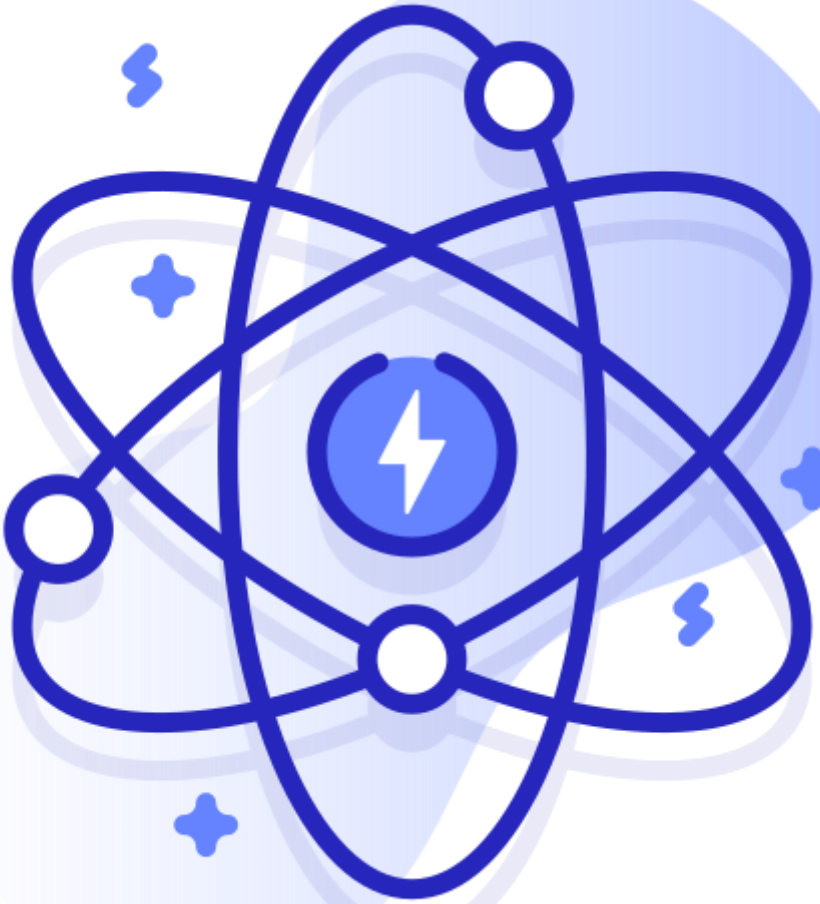


파이썬과 NUMPY를 이용한 양자컴퓨팅 시뮬레이션 소프트웨어 개발 및 실험

요약

파이썬과 넘파이를 이용하여 양자컴퓨팅 시뮬레이션 소프트웨어를 제작하고 이를 이용하여 다양한 양자알고리즘들을 실행 시켜보며 고전 컴퓨터에서 양자 컴퓨팅을 시뮬레이션 할 수 있는지 실험한다

<프로젝트 설명>

- 프로젝트 개요: 파이썬과 넘파이를 이용하여 양자컴퓨팅 시뮬레이션 소프트웨어를 제작하고 이를 이용하여 다양한 양자알고리즘들을 실행 시켜보며 고전 컴퓨터에서 양자 컴퓨팅을 시뮬레이션 할 수 있는지 실험한다
- 프로그램 명: Quself.py (Quantum Computing Library made by myself 의 줄임말)
- 지원하는 양자 게이트: 단일 게이트 12개(I, H, X, Y, Z, S, T, U3, RX, RY, RZ, $\sqrt{NOT}$게이트), 다중 큐비트 게이트 6개(CNOT, CY, CZ, CS, SWAP, TOFFOLI 게이트)를 모두 합쳐 총 18개의 양자 게이트 지원

<주요 기능>

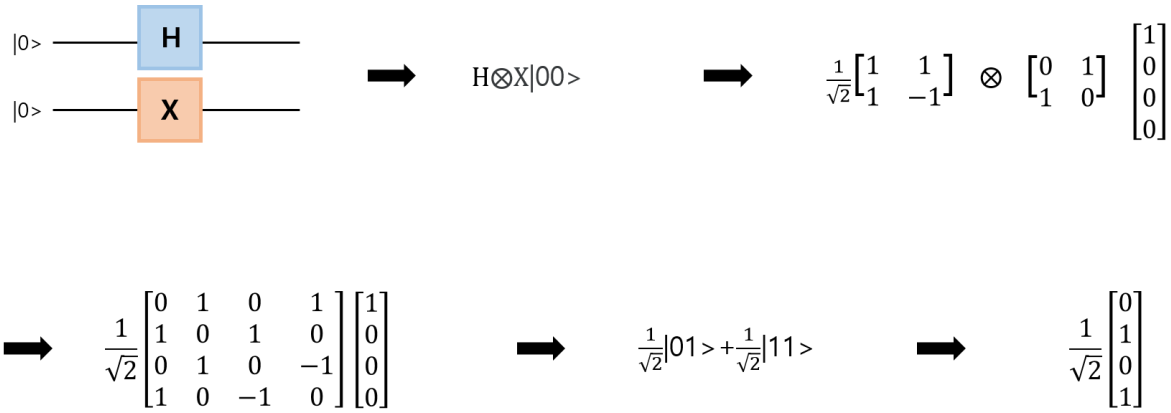
1. 양자 회로를 초기화 하고 양자 상태에 다양한 양자 게이트를 적용시킬 수 있다.
2. 양자상태를 측정하고 시각화 할 수 있다.
3. 그로버 알고리즘, 도이치-조사 알고리즘, 양자 난수 생성 알고리즘을 간단히 적용할 수 있다.

<소프트웨어 작동 방식 및 실험 진행 방식>

- 양자 컴퓨터의 계산 과정을 수학적인 관점에서 본다면 선형대수적으로 작동함. 양자 상태를 나타내는 벡터에 양자 게이트에 대응하는 행렬을 곱하여 각 상태들에 대한 확률진폭에 변화가 이루어 지고 해당 확률진폭의 제곱을 확률로 하여 큐비트를 측정했을 시 그 확률에 기반하여 값이 결정 됨. 이 프로젝트에서는 이러한 계산 과정을 그대로 코드로 구현함.
- 단일 양자게이트: 단일 양자게이트에 해당되는 행렬을 numpy array로 정의
- 다중 양자게이트: 일반적으로 소개된 행렬은 연속된 두 개 이상의 큐비트에 대한 것이므로 불연속적인 두 개 이상의 큐비트에는 적용 불가. 따라서 양자 게이트의 역할을 수행해주는 행렬을 적용하려는 큐비트에 따라 적절히 생성해주는 수식을 찾아 해당 수식을 코드로 구현하여 numpy array를 생성
- 양자 알고리즘: 이 프로젝트에서는 두 가지 방식으로 양자 알고리즘(양자 난수 생성, 그로버, 도이치-조사)이 실행 되는지 실험함. 1. 알고리즘에 적절한 양자 게이트들을 하나씩 적용 시켜가며 실험하는 방법. 2. 양자 게이트들을 적용하지 않고 알고리즘을 작동시키는데 필요한 행렬을 수식을 기반으로 생성하여 양자 상태에 적용시켜 주는 함수를 제작하여 간편하게 양자 알고리즘을 적용할 수 있는 방법
- 시뮬레이션의 한계: 고전 컴퓨터에서 단순히 양자컴퓨터의 수학적 계산 과정을 그대로

구현 할 시 발생하는 한계에 대해 실험함. 0번째 큐비트에 H게이트(하다마드 게이트)를 적용했을 시 최대 몇 개의 큐비트까지 생성했을 때 속도 문제가 나지 않을지 실험.

- 추가 실험: 생성한 양자난수를 mcm(몬테카를로 매소드)에 사용하여 원주율 구하기에 도전한다.



위 계산 과정을 넘파이로 구현함.

<프로그램 코드 구성>

회로 초기화: 인스턴스 생성시 회로의 큐비트 수 n 입력, 입력된 n을 통해 클래스 생성자에서 텐서곱을 활용해 $|0>^{\otimes n}$의 벡터(numpy array) 생성

```
self.one_qubit_zero = np.array([[1],[0]]) # |0>
self.base = np.array([[1]])
for i in range(n):
    self.base = np.kron(self.base,self.one_qubit_zero)
```

단일 큐비트 구현: 각 게이트에 해당하는 행렬을 넘파이를 통해 구현, 단일 큐비트 게이트를 적용할 회로 번호를 입력 받아 해당 회로에는 적용할 게이트를, 나머지 회로에는 항등 게이트(I gate)를 텐서곱을 하여 현재 양자 상태에 행렬곱을 통해 적용

Ex) 3큐빗 회로에서 두 번째 큐비트에 하다마드 게이트 적용 -> $I\otimes H\otimes I\ |\Psi>$

```
I_gate = np.array([[1,0],[0,1]])
H_gate = (1/sqrt(2))*np.array([[1,1],[1,-1]])
X_gate = np.array([[0,1],[1,0]])
Y_gate = np.array([[0,-1j],[1j,0]])
Z_gate = np.array([[1,0],[0,-1]])
```

Ex) H_gate를 행렬로 표현하면 $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$

```
def I(self,n):
    basis = self.make_uf_matrix(n,self.I_gate)
    self.base = basis.dot(self.base)
```

단일 게이트 적용 함수, make_uf_matrix는 현 양자상태에 곱할 행렬을 생성하는 함수 (앞의 예에서 I⊗H⊗I를 만드는 함수이다.)

다중 큐비트 구현: CNOT, CY, CZ, CS는 단일 게이트 NOT, Y, Z, S를 적용할 target 큐비트와 적용할 기준이 되는 control 큐비트를 지정해 주어야 함. (control이 1일 경우 target의 큐비트에 x, y, z, s를 적용한다) 두 큐비트에 control과 target을 적용할 때의 행렬을 생성해내는데 필요한 수식을 이용하여 구현

<행렬 ω1, ω2를 [1], 큐비트 번호를 x, 적용하려는 단일 게이트를 R이라 할 때 Controlled 계열의 수식>

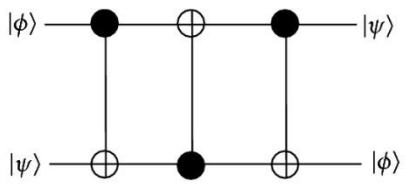
$$\omega 1 = \begin{cases} \omega 1 \otimes I, & x \neq control \\ \omega 1 \otimes |0><0|, & x = control' \end{cases} \quad \omega 2 = \begin{cases} \omega 2 \otimes I, & x \neq control \text{ and } x \neq tartget \\ \omega 1 \otimes |1><1|, & x = control \\ \omega 1 \otimes R, & x = tartget \end{cases}$$

M = ω1 + ω2 (M은 구하려는 행렬)

수식 적용의 예) CNOT(1,3) |0><0|⊗I⊗I + |1><1|⊗I⊗X -> control에는 각각 0과 1의 외적을 곱해주고 첫 항에는 control을 제외한 나머지 큐비트에 I 게이트를 두 번째 항에는 target에는 적용할 단일 게이트를 텐서곱을 해주고 나머지는 I 게이트를 적용시켜 두 식을 더한 행렬을 현재 양자상태에 곱해주는 방식을 취함.

```
for i in range(self.n_resister):
    if i == control:
        basis1 = np.kron(basis1,self.outer_zero)
    else:
        basis1 = np.kron(basis1,self.I_gate)
for i in range(self.n_resister):
    if i == control:
        basis2 = np.kron(basis2,self.outer_one)
    elif i == target:
        basis2 = np.kron(basis2,U)
    else:
        basis2 = np.kron(basis2,self.I_gate)
return basis1+basis2
```

SWAP게이트의 경우 연속된 두 큐비트에 대한 행렬은 쉽게 구할 수 있으나 연속 되지 않는 두 큐비트에 대한 행렬은 쉽게 구할 수 없음. 대신 CNOT게이트 3개를 이용해서 swap 게이트와 같은 효과를 낼 수 있음. Swap 함수를 실행하면 다음의 회로를 적용하는 방식으로 구현함..




```
def SWAP(self,target1,target2):
    if(target1==target2):
        raise ValueError("targets are same")
    if(target1>target2):
        target1,target2 = target2,target1
    self.CNOT(target1,target2)
    self.CNOT(target2,target1)
    self.CNOT(target1,target2)
```

측정 시뮬레이션: 양자컴퓨터를 가지고 있지 않기에 측정이란 행위를 직접 구현할 수 없지만 수학적으로 시뮬레이션 해볼 수는 있음. 각 양자상태의 확률 진폭의 제곱은 측정을 했을 때 그 양자 상태가 나올 확률을 의미함. 이를 바탕으로 각 양자 상태의 확률 제곱을 리스트에 넣고 해당

리스트의 값을 확률로 취하여 랜덤으로 값을 뽑아주는 함수를 구현. 측정 횟수를 지정할 수도 있으며 지정 횟수만큼 측정 시뮬레이션을 진행하도록 구현.

$$\alpha|0>+\beta|1> \Rightarrow \alpha^2 + \beta^2 = 1$$

```
def Measure(self,shot=1):
    probabilities = []
    result = [0]*2**self.n_resister
    for i in range(len(self.base)):
        probabilities.append(abs(self.base[i][0])**2)
    x = np.arange(2**self.n_resister)
    for i in range(shot):
        b = np.random.choice(x, 1, replace=False, p=probabilities)
        result[int(b)] += 1
    return result
```

시각화: 2가지 종류의 시각화 기능을 matplotlib을 통해 구현

1. 현재 양자 상태의 확률진폭 및 확률을 시각화
2. 측정 결과를 인자로 받아 n번의 측정 결과를 시각화

```
# Visualization the statevector and Probabilities
def graph(self):
    fig = plt.figure(figsize=(10,4))
    statevector = []
    probabilities = []
    for i in range(len(self.base)):
        statevector.append(self.base[i][0])
        probabilities.append(abs(self.base[i][0])**2) #확률진폭의 제곱
    x = np.arange(2**self.n_resister)
    ax = fig.add_subplot(1,2,1)
    ax2 = fig.add_subplot(1,2,2)
    ax.bar(x,statevector)
    ax2.bar(x,probabilities,color = 'r')
    ax.set_xlabel("value",size=14)
    ax2.set_xlabel("value",size=14)
    ax.set_ylabel("amplitude",size=14)
    ax2.set_ylabel("probabilities",size=14)
    ax.set_title("Statevector")
    ax2.set_title("Probabilities")
    ax2.set_ylim([0, 1])
    plt.show()
```

```
def M_graph(self,result):
    x = np.arange(2**self.n_resister)
    y = result
    plt.bar(x,y)
    plt.show()
```

<작동 실험>

- 2큐빗 회로에서 $H\otimes H|00>$을 실행 해 봄
- 기대하는 결과: $H\otimes H|00> = H\otimes H|0>\otimes|0> = H|0>\otimes H|0> = \frac{1}{\sqrt{2}}(|0>+|1>)\otimes\frac{1}{\sqrt{2}}(|0>+|1>) = \frac{1}{2}(|00>+|01>+|10>+|11>)$로 확률진폭은 $\frac{1}{2}$로 동일하게 나오고 확률은 $\frac{1}{4}$로 동일하게 나오는 결과를 기대함.

```
import Quself as qs #제작한 소프트웨어를 라이브러리 화 함
qs.version
```

```
1.2
```

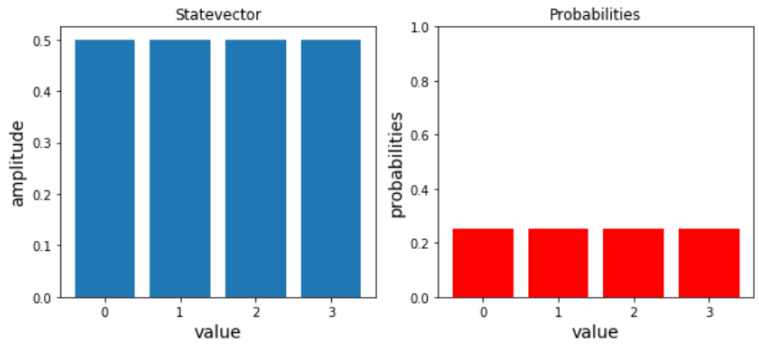
```
A = qs.Qcircuit(2) #2큐빗 회로를 생성함
A.base #초기 양자 상태를 출력함
```

```
array([[1],
       [0],
       [0],
       [0]])
```

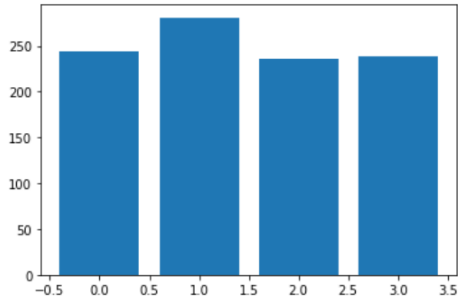
```
A.H(0) #0번 큐비트에 하다마드 게이트를 적용함
A.H(1) #1번 큐비트에 하다마드 게이트로 적용함
A.base #H⊗H|00>의 결과를 보여줌
```

```
array([[0.5],
       [0.5],
       [0.5],
       [0.5]])
```

```
A.graph() #양자 상태를 시각화
```




```
A.M_graph(A.Measure(1000)) #1000번의 측정행위를 했을때의 결과를 시각화
```

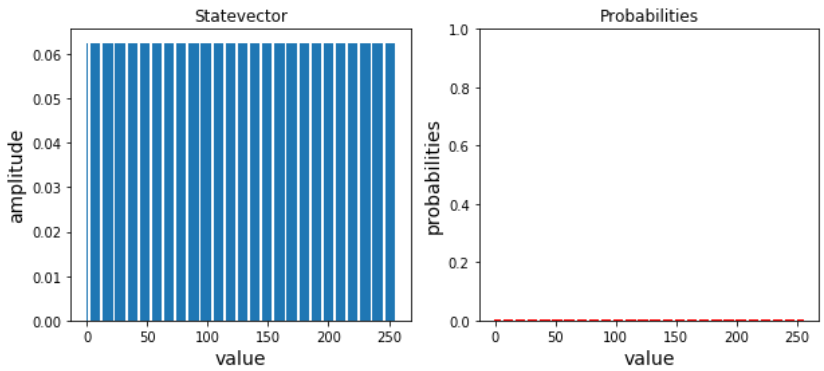


- 기대하는 결과와 동일한 결과를 확인할 수 있었으며 이외에도 다양한 회로를 실험한 결과 정상적으로 작동하는 것을 확인 할 수 있었음

<양자 알고리즘 실험>

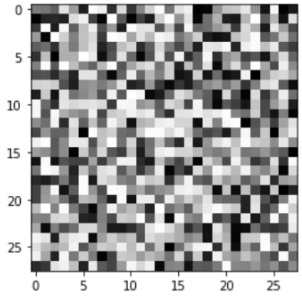
- 양자 난수 생성 알고리즘: 균등 분포에서의 난수를 생성하기 위한 알고리즘. n개의 큐비트를 생성하고 n개의 큐비트에 전부 H게이트 (하다마드 게이트)를 적용시켜 준다. 그리고 측정하게 되면 동일한 확률로 $0 \sim 2^n - 1$까지의 수중 하나를 랜덤으로 뽑아주는 효과를 얻을 수 있다.

```
def Hardarmard_N(state, n):
    for i in range(n):
        state.H(i)
    return state.base
```



8큐비트에 H게이트 적용

```
data = []
for i in range(28):
    for j in range(28):
        result = np.argmax(A.Measure(1))
        data.append(result)
image = np.reshape(data, [28, 28])
plt.imshow(image, cmap='Greys')
plt.show()
```



양자 난수를 생성하여 28x28이미지로 시각화 한 모습

- 그로버 알고리즘: 오라클과 디퓨저라는 함수가 필요. 찾으려는 값에 따라 오라클 회로와 디퓨저 회로가 바뀜. 그래서 회로를 매번 직접 짜는 대신 찾으려는 값에 따라 오라클 함수와 디퓨저 함수에 대한 행렬을 오라클과 디퓨저 함수의 수식에 따라 생성하여 적용하는 방식 사용

오라클 공식: $(I - 2|w><w|)*(|\Psi>)$

공식에 벡터의 외적이 사용되므로 외적을 구하는 함수를 작성함.

```
def outer_product(w):
    result = []
    for i in range(w.shape[0]):
        row = []
        for j in range(w.shape[0]):
            value = w[i][0]*w[j][0]
            row.append(value)
        result.append(row)
    return np.array(result)
```

```
def grover_oracle(state, target):
    base = make_basis(target)
    outer = outer_product(base)
    return state - 2*(outer.dot(state))
```

디퓨저 공식: $(2|\Psi><\Psi| - I)*oracle(|\Psi>)$

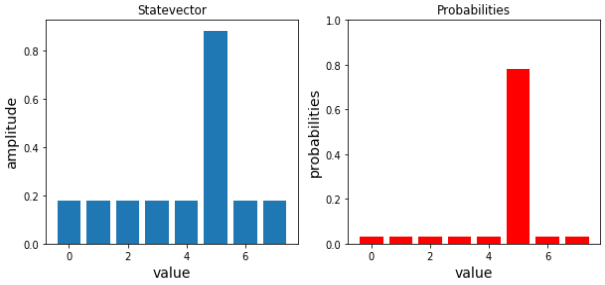
```
def grover_algorithm(state,target):
    oracle = grover_oracle(state, target)
    diffuser = 2*(outer_product(state)).dot(oracle) - oracle
    return diffuser
```

(디퓨저 함수)

실행 결과: 찾고자 하는 값을 2 진법 문자열로 함수에 주면 해당 확률 진폭이 다른 값들에 비해 월등히 높아지는 것을 확인할 수 있고 이는 성공적으로 그로버 알고리즘을 구현했음을 의미함..

```
circuit = qs.Qcircuit(3)
circuit.base = Hardarmard_N(circuit,3)
circuit.base = grover_algoritm(circuit.base,'101')
circuit.graph()
```

5 를 찾는 그로버 알고리즘 실행



5 의 확률 및 진폭이 높게 측정된다.

- 도이치-조사 알고리즘: 상수 함수와 균형함수를 구분해 주는 도이치 알고리즘을 적용할 수 있는 함수를 구현. Oracle 내부에 함수 f(x)가 구현 되어 있으며 f(0)$\oplus$f(1)=1 이면 균형, 0 이면 상수함수이다.

```
def Deutsch_Jozsa(circuit, oracle):
    Hardarmard_N(circuit,circuit.n_resister)
    oracle(circuit)
    Hardarmard_N(circuit,circuit.n_resister-1)
```

- 실험 결과: 개발한 소프트웨어로 양자 알고리즘들이 문제 없이 작동하는 것을 확인함.

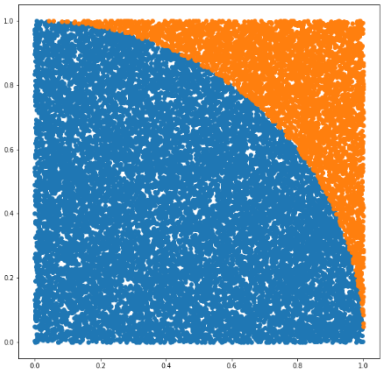
<추가 실험: 생성한 양자난수를 mcm(몬테카를로 매소드)에 사용하여 원주율 구하기>

- 양자 레지스터(큐비트)를 총 10 개인 상태로 진행.
- 모든 큐비트에 모두 H 게이트를 적용시켜 주어 $H^{\otimes 10}|0>^{\otimes 10}$의 양자 중첩 상태 구현.
- 측정을 통해 x 좌표와 y 좌표에 대한 양자 난수를 생성하여 1023 으로 나눈 값을 다시 변수 x 와 y 에 대입. $x^2+y^2<=1$ 일 경우 count 변수의 값을 1 씩 증가
- 4*(count)/n 의 값을 파이로 취하게 하여 n 번 반복

```
n = 20000
count = 0
# 시각화를 위한 리스트들
x_list = [] # 부채꼴 범위 내의 x좌표
y_list = [] # 부채꼴 범위 내의 y좌표
x_list2 = [] # 부채꼴 범위 밖의 x좌표
y_list2 = [] # 부채꼴 범위 밖의 y좌표
for i in range(n):
    x = np.argmax(circuit.Measure(1))/1023
    y = np.argmax(circuit.Measure(1))/1023
    if x**2+y**2<=1:
        count+=1
        x_list.append(x)
        y_list.append(y)
    else:
        x_list2.append(x)
        y_list2.append(y)
    print("진행도: ",i,"현재 파이: ",4*(count)/n)
print(4*(count)/n)
```

```
진행도:  19985 현재 파이:  3.1382
진행도:  19986 현재 파이:  3.1384
진행도:  19987 현재 파이:  3.1386
진행도:  19988 현재 파이:  3.1386
진행도:  19989 현재 파이:  3.1388
진행도:  19990 현재 파이:  3.139
진행도:  19991 현재 파이:  3.1392
진행도:  19992 현재 파이:  3.1394
진행도:  19993 현재 파이:  3.1396
진행도:  19994 현재 파이:  3.1398

진행도:  19995 현재 파이:  3.14
진행도:  19996 현재 파이:  3.1402
진행도:  19997 현재 파이:  3.1404
진행도:  19998 현재 파이:  3.1406
진행도:  19999 현재 파이:  3.1408
3.1408
```



(양자 난수를 이용한 몬테카를로 매소드를 시각화 한 결과)

- 결과: 균등한 확률로 0~1023까지의 난수를 지속적으로 생성하면서 x, y 좌표를 갱신해주며 시뮬레이션 한 결과 원주율 3.14와 근접한 값을 도출해 내는데 성공함.

<한계 테스트>

- 개발한 소프트웨어를 바탕으로 실험한 결과 고전컴퓨터에서도 양자컴퓨팅을 시뮬레이션 할 수 있다는 것을 알아냄. 동시에 고전 컴퓨터에서 단순히 양자컴퓨터의 수학적 계산 과정을 그대로 구현 할 시 한계가 존재 할지 알아보기로 함.
- 0번째 큐비트에 H게이트를 적용시켜 준다고 할 때 최대 몇 개의 큐비트까지 생성했을 때 안정적으로 시뮬레이션이 가능할지 테스트 해보기로 함.

```
import Quself as qs
from timeit import default_timer as timer
from datetime import timedelta
qs.version
```

```
1.2
```

```
def circuit_init_time(n):
    start = timer()
    A = qs.Qcircuit(n)
    end = timer()
    print("회로 초기화: ",timedelta(seconds=end-start),end=" ")
    start = timer()
    A.H(0)
    end = timer()
    print("   H게이트 적용 시간: ",timedelta(seconds=end-start))
```

```
for i in range(21):
    print("큐비트 %2d개 "%i,end="")
    circuit_init_time(i)
```

```
큐비트  0개 회로 초기화:  0:00:00.000020    H게이트 적용 시간:  0:00:00.000015
큐비트  1개 회로 초기화:  0:00:00.000098    H게이트 적용 시간:  0:00:00.000186
큐비트  2개 회로 초기화:  0:00:00.000165    H게이트 적용 시간:  0:00:00.000152
큐비트  3개 회로 초기화:  0:00:00.000253    H게이트 적용 시간:  0:00:00.000318
큐비트  4개 회로 초기화:  0:00:00.000653    H게이트 적용 시간:  0:00:00.000390
큐비트  5개 회로 초기화:  0:00:00.000363    H게이트 적용 시간:  0:00:00.001630
큐비트  6개 회로 초기화:  0:00:00.000411    H게이트 적용 시간:  0:00:00.000594
큐비트  7개 회로 초기화:  0:00:00.000556    H게이트 적용 시간:  0:00:00.006712
큐비트  8개 회로 초기화:  0:00:00.000736    H게이트 적용 시간:  0:00:00.003587
큐비트  9개 회로 초기화:  0:00:00.001333    H게이트 적용 시간:  0:00:00.009732
```

```
큐비트 10개 회로 초기화:  0:00:00.002088    H게이트 적용 시간:  0:00:00.037555
큐비트 11개 회로 초기화:  0:00:00.002191    H게이트 적용 시간:  0:00:00.173484
큐비트 12개 회로 초기화:  0:00:00.006300    H게이트 적용 시간:  0:00:00.835796
큐비트 13개 회로 초기화:  0:00:00.017805    H게이트 적용 시간:  0:00:03.218478
큐비트 14개 회로 초기화:  0:00:00.034797    H게이트 적용 시간:  0:00:12.348964
큐비트 15개 회로 초기화:  0:00:00.039472    H게이트 적용 시간:  0:02:19.056912
큐비트 16개 회로 초기화:  0:00:00.136032
```

- 결과: 13 큐비트부터 속도 저하가 발생하더니 15 큐비트에 하다마드 게이트 하나를 적용시키는데 2분 이상 소요됨. 16큐비트부터는 메모리 오류가 발생함.
- 결론: 고전 컴퓨터에서 단순히 양자컴퓨터의 수학적 계산 과정을 그대로 구현 할 시 큐비트가 늘어날 수록 계산 량이 지수배로 늘어나 속도가 느려진다는 한계를 알게 됨.

<결론 및 반성>

- 고전 컴퓨터에서도 수학적인 개념을 바탕으로 양자 컴퓨팅을 시뮬레이션 할 수 있다.
- 일반 pc에서 평균적으로 12큐비트까지의 양자컴퓨팅 시뮬레이션이 가능하다
- 큐비트 수가 늘어날수록 계산 량이 증가해 속도 및 메모리 문제가 발생한다.
- 한계 테스트 전에는 어떠한 경우에도 고전 컴퓨터에서도 양자컴퓨팅을 시뮬레이션 할 수 있다고 성급한 판단을 내렸다. 한계를 실험하면서 문제점을 발견했고 너무 성급하게 결론을 냈다는 생각을 하게 되었으며 SW 제작과 실험을 하면서 여러 변수들에 대해 생각해볼 필요성과 이를 간과할 시 소프트웨어와 실험 결과에 치명적인 오류를 만든다는 점을 알게 되었다.

<전체 소스 코드 및 사용 예시>

*소스 코드 뒤에 해당 소프트웨어의 사용예시를 확인할 수 있습니다.

Quself.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
version = 1.2
help =
("I","H","X","Y","Z","S","T","SQRT_X","U","RX","RY","RZ","CNOT","CY","CZ","CS","SWAP","TOFFOLI")
class Qcircuit:
    # outer result
    outer_zero = np.array([[1,0],[0,0]]) # |0><0|
    outer_one = np.array([[0,0],[0,1]]) # |1><1|
    # single Quantum Gates
```

```
    I_gate = np.array([[1,0],[0,1]])
    H_gate = (1/math.sqrt(2))*np.array([[1,1],[1,-1]])
    X_gate = np.array([[0,1],[1,0]])
    Y_gate = np.array([[0,-1j],[1j,0]])
    Z_gate = np.array([[1,0],[0,-1]])
    S_gate = np.array([[1,0],[0,1j]])
    T_gate = np.array([[1,0],[0,(1+1j)/math.sqrt(2)]])
    SqrtX_gate = (1/math.sqrt(2))*np.array([[1,-1],[1,1]])
    # Class Initalize
    def __init__(self, n):
        self.n_resister = n
        self.one_qubit_zero = np.array([[1],[0]]) # |0>
        self.one_qubit_one = np.array([[0],[1]]) # |1>
        self.base = np.array([[1]])
        for i in range(n):
            self.base = np.kron(self.base,self.one_qubit_zero)
    # Function Making a Uf Matrix
    def make_uf_matrix(self,n,gate):
        basis = np.array([[1]])
        for i in range(self.n_resister):
            if i == n:
                basis = np.kron(basis,gate)
            else:
                basis = np.kron(basis,self.I_gate)
        return basis
    # Make n qubit gate
    def Two_qubit_gate_to_Nresister(self, U, control, target):
        basis1 = np.array([[1]])
        basis2 = np.array([[1]])
        if control == target:
            raise ValueError("control and target is same!!")
        if self.n_resister < 2:
            raise ValueError("This gate requires least 2 qubits but your circuit is just one
or zero qubits")
        if control >= self.n_resister or target >= self.n_resister:
            raise ValueError("control and target should be small than N")
        for i in range(self.n_resister):
            if i == control:
                basis1 = np.kron(basis1,self.outer_zero)
            else:
                basis1 = np.kron(basis1,self.I_gate)
        for i in range(self.n_resister):
            if i == control:
                basis2 = np.kron(basis2,self.outer_one)
            elif i == target:
                basis2 = np.kron(basis2,U)
            else:
                basis2 = np.kron(basis2,self.I_gate)
        return basis1+basis2
    # Apply one qubit gate to the circuit
    def I(self,n):
        basis = self.make_uf_matrix(n,self.I_gate)
        self.base = basis.dot(self.base)
    def H(self,n):
        basis = self.make_uf_matrix(n,self.H_gate)
        self.base = basis.dot(self.base)
    def X(self,n):
        basis = self.make_uf_matrix(n,self.X_gate)
```

```
        self.base = basis.dot(self.base)
    def Y(self,n):
        basis = self.make_uf_matrix(n,self.Y_gate)
        self.base = basis.dot(self.base)
    def Z(self,n):
        basis = self.make_uf_matrix(n,self.Z_gate)
        self.base = basis.dot(self.base)
    def S(self,n):
        basis = self.make_uf_matrix(n,self.S_gate)
        self.base = basis.dot(self.base)
    def T(self,n):
        basis = self.make_uf_matrix(n,self.T_gate)
        self.base = basis.dot(self.base)
    def SqrtX(self,n):
        basis = self.make_uf_matrix(n,self.SqrtX_gate)
        self.base = basis.dot(self.base)
    def U(self,n,theta,phi,lamda):
        U_gate = np.array([[math.cos(theta/2),-
math.e**(1j*lamda)*math.sin(theta/2)],[math.e**(1j*phi)*math.sin(theta/2),math.e**(1j*(phi+
lamda))*math.cos(theta/2)]])
        basis = self.make_uf_matrix(n,U_gate)
        self.base = basis.dot(self.base)
    def RX(self,n,theta):
        self.U(n,theta,-(np.pi/2),np.pi/2)
    def RY(self,n,theta):
        self.U(n,theta,0,0)
    def RZ(self,n,phi):
        self.U(n,0,0,phi)
    # Apply two qubit gate to the circuit
    def CNOT(self,control,target):
        basis = self.Two_qubit_gate_to_Nresister(U = self.X_gate,control = control, target
= target)
        self.base = basis.dot(self.base)
    def CY(self,control,target):
        basis = self.Two_qubit_gate_to_Nresister(U = self.Y_gate,control = control, target
= target)
        self.base = basis.dot(self.base)
    def CZ(self,control,target):
        basis = self.Two_qubit_gate_to_Nresister(U = self.Z_gate,control = control, target
= target)
        self.base = basis.dot(self.base)
    def CS(self,control,target):
        basis = self.Two_qubit_gate_to_Nresister(U = self.S_gate,control = control, target
= target)
        self.base = basis.dot(self.base)
    def SWAP(self,target1,target2):
        if(target1==target2):
            raise ValueError("targets are same")
        if(target1>target2):
            target1,target2 = target2,target1
        self.CNOT(target1,target2)
        self.CNOT(target2,target1)
        self.CNOT(target1,target2)
    # Apply three qubit gate to the circuit
    def TOFFOLI(self,control1,control2,target):
        if((control1==control2 and control1 ==
target)or(control1==target)or(control1==control2)or(control2==target)):
            raise ValueError("given values are not collect")
```

```
        control1 += 1
        control2 += 1
        target += 1
        basis = np.array([([0] * 2**self.n_resister)] * 2**self.n_resister)
        M = 2**(self.n_resister-control1)+2**(self.n_resister-control2)
        d = lambda i : 2**(self.n_resister-target) if (2**(self.n_resister-target)) & (i)
== 0 else -(2**(self.n_resister-target))
        for i in range(2**self.n_resister) :
            for j in range(2**self.n_resister) :
                basis[i][j] = 1 if (i & M == M and j == d(i) + i) or (i & M != M and j == i)
else 0
        self.base = basis.dot(self.base)
    # Visualization the statevector and Probabilities
    def graph(self):
        fig = plt.figure(figsize=(10,4))
        statevector = []
        probabilities = []
        for i in range(len(self.base)):
            statevector.append(self.base[i][0])
            probabilities.append(abs(self.base[i][0])**2) #확률진폭의 제곱
        x = np.arange(2**self.n_resister)
        ax = fig.add_subplot(1,2,1)
        ax2 = fig.add_subplot(1,2,2)
        ax.bar(x,statevector)
        ax2.bar(x,probabilities,color = 'r')
        ax.set_xlabel("value",size=14)
        ax2.set_xlabel("value",size=14)
        ax.set_ylabel("amplitude",size=14)
        ax2.set_ylabel("probabilities",size=14)
        ax.set_title("Statevector")
        ax2.set_title("Probabilities")
        ax2.set_ylim([0, 1])
        plt.show()
    # Measure all qubits (simulation)
    def Measure(self,shot=1):
        probabilities = []
        result = [0]*2**self.n_resister
        for i in range(len(self.base)):
            probabilities.append(abs(self.base[i][0])**2)
        x = np.arange(2**self.n_resister)
        for i in range(shot):
            b = np.random.choice(x, 1, replace=False, p=probabilities)
            result[int(b)] += 1
        return result
    def M_graph(self,result):
        x = np.arange(2**self.n_resister)
        y = result
        plt.bar(x,y)
        plt.show()
    # Print Circuit Value
    def __str__(self):
        output = ""
        for i in range(2**self.n_resister):
            output+="{0}|{1}>".format(complex(self.base[i]),i)
            if i != 2**self.n_resister-1:
                output+="+"
        return output
    # delete instance
```

```
    def __del__(self):
        pass
```

quantum_algorithm.py

```
import Quself as qs
import numpy as np
def Hardarmard_N(state, n):
    for i in range(n):
        state.H(i)
    return state.base
def make_basis(target):
    one_qubit_zero = np.array([[1],[0]]) #|0>
    one_qubit_one = np.array([[0],[1]]) #|1>
    base = np.array([[1]])
    for i in range(len(target)):
        if target[i]=='0':
            base = np.kron(base,one_qubit_zero)
        elif target[i]=='1':
            base = np.kron(base,one_qubit_one)
        else:
            raise ValueError("target 입력이 잘못되었습니다.")
    return base
def outer_product(w):
    result = []
    for i in range(w.shape[0]):
        row = []
        for j in range(w.shape[0]):
            value = w[i][0]*w[j][0]
            row.append(value)
        result.append(row)
    return np.array(result)
#grover algorithm-------------------------------------------------------------------------
-------------------------
def grover_oracle(state, target):
    base = make_basis(target)
    outer = outer_product(base)
    return state - 2*(outer.dot(state))  # (I-2|target><target|)*(|state>)
def grover_algorithm(state,target):
    oracle = grover_oracle(state, target)
    diffuser = 2*(outer_product(state)).dot(oracle) - oracle # (2|state><state| -
I)*oracle(|state>)
    return diffuser
#-------------------------------------------------------------------------------------------
-------------------------
#Deutsch-Jozsa algorithm-------------------------------------------------------------------
-------------------------
def Deutsch_Jozsa(circuit, oracle):
    Hardarmard_N(circuit,circuit.n_resister)
    oracle(circuit)
    Hardarmard_N(circuit,circuit.n_resister-1)
#-------------------------------------------------------------------------------------------
-------------------------
```

Quself 사용 예시

In [8]:
```
import Quself as qs # Quself 불러오기
import quantum_algorithm as qa # Quself 기반 양자 알고리즘 라이브러리 불러오기
qs.version
```

Out[8]: 1.2

Quself가 지원하는 양자 게이트

In [9]:
```
qs.help
```

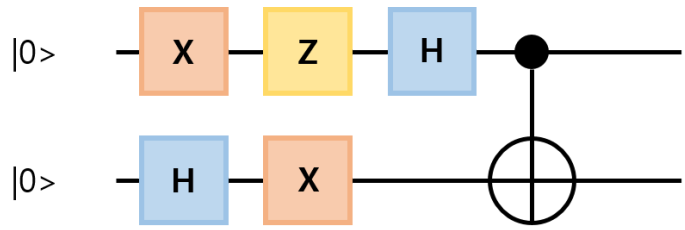
Out[9]:
```
('I',
 'H',
 'X',
 'Y',
 'Z',
 'S',
 'T',
 'SQRT_X',
 'U',
 'RX',
 'RY',
 'RZ',
 'CNOT',
 'CY',
 'CZ',
 'CS',
 'SWAP',
 'TOFFOLI')
```

양자 회로 초기화 하기

In [10]:
```
A = qs.Qcircuit(2) # qs.Qcircuit(큐비트 수)
A.base # 현재 양자상태 출력
```

Out[10]:
```
array([[1],
       [0],
       [0],
       [0]])
```

양자 게이트 적용하기



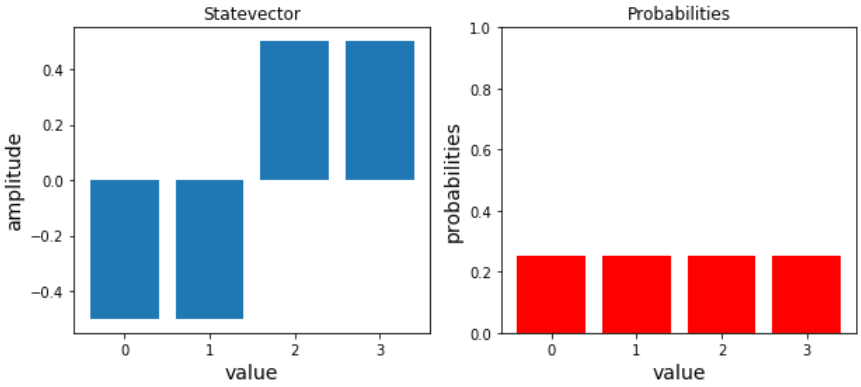
* 위 회로를 Quself로 구현하면 다음과 같다

In [11]:
```
A.X(0) # qs.X(적용하려는 큐비트 번호)
A.H(1) # qs.H(적용하려는 큐비트 번호)
A.Z(0) # qs.Z(적용하려는 큐비트 번호)
A.X(1)
A.H(0)
A.CNOT(0,1) # qs.CNOT(컨트롤 큐비트 번호, 타켓 큐비트 번호)
A.base # 회로 적용 후 양자 상태 출력
```

Out[11]:
```
array([[-0.5],
       [-0.5],
       [ 0.5],
       [ 0.5]])
```

양자상태 시각화

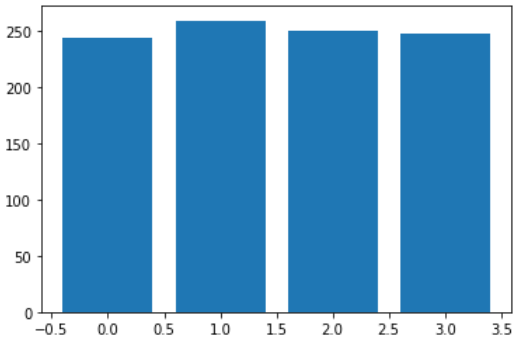
In [13]:
```
A.graph() # 현재 양자상태의 확률진폭과 확률을 보여준다
```



측정 및 측정 결과 시각화 하기

In [16]:
```
measure_once = np.argmax(A.Measure(1)) # qs.Measure(측정 횟수) = 측정해서 나온 결과값의 인덱스를 1증가시켜 리스트를 만들어 준다
print("한번 측정해서 얻은 값: ",measure_once)
A.M_graph(A.Measure(1000)) # 1000번 측정해서 그 결과를 시각화 해준다
```

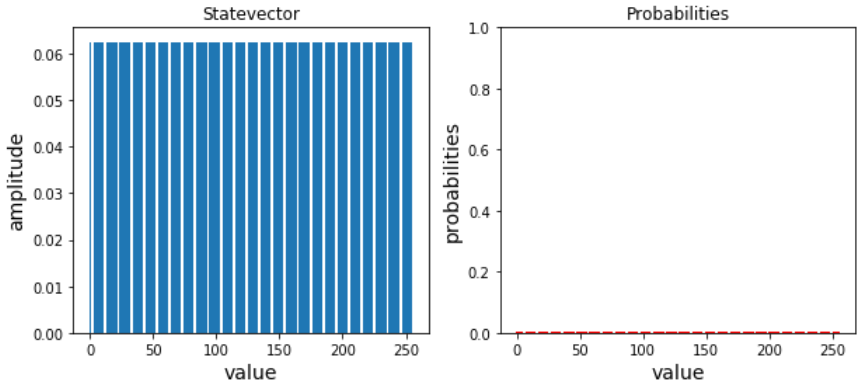
```
한번 측정해서 얻은 값:  2
```

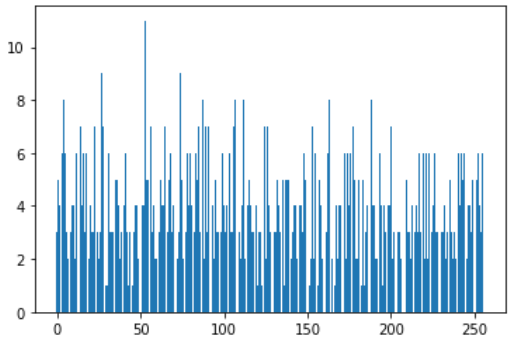


양자 알고리즘 적용 예시

양자 난수 생성

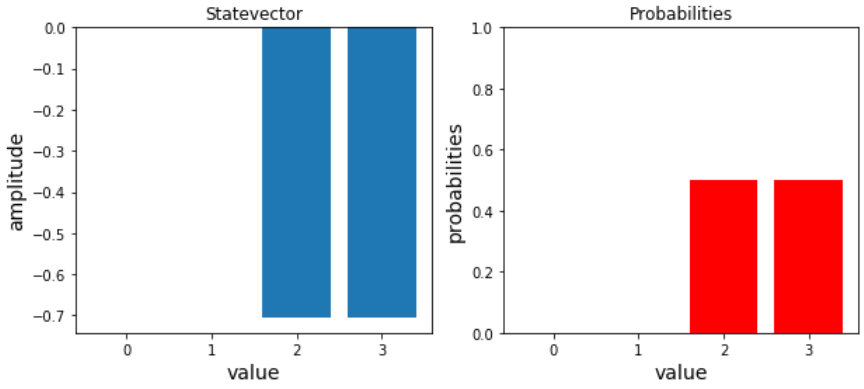
In [31]:
```
A = qs.Qcircuit(8) # 8큐비트 양자 회로 생성
qa.Hardarmard_N(A,A.n_resister) # Hardarmard_N(양자 회로, H게이트 적용 개수)
A.graph() # 확률 진폭 및 확률 시각화
A.M_graph(A.Measure(1000)) # 1000번의 측정 결과 시각화
```





도이치-조사 알고리즘

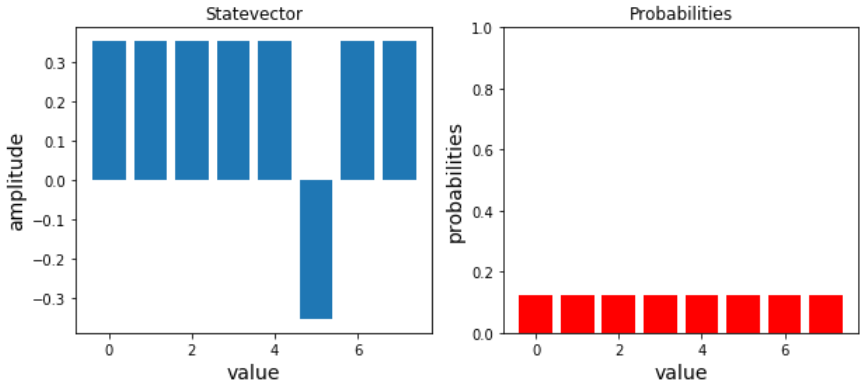
In [34]:
```
# 오라클 함수 생성
A = qs.Qcircuit(2)
def oracle(circuit):
    circuit.X(1)
    circuit.Z(0)
    circuit.CNOT(1,0)
# 도이치 조사 알고리즘 적용
A.X(1)
qa.Deutsch_Jozsa(A,oracle)
A.graph()
```



- 2와 3 즉 "10"과 "11"이 나왔다. 0번째 큐비트가 1로 측정 되었음으로 즉 f(0) ⊕ f(1)=1이므로 오라클 함수는 균형함수이다.

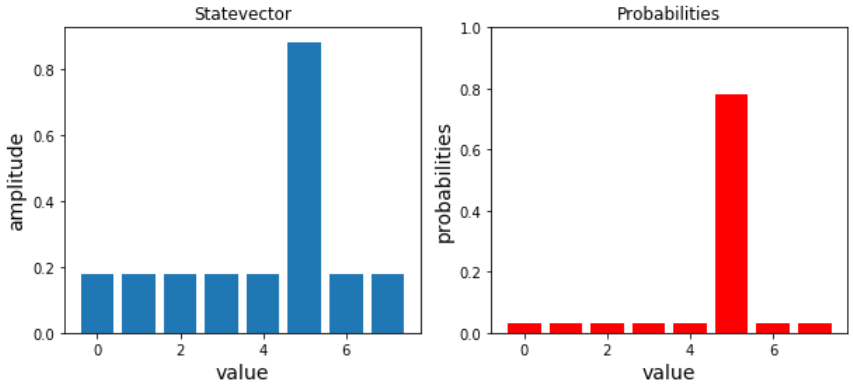
그로버 알고리즘

In [36]:
```
A = qs.Qcircuit(3) # 3큐비트 양자 회로 생성
qa.Hardarmard_N(A,A.n_resister) # 모든 큐비트에 H게이트 적용
A.base = qa.grover_oracle(A.base,'101') # 5를 찾는 오라클 함수 적용
A.graph()
```



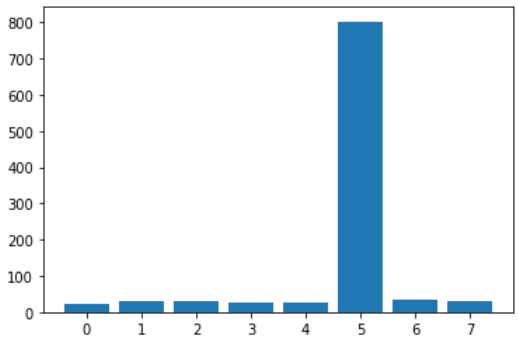
- 그로버 알고리즘의 2번째 단계로써 찾고자 하는 값의 확률진폭이 성공적으로 음수로 바뀐것을 확인할 수 있다

In [54]:
```
A = qs.Qcircuit(3) # 3큐비트 양자 회로 생성
qa.Hardarmard_N(A,A.n_resister) # 모든 큐비트에 H게이트 적용
A.base = qa.grover_algorithm(A.base,'101') # 오라클 함수와 디퓨저 함수를 동시에 적용할 수 있는 함수
A.graph()
```



- 찾고자 하는 값인 5의 확률진폭이 다른 값들에 비해 눈에 띄게 높아졌다.

In [55]:
```
A.M_graph(A.Measure(1000))
```



- 1000번의 측정 결과