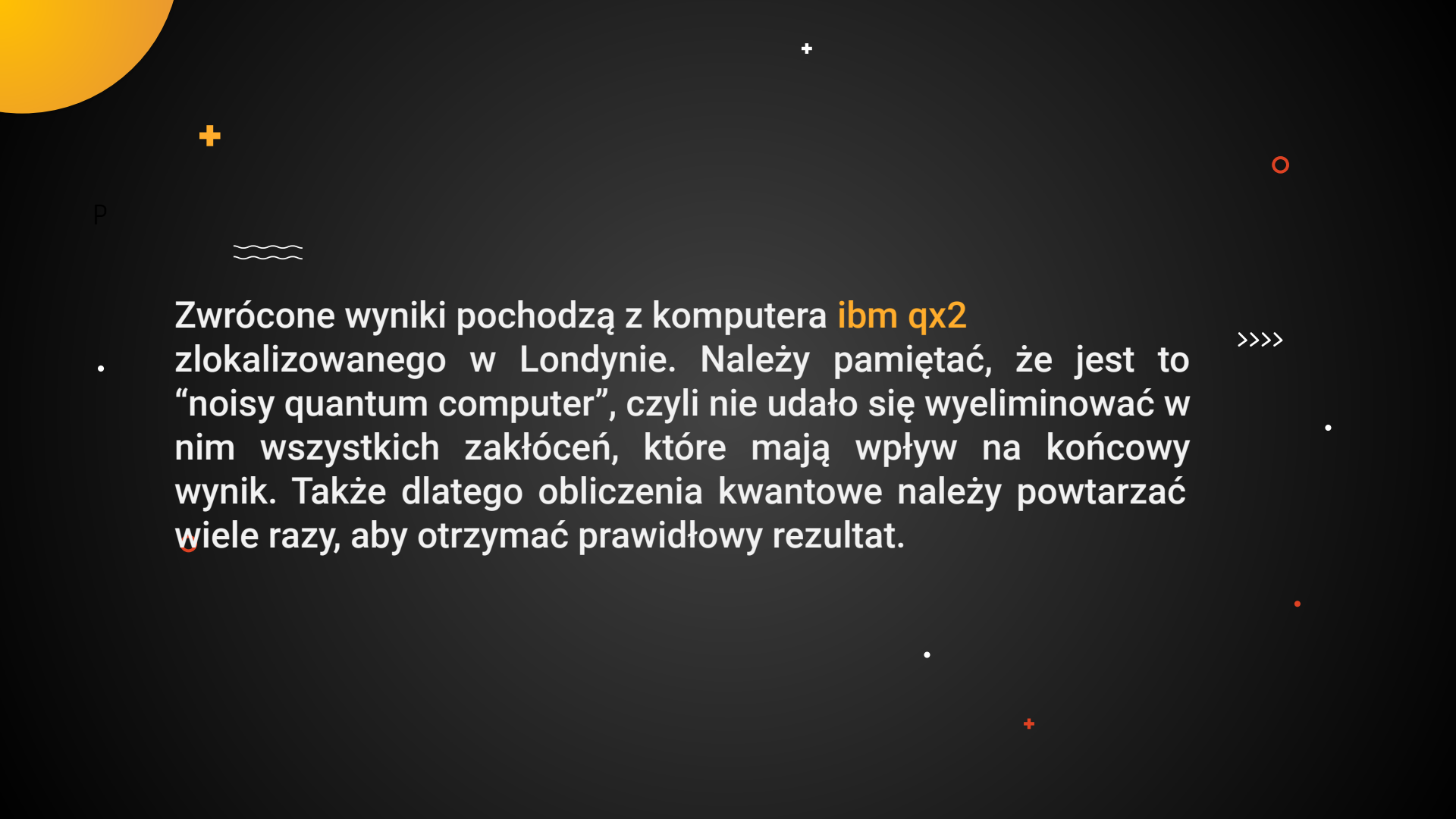
Run

Otrzymaliśmy następujące wyniki:

Histogram



Tym razem otrzymaliśmy całe spektrum wyników, z których najbardziej prawdopodobnym jest 1101. Przyjmujemy go zatem za prawidłowy wynik.



P

~~~~~

Zwrócone wyniki pochodzą z komputera **ibm qx2** zlokalizowanego w Londynie. Należy pamiętać, że jest to “noisy quantum computer”, czyli nie udało się wyeliminować w nim wszystkich zakłóceń, które mają wpływ na końcowy wynik. Także dlatego obliczenia kwantowe należy powtarzać wiele razy, aby otrzymać prawidłowy rezultat.

>>>>

+



IBM Q System



# The Sounds of IBM: IBM Q



The background is dark gray with various abstract elements: a large yellow and orange curved shape on the right, a red trapezoid on the left, and several small white and red symbols (plus signs, dots, wavy lines, and chevrons) scattered across the space.

# Implementacja z użyciem Qiskit





**Qiskit** jest frameworkiem do pisania programów kwantowych. Bardziej skomplikowane projekty byłyby trudne do implementacji w **OpenQASM**, który jest językiem niskiego poziomu. Takie projekty pisane są w **Pythonie** używając frameworka **Qiskit**.

W dalszej części prezentacji pokazana zostanie implementacja algorytmu Bernsteina-Vaziraniego w Qiskit. Poszczególne instrukcje będą wyjaśnione w formie komentarzy w kodzie programu.



[Kod w Pythonie na github](#)

In [1]: `from qiskit import *`

In [2]: `sekret = '1101' # sekretna liczba  
n = len(sekret) # ile cyfr ma sekretna liczba  
circuit = QuantumCircuit(n+1,n) # nowy obwód kwantowy z n+1 qubitami (dodatkowy qubit dla  
bramek controlled NOT - cX) oraz n klasycznymi bitami`

In [ ]:

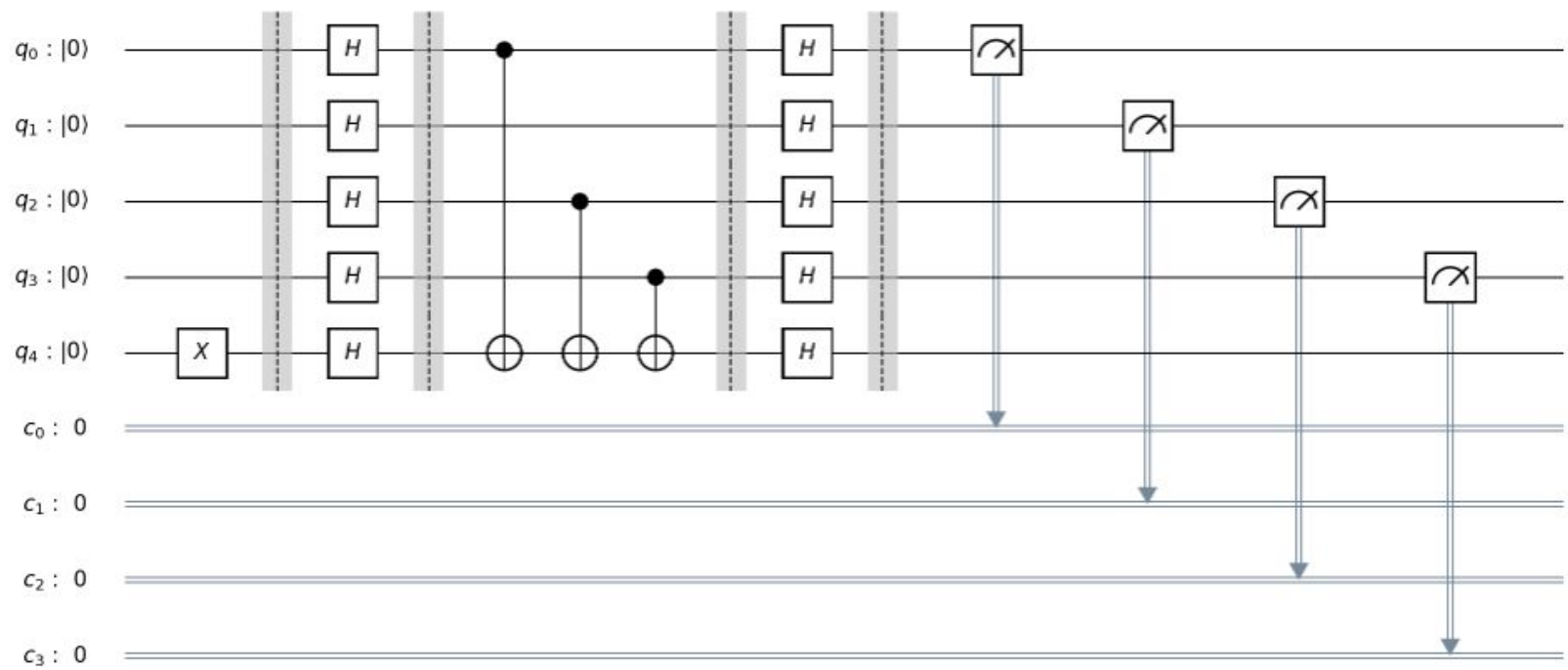
In [3]: `circuit.x(n) # zastosowanie bramki X (odpowiednik NOT) na ostatnim qubicie (qubity  
indeksowane są 0...n, ostatni z nich ma więc indeks n)  
circuit.barrier()  
circuit.h(range(n+1)) # zastosowanie bramki Hadamarda (H) (wprowadzającej qubit w stan  
superpozycji) na wszystkich qubitach  
circuit.barrier()`

```
In [4]: for i, one in enumerate(reversed(sekret)):
        if one == '1':
            circuit.cx(i, n) # zastosowanie bramki cX (cX operuje na parach qubitów, jeśli
                             qubit kontrolny jest w stanie 1, to na drugim qubicie z pary wykonuje negację)
        circuit.barrier()
```

```
In [5]: circuit.h(range(n+1)) # zastosowanie bramki H na wszystkich qubitach
        circuit.barrier()
        circuit.measure(range(n), range(n)) # sprawdzenie stanu qubitów i zapisanie go do
        klasycznych bitów
```

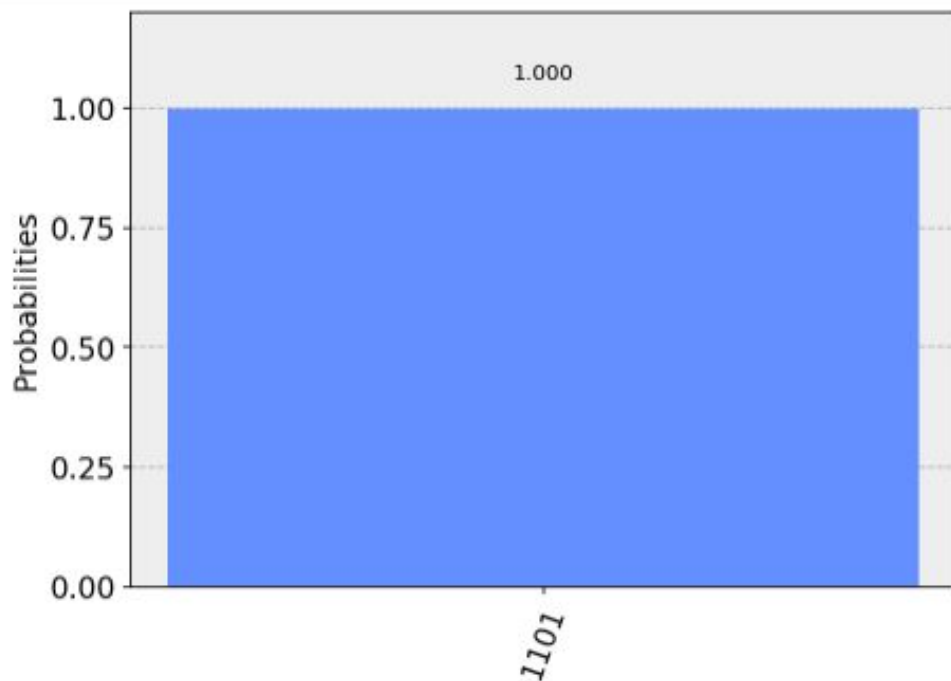
+

```
In [6]: %matplotlib inline
circuit.draw(output='mpl') # schemat utworzonego obwodu kwantowego
```



Out[6]:

```
In [7]: simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator, shots=1024).result() # wykonanie programu na symulatorze
from qiskit.visualization import plot_histogram
plot_histogram(result.get_counts(circuit)) # wyświetlenie wyników
```



Out[7]:



# Dziękujemy za uwagę

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.

Please keep this slide for attribution.

# Źródła:



- Using QISkit: The SDK for Quantum Computing >>>>

<https://www.youtube.com/watch?v=LSA3pYZtRGg>

- Bernstein-Vazirani Algorithm — Programming on Quantum Computers Ep 6 <https://www.youtube.com/watch?v=sqJlpHYI7oo>

- Github Qiskit <https://github.com/Qiskit/>