Qiskit'e Giriş ve Kuantum Algoritmalar

Prof. Dr. İhsan Yılmaz, Osman Ceylan

Contents

1 Giriş

1.1 Qiskit Nedir?

Qiskit, IBM tarafından geliştirilen, açık kaynak kodlu, bulut üzerinden gerçek kuantum bilgisayarlarına erişim izni tanıyan bir kuantum devre simülatörüdür. Qiskit bir devre simülatörünün dışında bünyesinde amacına yönelik farklı unsurları da barındıran çevresel bir yazılımdır.

1.2 Qiskit Ne Yapar?

Qiskit, kuantum sistemleri ve simülatörlerle etkileşim için gereken eksiksiz araç setini sağlayarak kuantum uygulamalarının geliştirilmesini hızlandırır.

1.3 Kurulum

Qiskit Python bulunduran tüm masaüstü işletim sistemlerinde çalışacak şekilde geliştirilmiştir. Linux, MacOS veya Windows üzerinde Python 3.6 veya üzeri bir sürüm ile kullanılabilir.

```
pip install -U qiskit
pip install -U qiskit-aer
```

Eğer jupyter notebook ortamında kurmak istiyorsanız komutların başına % işareti eklenmelidir.

```
from qiskit import __version__
print('Qiskit Sürümü: ', __version__)
```

```
Qiskit Sürümü:  1.0.2
```

1.3.1 Sorun Giderme

Qiskit kurulumu gereksinimlerin karşılanamadığı durumlarda tamamlanamaz. Böyle bir durumda sanal bir Python ortamı üzerinden devam edilmedir.

2 İş akışı

Bir kullanıcı herhangi bir yazılımı kullanmadan önce yazılımın nasıl çalıştığını bilmesi gerekmektedir. İlk öğrenmenin en başarılı yolu ise genel kullanım durumunun bir örnek üzerinden anlatılmasıdır.

2.1 İş akışının unsurları

Qiskit'i kullanırken, bir kullanıcı için genel kullanım durumu olarak aşağıdaki dört üst düzey adımdan oluşur:

- **Tasarım**: Bu aşamada kullanıcı kendi problemini temsil ettiği devreyi tasarlamalıdırlar.
- **Derleme**: Bu aşamada kullanıcı belirli bir kuantum hizmeti için devrelerini derlemelidir.
- **Çalıştırma**: Bu aşamada kullanıcı aktif değildir, yerelde veya bulut üzerinden devre simüle edilir.
- **Analiz**: Bu aşamada simülasyondan gelen sonuçlar kullanıcı tarafından analiz edilerek bir sonuç elde eder.

2.2 Adım Adım İlk Kullanım

Bu bölümde bir devre oluşturarak simülasyon yaptığımız bir örneği adım adım anlatacağız.

2.2.1 Paketlerin İçe Aktarımı

Bir uygulama geliştirme sürecinde her zaman ilk adım gereksinimlerin içe aktarımı olmuştur. Bu nedenle ilk olarak Qiskit'i içe aktarmak ile başlayacağız.

```
1 import numpy as np
2 from qiskit import QuantumCircuit, QuantumRegister,
   ClassicalRegister
3 from qiskit import transpile, assemble
4 from qiskit_aer import AerSimulator
5 from qiskit import visualization
6 from qiskit import quantum_info
```

İçe aktarımları detaylandırmak gerekirse:

- *Numpy*: Numpy Python üzerinde popüler çok amaçlı bir matematik kütüphanesidir.
- *QuantumCircuit*: Qiskit içerisinden kuantum devreyi içe aktarır.
- *QuantumRegister*: Qiskit içerisinde bulunan kübitleri tanımlamakta kullanılan sınıf.

- *ClassicRegister*: Qiskit içerisinde bulunan klasik bitleri tanımlamakta kullanılan sınıf.
- *transpie, assemble*: Qiskit üzerinde tasarlanan devreleri belirli bir simülatöre veya cihaza uygun biçimde derlenmesini sağlar.
- *AerSimulator*: Yüksek performanslı kuantum devre simülatörü.
- *visualization*: Çeşitli görsel grafik ve çizelgeleri oluşturmak için kullanılan modülüdür.
- *quantum_info*: Durum vektörü gibi çeşitli kuantum bilgi gösterimleri modülüdür.

2.2.2 Devrenin Başlatılması

Bu aşamada kullanıcılar devrelerini ilklemelidirler. Kullanıcı programlamak istediği kuantum devrenin kuantum ve klasik bit gereksinimlerini bilmelidirler. Biz bu örnek için 2 adet kuantum ve 2 adet klasik bit ile devre başlatacağız.

```
1 circuit = QuantumCircuit(2, 2)
```

veya

```
1 qubit_1 = QuantumRegister(1, name='A')
2 qubit_2 = QuantumRegister(1, name='B')
3 bits = ClassicRegister(2)
4 circuit = QuantumCircuit(qubit_1, qubit_2, bits)
```

veya

```
1 qubits = QuantumRegister(2, name='Alice\' qubits')
2 bits = ClassicRegister(2),
3 circuit = QuantumCircuit(qubits, bits)
```

Note: Devreler oluşturulduğunda her zaman temel durumda başlar.

2.2.3 Devreyi Tasarlamak

Bu aşamada kullanıcılar devre üzerinde istedikleri devre elemanlarını istedikleri yere konumlandırmadırlar. Qiskitte kullanılabilen devre elemanları şunlardır:

- **Kapı**: Operatör olarak da adlandırılan bu devre elemanı devrenin en önemli yapı taşıdır. Uygulanmak istenen kapının girdi sayısı ile kapının uygulandığı kübit sayısı eşleşmelidir.
- **Ölçüm**: Ölçüm operatörü ile kullanıcılar devrelerinde kübitler üzerinde ölçüm yapabilirler.
- **Bariyer**: Bu devre elemanı devre akışı içerisinde kapılar arasında bariyer sağlayarak oluşabilecek bazı karmaşıklıklardan kaçınabilirler.
- **İkili Kontrollü Kapılı**: Bu kapılar adından da anlaşılacağı gibi klasik bit kontrol olacak şekilde kübitlere kapı uygular.

```
1 qubits = QuantumRegister(2, name='my_qubits')
2 bits = ClassicRegister(2)
3 circuit = QuantumCircuit(qubits, bits, name='Bell')
4 circuit.h(qubits[0])
5 circuit.cx(qubits[0], qubits[1])
6 circuit.measure(qubits, bits)
```

Üstteki devre ile bir Bell çifti oluşturulur ve her iki kübiti de ölçülür:

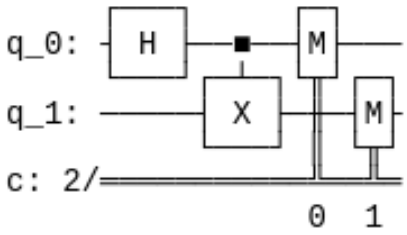
- *Hadamard: h()* ile belirlen kapı bir kübiti süperpozisyon durumuna getirir. Burada ilk kübite (0) uygulanmaktadır.
- *Kontrollü-NOT: cx()* ile belirtilen kapı iki kübitlik gerektirir. Burada ilk kübit (0) kaynak, ikinci kübit (1) ise hedeftir.
- *Ölçüm: measure()* ile ölçüm işlemi yapılabilir. İlk parametre ölçülecek kübitlerin listesini, ikinci parametre ise kaydedilmesinin istendiği klasik bitlerin listesini alır.

Note: Kübitlere aynı zamanda index sayıları ile de erişilebilir. **Örnek:** circuit.cx(0, 1)

2.2.4 Devrenin Çizimi

Tasarlanan devreler *draw()* yöntemi ile çizilebilirler.

```
1 circuit.draw()
```



Note: *output='mpl'* ile matplotlib kullanarak renkli çıktı alınabilir.

2.2.5 Devrenin Simülasyonu

Qiskit Aer, kuantum devreleri için yüksek performanslı bir simülatör çerçevesidir. Farklı simülasyon hedeflerine ulaşmak için birkaç simülatör sağlar. Biz bu örnekte için QasmSimulator kullandık. Devre 1000 kere simüle edilecek şekilde ayarladık.

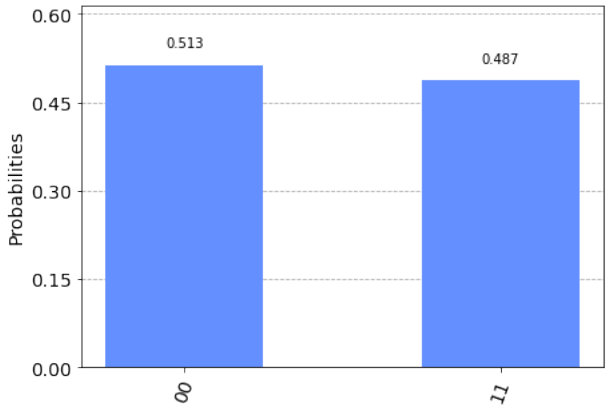
```
1 simulator = AerSimulator()
2 circuit = transpile(circuit, simulator)
3 result = simulator.run(circuit).result().get_counts(0)
4 print('Sonuçlar: ', result)
```

```
1 Sonuçlar: {'11': 487, '00': 513}
```

Rakamlardan görüleceği üzere Bell çifti oluşumun sonucunda 00 ve 11 durumları yaklaşık olarak eşittir ve diğer durumlara rastlanmamıştır. Bir sonraki bölümdeki yöntem ile bu çıktıları görselleştireceğiz.

2.2.6 Sonuçların Görselleştirilmesi

plot_histogram() metotu kullanılarak bir histogram oluşturulabilir.



3 Algoritmalar

Günümüze kadar gelen her ayrık matematik problemi klasik bilgisayarlar ile çözümü harcanan zaman ve enerji bakımından olanaklı görünmüyor. Kuantum bilgisayarlar ise klasiğe karşın olan üstünlüklerini kullanarak bu problemlere olanaklı çözümler getirebiliyor. Bu bölümün tamamında sadece kuantumda yapılabilen veya kuantum bilgisayarlar ile yapılmasının daha olanaklı olduğu algoritmaların Qiskit üzerinde kodlanmasını anlatacağız.

3.1 Işınlama Algoritması

Bu algoritma ile bir kuantum durumunu kübitler kullanılarak iki taraf arasında gönderimini yapacağız.

3.1.1 Kurulum

İlk olarak gerekli metotları içe aktaralım. Yerel bir simülasyon için aşağıdaki tüm içe aktarımlar gerekli değildir.

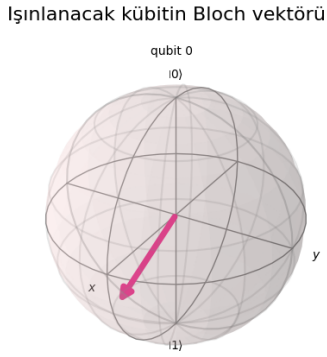
```
1 import numpy as np
2 from qiskit import QuantumCircuit, QuantumRegister,
  ClassicalRegister
3 from qiskit import transpile, assemble
4 from qiskit_aer import AerSimulator
5 from qiskit.visualization import plot_histogram
6 from qiskit import quantum_info
7 from qiskit.visualization import plot_bloch_multivector
```

Bu aşamada kuantum devremizi başlatalım. 3 adet kübite ve 3 adet farklı klasik bite ihtiyacımız var.

```
alice_teleport = QuantumRegister(1, name='alice_teleport')
alice_entangle = QuantumRegister(1, name='alice_entangle')
bob_entangle = QuantumRegister(1, name='bob_entangle')

alice_teleport_measure = ClassicalRegister(1, name="alice_tp_bit")
alice_entangle_measure = ClassicalRegister(1, name="alice_bell_bit")
bob_entangle_measure = ClassicalRegister(1, name="bob_bell_bit")

teleportation_circuit = QuantumCircuit(alice_teleport, alice_entangle, bob_entangle,
alice_teleport_measure, alice_entangle_measure,
bob_entangle_measure)
teleportation_circuit.draw(output='mpl')
```

3.1.2 Adım 0: Işınlama için rastgele bir kübit durumu seçilir

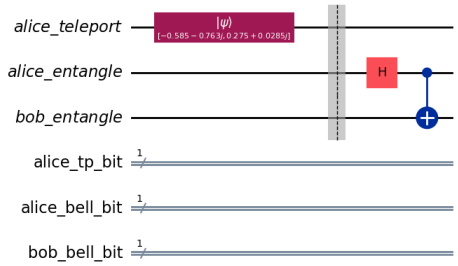
```
teleport_qubit_state = quantum_info.random_statevector(2)
print(teleport_qubit_state)
plot_bloch_multivector(teleport_qubit_state, title='Işınlanacak kübitin Bloch vektörü')
```



3.1.3 Adım 1: Bell durumu paylaşma

Bu adımda aşağıdaki kod ile devremiz üzerinde Bell durumunu oluşturacağız. Bu işlemi bir üçüncü tarafın yaptığını ve Alice ile Bob'a eşleri gönderdiğini düşünerek devam edeceğiz.

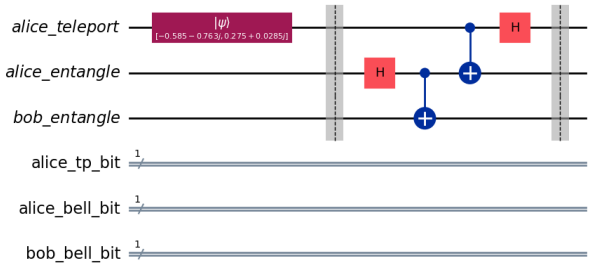
```
1 teleportation_circuit.initialize(teleport_qubit_state,
    [0])
2 teleportation_circuit.barrier()
3 teleportation_circuit.h(alice_entangle)
4 teleportation_circuit.cx(alice_entangle, bob_entangle)
5 teleportation_circuit.draw(output='mpl')
```



3.1.4 Adım 2: Alice qubiti kodlar

Bu aşamada Alice, kendisi dolaşıklık eşi hedef ve ışınlamak istediği durum olan kontrol kübiti olmak üzere *Kontrollü-not* kapısı ve ardından *Hadamard* uygular.

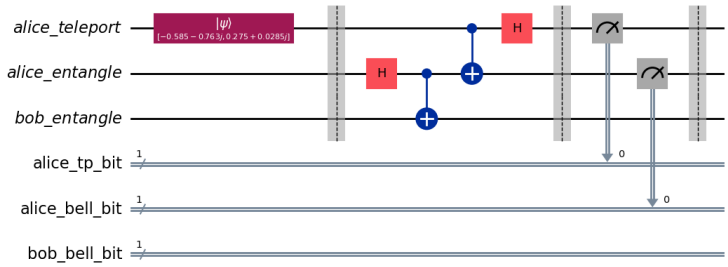
```
1 teleportation_circuit.cx(alice_teleport, alice_entangle
    )
2 teleportation_circuit.h(alice_teleport)
3 teleportation_circuit.barrier()
4 teleportation_circuit.draw(output='mpl')
```



3.1.5 Adım 3: Alice kendi kübitlerini ölçmesi ve ölçüm sonuçlarını Bob'a iletmesi

Bu aşamada Alice kendi elinde olan kübitleri ölçer ve ölçüm sonuçlarını Bob'a gönderir.

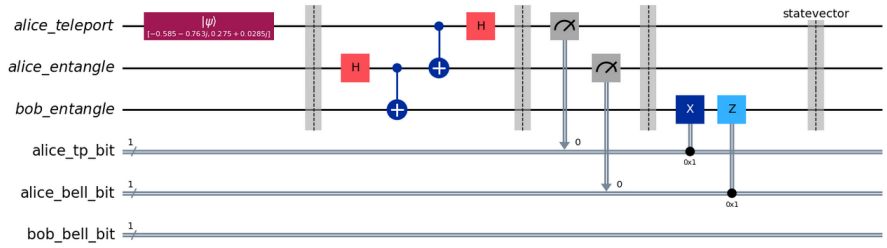
```
1 teleportation_circuit.measure(alice_teleport,
    alice_teleport_measure)
2 teleportation_circuit.measure(alice_entangle,
    alice_entangle_measure)
3 teleportation_circuit.barrier()
4 teleportation_circuit.draw(output='mpl')
```



3.1.6 Adım 4: Bob ölçüm sonuçlarına göre kendi kübitine X ve/veya Z uygulaması

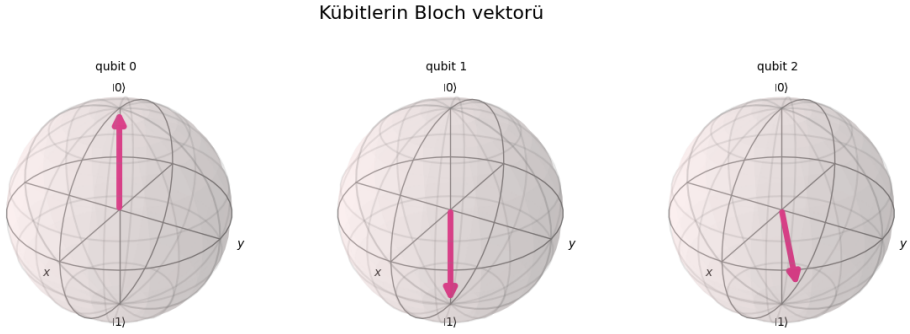
Ölçüm sonuçlarını alan Bob, bu verilere göre kendi elindeki kübitine kapı veya kapılar uygular. Kodda *c_if(bit, değer)* metotu eğer verilen bit değer ile aynı ise kapıyı uygula demektir.

```
1 teleportation_circuit.x(bob_entangle).c_if(
    alice_teleport_measure, 1)
2 teleportation_circuit.z(bob_entangle).c_if(
    alice_entangle_measure, 1)
3 teleportation_circuit.save_statevector()
4 teleportation_circuit.draw(output='mpl')
```



3.1.7 Adım 5: Bob'un elindeki kübitin incelenmesi

```
1 simulator = AerSimulator(method='statevector')
2 teleportation_circuit = transpile(teleportation_circuit
  , simulator)
3 result = simulator.run(teleportation_circuit).result()
4 statevector = result.get_statevector(
  teleportation_circuit)
5 plot_bloch_multivector(statevector, title='Kübitlerin
  Bloch vektorü')
```



3.2 Quantum Fourier Dönüşümü

Bu algoritma klasikte sinyal işlemede ve veri sıkça kullanılanmaktadır. Kuantumda ise dalga fonksiyonunun genlikleri üzerinde bu dönüşümü uygular.

3.2.1 Kurulum

İlk olarak Qiskit'ten gerekli metotları içe aktaralım.

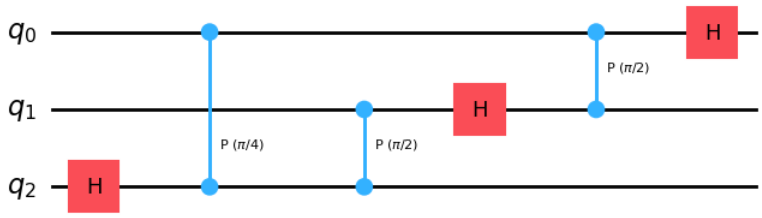
```
import numpy as np
from qiskit import QuantumCircuit, transpile, assemble
from qiskit_aer import AerSimulator
from qiskit import quantum_info
from qiskit.visualization import plot_histogram,
  plot_bloch_multivector
```

3.2.2 Adım 1: Devre hazırlama

Bu adımda genel bir kuantum fourier dönüşümü yöntemi yazacağız. Bunu yaparken *Kontrollü-faz(cp)* kapılarını farklı açılarda kullanacağız. Aşağıdaki fonksiyonu 3 kübit için test edelim ve devreyi yazdıralım. Aşağıda farklı açılarda kullandığımız bu *Kontrollü-faz* kapılarının aynı zamanda Z, T, S kapısı gibi özel isimleri bulunmaktadır.

```
def qft_olustur(circuit, n):
if n == 0:
  return circuit
n -= 1
circuit.h(n)
for qubit in range(n):
  circuit.cp(np.pi/2**(n-qubit), qubit, n)
qft_olustur(circuit, n)
return circuit
```

```
qubit_count = 3
fourier_circuit = QuantumCircuit(qubit_count)
qft_olustur(fourier_circuit, qubit_count)
fourier_circuit.draw(output='mpl')
```

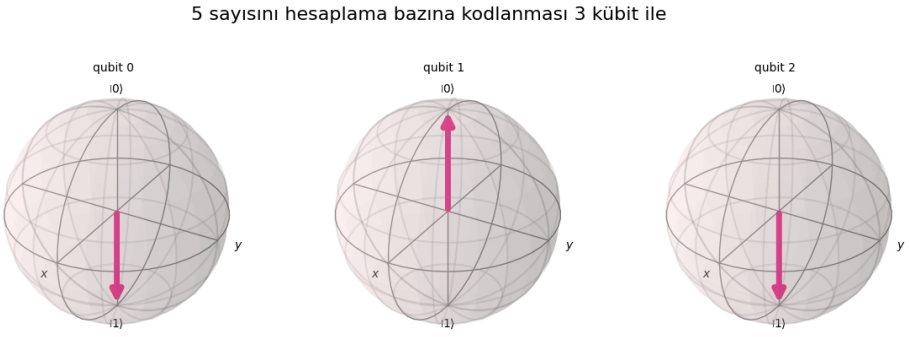


3.2.3 Adım 2: Simülasyon

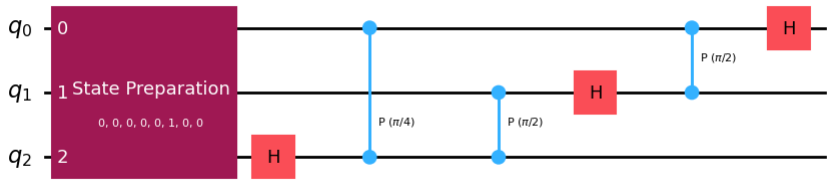
Kuantum fourier dönüşümünün etkisini anlamak için *bloch* küresinde yaptığı etkiyi anlamak gerekir. Bu nedenle 3 kübit üzerinde "5" rakamını QFT ile kübitlere kodlayacağız ve etkiyi gözlemleyeceğiz.

```
number = 5
sv = np.zeros(2**qubit_count)
sv[number] += 1
sv = quantum_info.Statevector(sv)
print(f'{number} sayısını hesaplama bazına kodlanırsa: ')
sv.draw(output='latex')
```

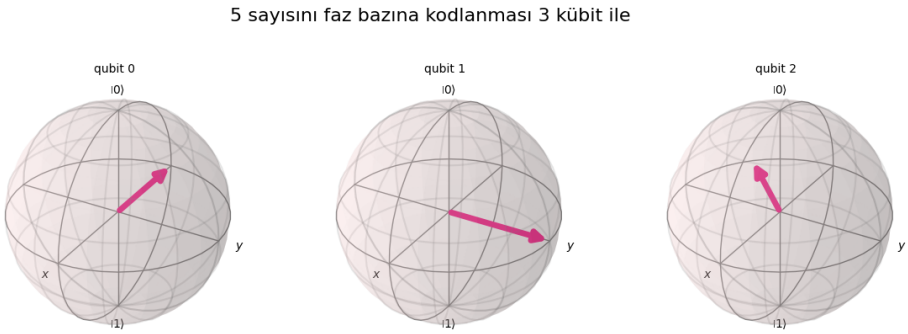
$|101\rangle$




```
fourier_circuit = QuantumCircuit(qubit_count)
fourier_circuit.prepare_state(sv, range(qubit_count))
qft_olustur(fourier_circuit, qubit_count)
fourier_circuit.draw(output='mpl')
```



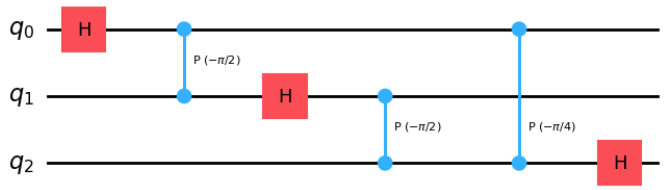

```
sav = quantum_info.Statevector(fourier_circuit)
sav.draw(output='latex')
```



3.2.4 Ters Quantum Fourier Dönüşümü

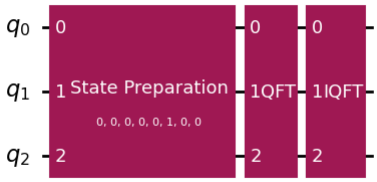
Ters kuantum fourier dönüşümü bir önceki fourier dönüşümünü adından da anlaşıldığı üzere tersler. Bir önceki kaldığımız noktadan devam ederek örneğimize ters kuantum fourier dönüşümünü uygulayalım. Aşağıdaki *ters_qft_olustur()* fonksiyonu ile devre oluşturabiliriz. Alsında yaptığımız bir fourier devresi oluşturmak ve *inverse()* yöntemi ile tersini bulmak olacak.

```
def ters_qft_olustur(n):
  circuit = QuantumCircuit(n)
  circuit = qft_olustur(circuit, n)
  circuit = circuit.inverse()
  return circuit

qubit_count = 3
fourier_circuit = ters_qft_olustur(qubit_count)
fourier_circuit.draw(output='mpl')
```



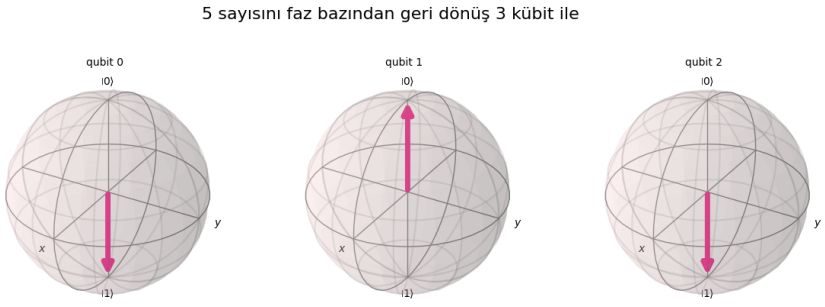
Şimdi ise fourier dönüşümünde kullandığımız örneği tekrar oluşturup devamına ters kuantum fourier dönüşümü ekleyelim ve kübitleri ölçelim.

```
number = 5
sv = np.zeros(2**qubit_count)
sv[number] += 1
sv = quantum_info.Statevector(sv)
print(f'{number} versi hesaplama bazına kodlanırsa: ')
sv.draw(output='latex')

fourier_circuit = QuantumCircuit(qubit_count)
fourier_circuit.prepare_state(sv, range(qubit_count))
qft_olustur(fourier_circuit, qubit_count)
fourier_circuit.draw(output='mpl')

fourier = QuantumCircuit(qubit_count)
fourier.prepare_state(sv, range(qubit_count))
qft = qft_olustur(QuantumCircuit(qubit_count),
  qubit_count)
qft.name = 'QFT'

ters_qft = ters_qft_olustur(qubit_count)
ters_qft.name = 'IQFT'

fourier.append(qft, range(qubit_count))
fourier.append(ters_qft, range(qubit_count))
fourier.draw(output='mpl')
```



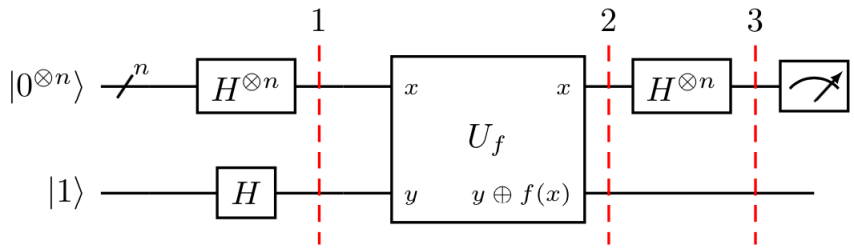
```
plot_bloch_multivector(sav, title=f'{number} sayısını faz bazına kodlanması {qubit_count} kübit ile')
```

OpenQasm simülatör ile simüle edip bulduğumuz ölçüm sonuçlarını histograma ekleyelim. Kuantum fourier dönüşümü öncesinde **5** rakamının ikili gösterimi olan **101** ile kodlamıştık. Tersleme sonucunda **101** ölçüm sonucunu beklemekteyiz.



3.3 Deutsch-Jozsa Algoritması

Bu algoritma tek bit ile çalışan Deutsch algoritmasının n bite uzatılmış versiyonudur. Bu algoritma girdi olarak gelen bitleri 0 veya 1' e dönüştüren gizli bir fonksiyona sahiptir. Kuantum bilgisayarlarda kodlamak için bu algoritma $|x>|y>$ durumunu $|x>|y + f(x)>$ durumuna dönüşümünü yapan bir $U_f(oracle)$ fonksiyonundan faydalanır. Algoritma 4 adımda tamamlanır. Aşağıdaki şekil bu adımları devre üzerinde gösterilmektedir.



3.3.1 Kurulum

Her zaman olduğu gibi bu adımda kütüphaneleri içe aktarıyoruz.

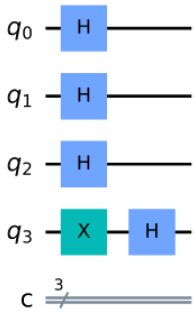
```
1 import numpy as np
2 from qiskit import QuantumCircuit, QuantumRegister,
   ClassicalRegister
3 from qiskit import transpile, assemble
4 from qiskit_aer import AerSimulator
5 from qiskit.visualization import plot_histogram
6 from qiskit import quantum_info
7 from qiskit.visualization import plot_bloch_multivector
```

3.3.2 Adım 1: Hadamard

Bu adımda tüm girdi qubitlerine *Hadamard* ve son kübite bir de X kapısı uyguluyoruz.

```
1 qubit_count=3
2 dj_circuit = QuantumCircuit(qubit_count+1, qubit_count)
3
4 for qubit in range(qubit_count):
5   dj_circuit.h(qubit)
6
7 dj_circuit.x(qubit_count)
```

```
dj_circuit.h(qubit_count)
dj_circuit.draw(output='mpl')
```



3.3.3 Adım 2: Oracle fonksiyonu

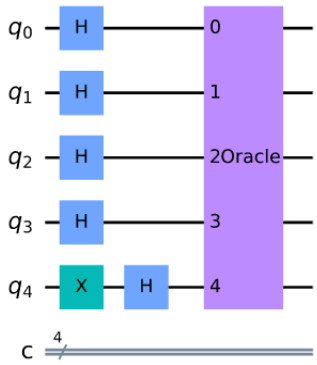
Bu adım bu algoritmasının kilit adımıdır. Bu noktada ya dengeli ya da sabit bir Oracle fonksiyonuna ihtiyaç vardır. Aşağıdaki koddaki metot bize istediğimiz Oracle fonksiyonunun devresini vermektedir.

```
def dj_oracle(case, n):
# Bu devre n+1 kübite sahip: girdi kübit boyutu,
# ve bir de cikti kübit
oracle_qc = QuantumCircuit(n+1)

# Dengeli oracle isteniyor ise:
if case == "balanced":
    b = np.random.randint(1,2**n)
# Ikili stringi formatlayalim.
b_str = format(b, '0'+str(n)+'b')

# Sonra, ilk X kapilarini koyalim. Ikili stringteki her
    rakam bir
# kübite denk geldigi için, string 1 ise X kapisi
   gerekir.
for qubit in range(len(b_str)):
  if b_str[qubit] == '1':
  oracle_qc.x(qubit)

# Cikti qubiti hedef olacak sekilde,
# tum kübitler ile CNOT
```

```
for qubit in range(n):
  oracle_qc.cx(qubit, n)

# Sonra, son X kapilarini koyalim.
for qubit in range(len(b_str)):
  if b_str[qubit] == '1':
    oracle_qc.x(qubit)

# Sabit oracle isteniyor ise:
if case == "constant":
# Ilk olarak hangi sabit cikti isteniyor rastgele
  secelim.
output = np.random.randint(2)

# Rasgele 1 gelirse, son kübite X uygulayalim.
if output == 1:
oracle_qc.x(n)

# Devreyi kapi yapar.
oracle_gate = oracle_qc.to_gate()
oracle_gate.name = "Oracle"
return oracle_gate
```

Bu yöntemi kullanarak bir Oracle fonksiyonunu devreye yerleştirelim. Burada dengeli veya sabit Oracle seçebilirsiniz.

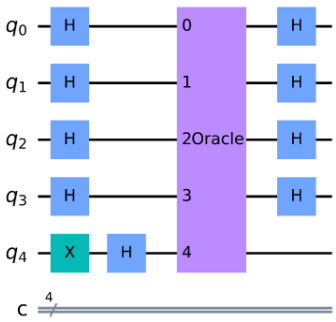
```
dj_circuit.append(dj_oracle("balanced", qubit_count),
  range(qubit_count+1))
dj_circuit.draw(output='mpl')
```



3.3.4 Adım 3: Tekrar Hadamard

Bu adımda çıktı kübiti hariç tüm kübitlere *Hadamard* kapısı uygulanır.

```
for qubit in range(n):
    dj_circuit.h(qubit)
dj_circuit.draw()
```



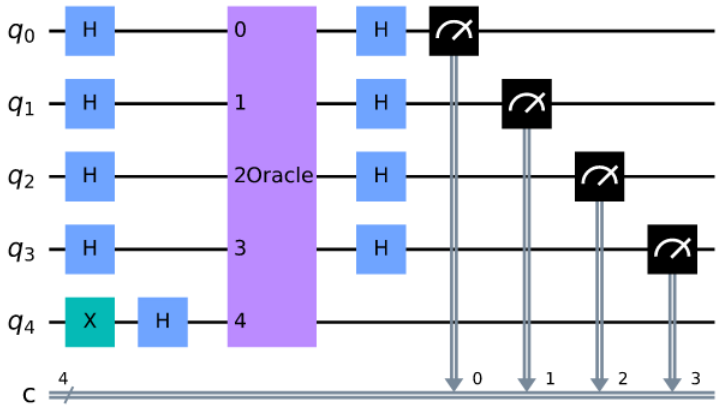
3.3.5 Adım 4: Ölçüm eklenmesi ve simülasyon

Bu adımda devrenin tüm girdi kübitlerine ölçüm eklenir ve devre simüle edilir.

```
sav = quantum_info.Statevector(dj_circuit)
sav.draw(output='latex')
```

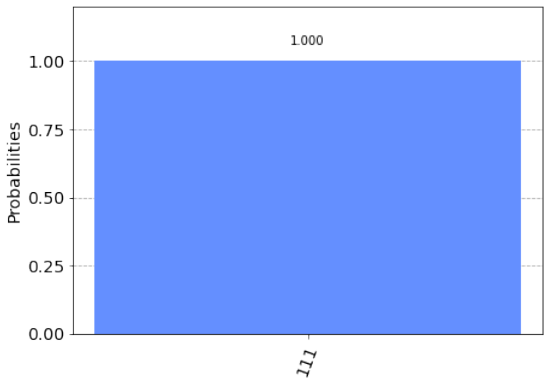
$-\frac{\sqrt{2}}{2}|0111\rangle + \frac{\sqrt{2}}{2}|1111\rangle$

```
for i in range(n):
    dj_circuit.measure(i, i)
dj_circuit.draw()
```



Devremiz tamamlandığına göre simüle edebiliriz. Simülasyon sağlayıcısı olarak Aer kullanacağız.

```
1 simulator = AerSimulator()
2 dj_circuit = transpile(dj_circuit, simulator)
3 result = simulator.run(dj_circuit).result().get_counts
   (0)
4 plot_histogram(result)
```



3.4 Shor Algoritması

Shor algoritmasının amacı tekrarlayan serilerin periyodunu bulmaktadır. Klasikte bu işlem zaman bakımından üstel artan olduğu için büyük asal rakamlarda maliyetlidir. Fakat bu işlem kuantum bilgisayarlarda daha imkanlı olması günümüzde asal sayılar ile korunan internetin güvenliği gibi alanları tehdit etmektedir.

3.4.1 Kurulum

Her zamanki gibi Qiskit'i içe aktararak başlıyoruz.

```
import numpy as np
import pandas as pd
from fractions import Fraction
from math import gcd

from qiskit import QuantumCircuit, QuantumRegister,
  ClassicalRegister
from qiskit import transpile, assemble
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit import quantum_info
from qiskit.visualization import plot_bloch_multivector
```

Her bir a ve N değeri için $U|y> = |aymodN>$ denkliğini sağlayan bir oracle fonksiyonuna ihtiyacımız vardır. Bizler bu örnek için $a = 7$ ve $N = 15$ olarak alacağız. 8 adet sayma kübiti ve 15'i ikili tabanda temsil edebilecek 4 adet kayıt kübiti gerekmektedir.

Note: Bu örnek 15 sayısı için geçerli bir devre oluşturmaktadır. Farklı sayılar için farklı devreler oluşturulması gerekir. Fakat a değeri $a \in [2, 7, 8, 11, 13]$ seçilebilir.

```
a = 7
N = 15

# 15 için 4 kübite ihtiyaç var (15='1111'). 2*4=8 adet
  kübit gerekir.
n_count = 8  # Sayma kübit sayısı
```

3.4.2 Adım 1: Hadamard

Bu adımda tüm sayma kübitlerine *Hadamard* uygulanır ve N=15 olacak şekilde kayıtçı kübitler başlatılır.

```
shor_circuit = QuantumCircuit(n_count + 4, n_count)

for q in range(n_count):
  shor_circuit.h(q)

# Bir adet anchilla kübitimizi |1> yapılır.
shor_circuit.x(3+n_count)

# Kontrollü a%15 kapılarını ekleme.
for q in range(n_count):
  shor_circuit.append(c_amod15(a, 2**q),
    [q] + [i+n_count for i in range(4)])

# Kuantum ters QFT eklenir.
shor_circuit.append(qft_dagger(n_count), range(n_count)
  )

# Ölçüm eklenir.
shor_circuit.measure(range(n_count), range(n_count))
shor_circuit.draw(output='mpl', fold=-1)
```

3.4.3 Adım 2: AmodN kapısı

Bu adımda 7mod15 için Oracle fonksiyonunu ekleyeceğiz.

```
def c_amod15(a, power):
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must be 2,7,8,11 or 13")
    U = QuantumCircuit(4)
    for iteration in range(power):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a == 11:
            U.swap(1,3)
```

```
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = "%i^%i mod 15" % (a, power)
    return U.control()
```

Yukarıdaki metot bizlere a değeri ve farklı kuvvetlerde bir kontrollü Oracle fonksiyonu üretmektedir. Daha sonra aşağıdaki gibi devreye bu kapıları ekleyelim.

```
# Do controlled-U operations
for q in range(n_count):
    qc.append(c_amod15(a, 2**q),
              [q] + [i+n_count for i in range(4)])
```

3.4.4 Adım 3: Ters QFT

Bu adımda devremizin sayma kübitlerine ters fourier dönüşümü uygulayacağız. Önceki bölümlerden de hatırlayacağız üzere hızlıca bir ters kuantum fourier dönüşüm metotu yazalım.

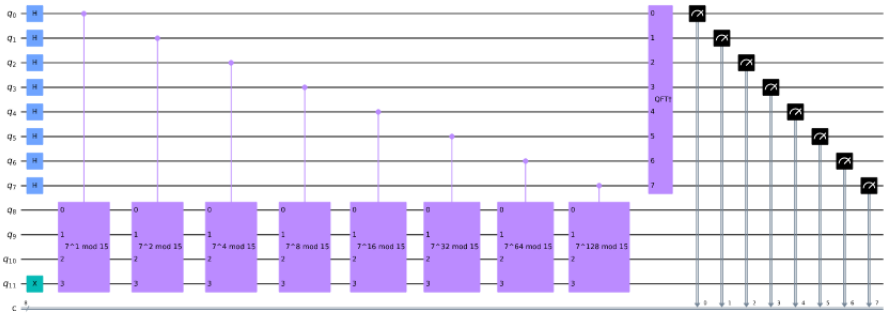
```
def qft_dagger(n):
    qc = QuantumCircuit(n)
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "TQFT"
    return qc

qc.append(qft_dagger(n_count), range(n_count))
qc.draw(fold=-1)
```

3.4.5 Adım 4: Ölçüm

Bu adımda tüm sayma kübitleri ölçüp oluşan devreyi simüle edeceğiz.

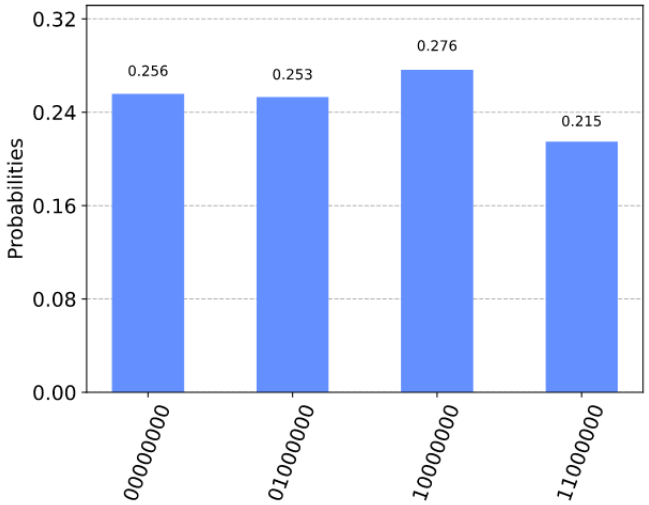
```
qc.measure(range(n_count), range(n_count))
qc.draw(fold=-1)
```

Çizim metotu olan *draw()* başka parametreler alabilmektedir. Bu örneğimizde *fold* parametresi devreyi katlamadan yazdırmaktadır. *"mpl"* parametresi ise bize renkli bir çıktı verir.



Şimdi aşağıdaki kod ile simüle ederek histogram üzerinde sonuçlarımıza bakalım. Bu örnekte Aer simülatörünü seçtik.

```
simulator = AerSimulator()
shor_circuit = transpile(shor_circuit, simulator)
counts = simulator.run(shor_circuit).result().
  get_counts(0)
plot_histogram(counts)
```



Bu adım ile tüm kuantum işlemler sona erer. Bundan sonraki adım klasik olarak devam edecektir.

3.4.6 Adım 5: Periyot bulma

Bu adımda ilk olarak *pandas* kullanılarak çıkan sonuçların belirttiği fazları incelemekle başlayacağız.

```
rows, measured_phases = [], []
for output in counts:
    decimal = int(output, 2)  # Convert (base 2) string to decimal
    phase = decimal/(2**n_count)  # Find corresponding eigenvalue
    measured_phases.append(phase)
    rows.append([f"{output}(bin) = {decimal:>3}(dec)",
                 f"{decimal}/{2**n_count} = {phase:.2f}"])
headers=["Register Output", "Phase"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

	Register Output	Phase
0	00000000(bin) = 0(dec)	0/256 = 0.00
1	01000000(bin) = 64(dec)	64/256 = 0.25
2	11000000(bin) = 192(dec)	192/256 = 0.75
3	10000000(bin) = 128(dec)	128/256 = 0.50

Şimdi bulunan bu fazları *Fraction* kullanarak denk geldiği **r** periyodunu bulalım.

```
rows = []
for phase in measured_phases:
    frac = Fraction(phase).limit_denominator(15)
    rows.append([phase, f"{frac.numerator}/{frac.denominator}", frac.denominator])

headers=["Phase", "Fraction", "Guess for r"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

	Phase	Fraction	Guess for r
0	0.00	0/1	1
1	0.25	1/4	4
2	0.75	3/4	4
3	0.50	1/2	2

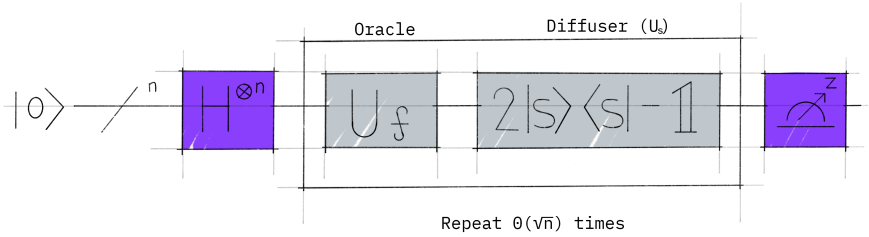
Şimdi bulduğumuz **r=4** periyodunun **N=15**'den küçük ortak bölenlerini bulalım.

```
1 # En cok rastlanan periyot 4.
2 r = 4
3 guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
4 print(guesses)
5
6 ... [3, 5]
```

Böylelikle 15 sayısının asal çarpanlarını kuantum bilgisayarlar üzerinde periyot hesaplayarak bulmuş olduk.

3.5 Grover Algoritması

Kuantum bilgisayarların bir diğer avantajı ise veritabanlarında hızlı aramalar gerçekleştiriyor olmasıdır. Grover algoritması ile çok hızlı şekilde veriler arasında aranan bir veriyi bulabiliriz. $U_w|x> = -(1)f(x)|x>$ Oracle fonksiyonu ile genlik arttıran diffuser bu algoritmanın omurgasını oluşturur.



3.5.1 Kurulum

Her zaman olduğu gibi Qiskit ve gerekli kütüphaneler içe aktarılarak başlanır.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 from qiskit import QuantumCircuit, QuantumRegister,
    ClassicalRegister
5 from qiskit import transpile, assemble
6 from qiskit_aer import AerSimulator
7 from qiskit.visualization import plot_histogram
```

```
from qiskit import quantum_info
from qiskit.visualization import plot_bloch_multivector
```

3.5.2 Adım 1: Hadamard

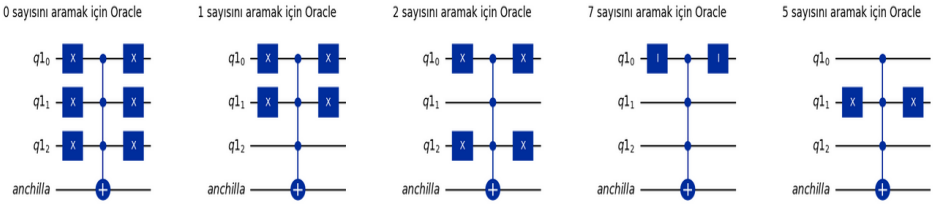
Bu adımda devreyi oluşturup her kübite *Hadamard* uygulayacağız. Biz bu örneği 3 kübit ile yapacağız ve ilk kübiti aranan olarak işaretleyeceğiz.

```
grover_circuit = QuantumCircuit(qubit_count)
for i in range(qubit_count):
    grover_circuit.h(i)
```

3.5.3 Adım 2: Oracle

Bu adımda aranan değer (bu örnekte 5) için bir Oracle fonksiyonu ekleyeceğiz.

```
qubit_count = 3
oracle = QuantumCircuit(qubits, anchilla, name='Oracle'
  )
oracle.x(1)
oracle.mcx(list(range(qubit_count)), anchilla)
oracle.x(1)
oracle.draw(output='mpl')
```

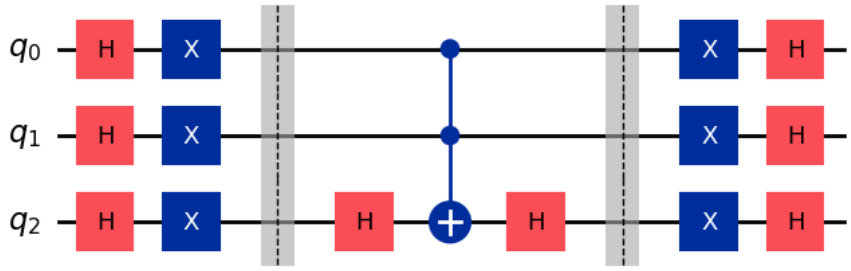


3.5.4 Adım 3: Diffuser

Bu adımda 3 kübit üzerinde etkili bir diffuser devresi oluşturup tasarıma ekleyeceğiz.

Note: Çok kontrollü X kapısı kuantum devreye QuantumCircuit().mcx(list(kontrol-kübitler), hedef-kübit) kullanarak eklenebilir.

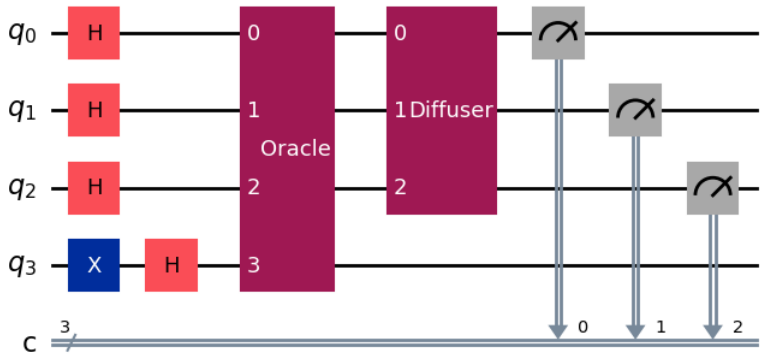
```
diffuser_circuit = QuantumCircuit(qubit_count, name='Diffuser')

# Tam süperposizyon oluştur (H kapısı)
for qubit in range(qubit_count):
diffuser_circuit.h(qubit)

# |00..0> -> |11..1> dönüşümü yap (X kapısı)
for qubit in range(qubit_count):
  diffuser_circuit.x(qubit)

diffuser_circuit.barrier()
# Multi kontrollü-X kapısı ekle (MCX)
diffuser_circuit.h(qubit_count-1)
diffuser_circuit.mcx(list(range(qubit_count-1)), qubit_count-1)
diffuser_circuit.h(qubit_count-1)

diffuser_circuit.barrier()
# |11..1> -> |00..0> dönüşümü (X kapısı)
for qubit in range(qubit_count):
  diffuser_circuit.x(qubit)

# Süperpozisyondan geri dön (H-kapısı)
for qubit in range(qubit_count):
  diffuser_circuit.h(qubit)

# Devreyi çiz
diffuser_circuit.draw(output='mpl')
```



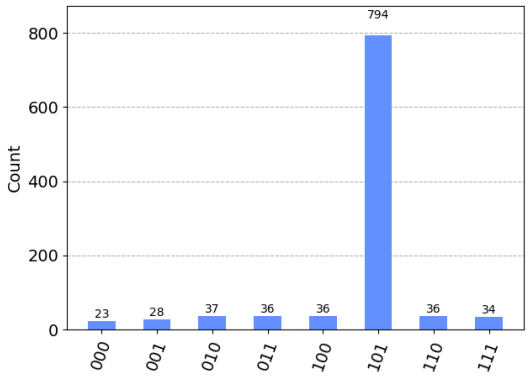
3.5.5 Adım 4: Ölçüm

Bu adımda devredeki tüm kübitlere ölçüm ekleyip simüle edeceğiz.

```
grover_circuit = QuantumCircuit(qubit_count+1,
  qubit_count)

for i in range(qubit_count):
  grover_circuit.h(i)

grover_circuit.x(qubit_count)
grover_circuit.h(qubit_count)

# Adım sayısı en az sqrt(N) kadar seçilmeli
step_count=1
for i in range(step_count):
  grover_circuit.append(oracle, range(qubit_count+1))
  grover_circuit.append(diffuser_circuit, range(
  qubit_count))

grover_circuit.measure(range(qubit_count), range(
  qubit_count))
grover_circuit.draw(output='mpl')
```



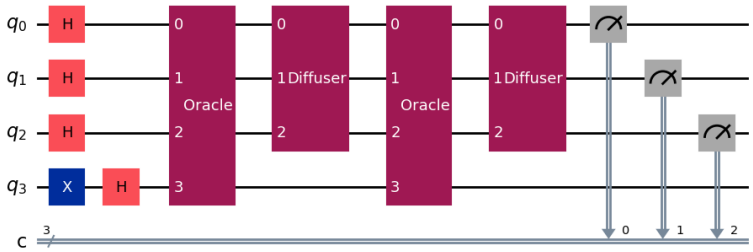

```
simulator = AerSimulator()
grover_circuit = transpile(grover_circuit, simulator)
result = simulator.run(grover_circuit).result().
  get_counts(0)
plot_histogram(result)
```

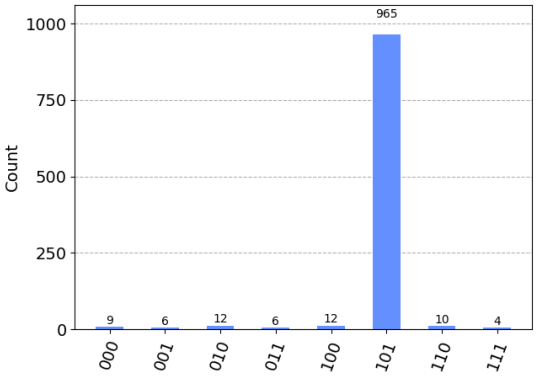


Yukarıdaki figürde görüleceği üzere aradığımız çıktı "5" sayısı en çok hesaplanmaktadır.

3.5.6 Adım 4.1: 2 adımda Grover Arama Algoritması

İlk örnekte 1 adım Grover arama algoritması gerçekleştirildi ve %79.4 oranında 5 sayısı elde edildi. İki adımda ise devre ve simülasyon sonuçları aşağıdaki biçimdedir.

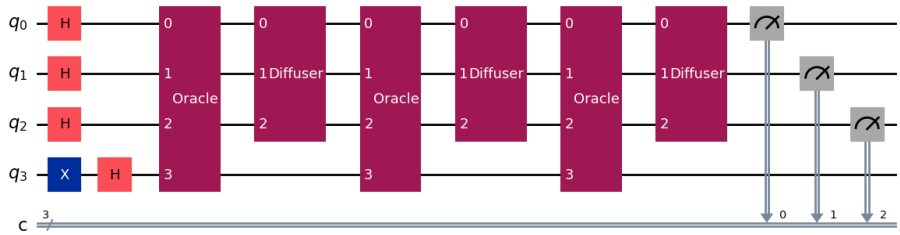


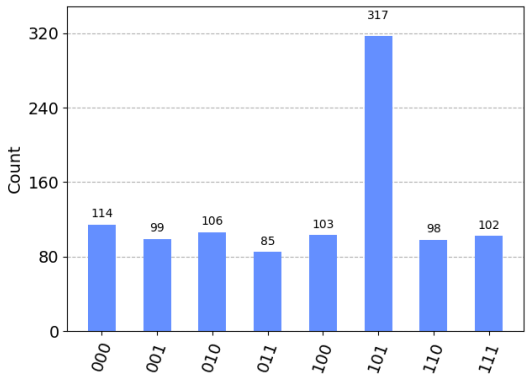


İlk örneğe göre 2 adımda Grover daha iyi (%96.5) sonuç vermektedir. Bunun sebebi ise adım sayısının $\sqrt{5} \approx 2.23$ olmasıdır.

3.5.7 Adım 4.2: 3 adımda Grover Arama Algoritması

İlk örnekte 2 adım Grover arama algoritması gerçekleştirildi ve %79.4 oranında 5 sayısı elde edildi. Sonra iki adımda ise %96.5 başarı elde etmiştir. Üç adımda ise devre ve simülasyon sonuçları aşağıdaki biçimdedir.

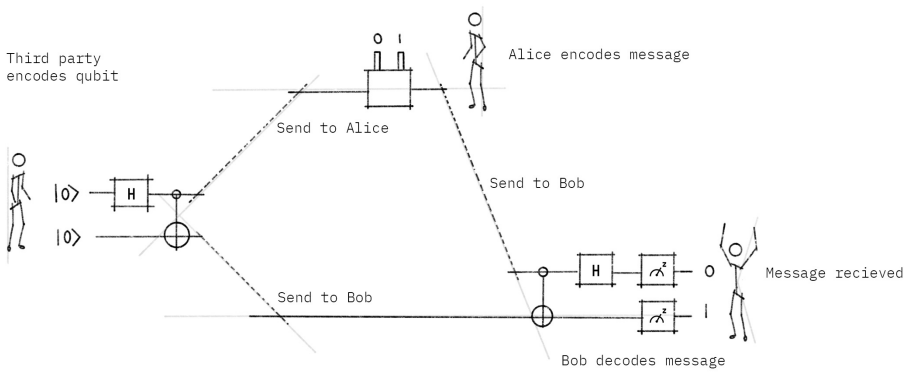




İlk örneklere göre 3 adımda Grover daha kötü (%31.7) sonuç vermektedir. Bu nedenle Grover optimum adım sayısında gerçekleştirilmelidir.

3.6 Süperyoğun Kodlama Algoritması

Bu algoritma ışınlanma algoritmasının bir gelişmiş versiyonudur. Bu algoritma bir Bell çifti ile 2 klasik biti göndermektedir. Aşağıdaki görselde algoritmanın nasıl çalıştığı vurgulanmaktadır.



3.6.1 Kurulum

Her zamanki gibi kodlamaya başlamadan önce gerekli kütüphaneler içe aktarılmalıdır.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 from qiskit import QuantumCircuit, QuantumRegister,
  ClassicalRegister
5 from qiskit import transpile, assemble
6 from qiskit_aer import AerSimulator
7 from qiskit.visualization import plot_histogram
8 from qiskit import quantum_info
9 from qiskit.visualization import plot_bloch_multivector
```

3.6.2 Adım 1: Bell çiftinin hazırlanması

Bu adımda üstteki görselde görüleceği üzere 3.taraf kişi tarafından bir Bell çifti oluşturulup Alice ile Bob'a gönderilmektedir.

```
def create_bell_pair():
  alice = QuantumRegister(1, name='Alice')
  bob = QuantumRegister(1, name='Bob')
  qc = QuantumCircuit(alice, bob)
  qc.h(0)
  qc.cx(0, 1)
  return qc

# Bell çiftini kim oluşturduğu önemli değil,
# fakat karşı tarafa bir eşini göndermelidir.
create_bell_pair().draw(output='mpl')
```

3.6.3 Adım 2: Mesajın kodlanması

Bu adımda Alice, göndermek istediği 2 bite göre kendisine gelen kübite kapılar uygular ve Bob'a gönderdiği varsayılır.

```
def encode_message(qc, qubit, msg):
  if len(msg) != 2 or not set([0,1]).issubset({0,1}):
      raise ValueError(f"mesaj 2 bitlik olmalı!")

  # Mesajın ikinci biti 1 ise Alice X uygular.
  if msg[1] == "1":
  qc.x(qubit)

  # Mesajın ilk biti 1 ise Alice Z uygular.
  if msg[0] == "1":
    qc.z(qubit)
  return qc

  # Örnek olarak mesaj='11' ise Alice'in uyguladığı kap
  ılar
encode_message(QuantumCircuit(QuantumRegister(1, 'Alice
  '), QuantumRegister(1, 'Bob')), 0, '11').draw(output=
  'mpl')
```

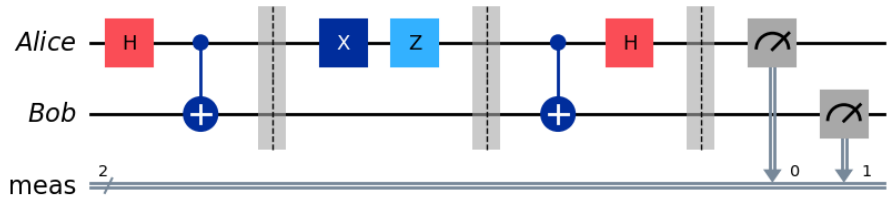
3.6.4 Adım 3: Bob'un mesajı çözmesi

Bob bu adımda 2 kübite sahiptir. İlk kübit Bell çiftinin kendisine ait olan kübit ve diğeri ise Alice'in bir önceki adımda gönderdiği kübittir. Bob aşağıdaki kuantum kapıları uyguladıktan sonra ölçüm yapması gerekir.

```
def decode_message(qc):
  qc.cx(0, 1)
  qc.h(0)
  return qc

# Alice'ten kodlanış kübiti alan Bob'un elinde 2 kübit vardır.
# Bob aşağıdaki gibi kapıları uygulayark çözümleme yapar.
decode_message(QuantumCircuit(QuantumRegister(1, 'Alice'), QuantumRegister(1, 'Bob'))).draw(output='mpl')
```

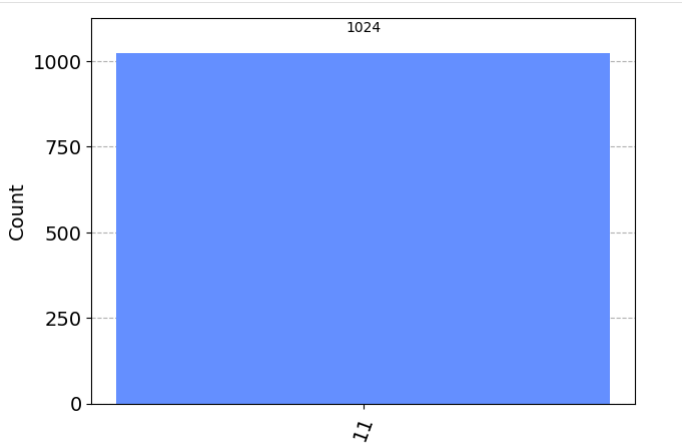
Şimdi oluşan tüm devreye bir göz atalım.

```
sdc_circuit = create_bell_pair()
sdc_circuit.barrier()

message = '11'
sdc_circuit = encode_message(sdc_circuit, 0, message)
sdc_circuit.barrier()

sdc_circuit = decode_message(sdc_circuit)
sdc_circuit.measure_all()

sdc_circuit.draw(output='mpl')
```



Devreyi aşağıdaki kod ile simüle ederek ölçüm sonuçlarına bakalım.

```
simulator = AerSimulator()
sdc_circuit = transpile(sdc_circuit, simulator)
result = simulator.run(sdc_circuit).result().get_counts(0)
plot_histogram(result)
```



Yukarıdaki şekilden görüleceği üzere Alice'in gönderdiği **10** mesajını Bob ölçüm sonuçlarında elde etmiş oldu.

3.7 Simon Algoritması

Simon algoritması bir karakutu fonksiyonu kullanarak gizli bir mesajı kübitlere kodlamaktadır.

3.7.1 Kurulum

Her algoritma kodlamanın başlangıcında olduğu gibi Qiskit ve gerekli kütüphaneleri içe aktararak başlarız.

```
import matplotlib.pyplot as plt
import numpy as np

from qiskit import QuantumCircuit, QuantumRegister,
  ClassicalRegister
from qiskit import transpile, assemble
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit import quantum_info
from qiskit.visualization import plot_bloch_multivector
```

3.7.2 Adım 1: Hadamard

Bu adımda gizli mesajımızı seçip gereken kübit sayısını hesaplayacağız. Sonra kübitlerin ilk yarısına *Hadamard* uygulayacağız.

```
b = '110'
n = len(b)

# 2*n adet kübite ihtiyaç var.
simon_circuit = QuantumCircuit(n*2, n)

# Süperpozisyon'a geçirilir.
simon_circuit.h(range(n))
simon_circuit.barrier()
```

3.7.3 Adım 2: Simon Oracle

Bu adımda Simon algoritması için gereken karakutu fonksiyonunu ekleyeceğiz.

```
simon_circuit.cx(0, 3)
simon_circuit.cx(1, 4)
simon_circuit.cx(2, 5)
```

```
4 simon_circuit.cx(1, 4)
5 simon_circuit.cx(1, 5)
6 simon_circuit.barrier()
```

3.7.4 Adım 3: Tekrar Hadamard

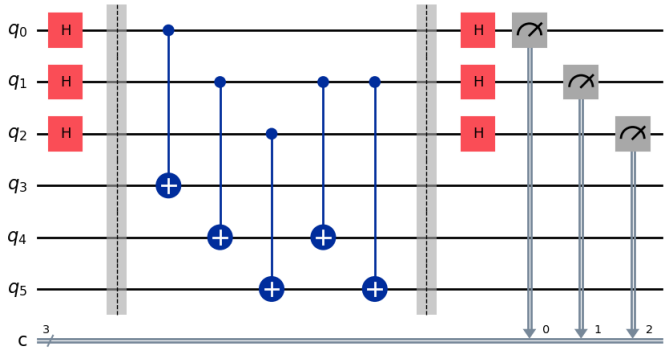
Bu adımda kubilerin ilk yarısı yani girdi kübitlere tekrardan *Hadamard* kapısını uyguluyoruz.

```
1 simon_circuit.h(range(n))
```

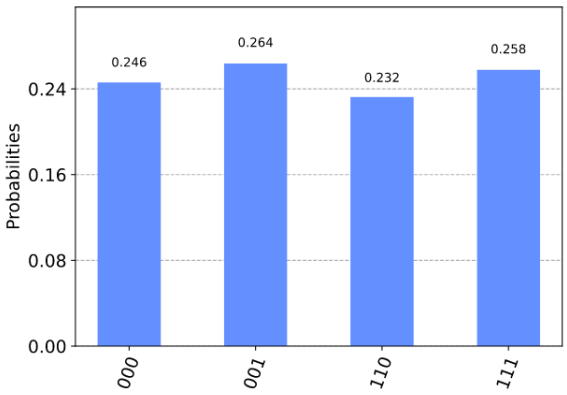
3.7.5 Adım 4: Ölçüm

Bu adımda devredeki tüm kübitlere ölçüm ekleyeceğiz ve devreyi simüle edeceğiz.

```
1 simon_circuit.measure(range(n), range(n))
2 simon_circuit.draw(output='mpl')
```




```
1 simulator = AerSimulator()
2 simon_circuit = transpile(simon_circuit, simulator)
3 counts = simulator.run(simon_circuit).result().
  get_counts(0)
4 plot_histogram(counts)
```



Şimdi yukarıdaki sonuçları kullanarak gizli mesaj **b**'yi bulalım.

```
def bdotz(b, z):
    accum = 0
    for i in range(len(b)):
        accum += int(b[i]) * int(z[i])
    return (accum % 2)

for z in counts:
    print( '{}.{} = {} (mod 2)'.format(b, z, bdotz(b,z)
  ))
```

```
110.001 = 0 (mod 2)
110.000 = 0 (mod 2)
110.111 = 0 (mod 2)
110.110 = 0 (mod 2)
```

Bu sonuçlar kullanılarak $b = 110$ aşağıdaki denklemleri çözerek elde edilebilir:
Örnek olarak çözülen `001` için:

$$b \cdot 001 = 0$$

$$(b_2 \cdot 0) + (b_1 \cdot 0) + (b_0 \cdot 1) = 0$$

$$\cancel{(b_2 \cdot 0)} + \cancel{(b_1 \cdot 0)} + (b_0 \cdot 1) = 0$$

$$b_0 = 0$$

$$b \cdot 111 = 0$$

$$(b_2 \cdot 1) + (b_1 \cdot 1) + \cancel{(0 \cdot 1)} = 0$$

$$(b_2 \cdot 1) + (b_1 \cdot 1) = 0$$

Bu sonuçlar kullanılarak:
$b_2 = b_1 = 0, \quad b = 000$
veya
$b_2 = b_1 = 1, \quad b = 110$
bulunur.

Yukarıdaki metot bizlere tüm $b.z$ iç çarpım sonuçlarını göstermektedir. Buradan **b**'nin **110** olduğunu doğrulayabiliriz.