Grover's Algorithm Implementation and Success Probability Dependence On Grover's Iterates

Using Python & Qiskit

Mohamed Sami ShehabEldin

s-mohamed.sami@zewailcity.edu.eg

201700936

Zewail City of Science and Technology

Outline

Part 1: General Grover's Algorithm

Grover's Algorithm is a quantum search algorithm that can speed up classical search algorithms quadratically. This can be implemented on 3 steps Initialization, Amplitude Amplification: Question some elements (winners), Amplitude Amplification: Boosting winners amplitudes.

1.1 Initialization

Searching problems can stated as "Given an unstructured list, find if certain element exists and find its location"

In Grover's Algorithm we deal the unsorted list as a linear compination of its elements. That is given list {2,5,0,6,1,3,7,4}, we will feed it to our circuit as:

$$|S>=\frac{1}{\sqrt{8}}(|0>+|1>+|2>+|3>+|4>+|5>+|6>+|7>)$$

these elements can be the label of computational basis, and in general, for N elements list, it can be dealt as:

$$|S>=\frac{1}{\sqrt{N}}(|00...00>+|00...01>+|00...10>+...+|11...11>)$$

Or more compactly, we start our list as:

$$|S>\equiv H^{\otimes n}|0>^{\otimes n} \qquad ;where \quad n=log_2(N)$$

This is easy to impelemet with Qiskit. I will do it as a function for any general number of Qubits n (any list N), and give example for n=3 (list of 8 elements).

In [47]:

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

# importing Qiskit
from qiskit import IBMQ, Aer, QuantumCircuit, ClassicalRegister, QuantumRegi
from qiskit.providers.ibmq import least_busy

# import basic plot tools
from qiskit.visualization import plot_histogram
```

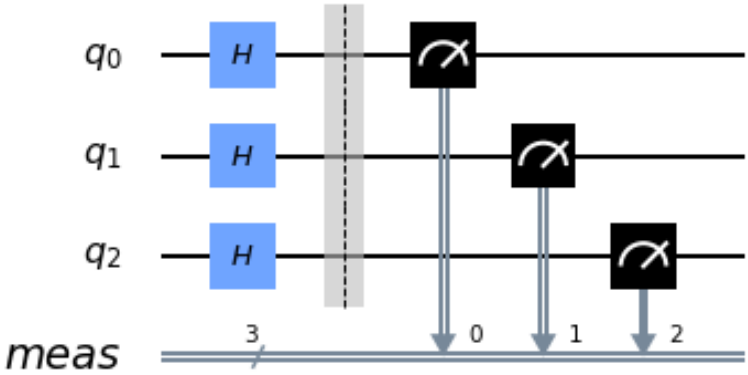
In [90]:

```
def grover_initialize(nqubits=3):
    '''
    :Param nqubits: the number of qubits or log2(lenth of the list)
    :return: quantum circuit with all 2^n basis in uniform linear compinatic
    '''
    qc=QuantumCircuit(nqubits)
    qc.h(range(nqubits))
    return qc
```

In [91]:

```
#lets see how ot looks for 3 qubits or 8 elements list
qc=grover_initialize(3)
qc.measure_all()
qc.draw('mpl')
```

Out[91]:



we can also see the result of measurement on simulator and real device. I will make them in function as I use them a lot.

In [92]:

```
def simulator_result(qc,shots=1024):
    qasm_simulator = Aer.get_backend('qasm_simulator')
    results = execute(qc, backend=qasm_simulator, shots=shots).result()
    answer = results.get_counts()
    return answer
```

In [93]:

```
def device_result(qc, shots=1024, optimization_level=3):
    provider = IBMQ.load_account()
    device = least_busy(provider.backends(filters=lambda x: x.configuration(
                                    not x.configuration().simulator and x.sta
    print("Running on current least busy device: ", device)
    # Run our circuit on the least busy backend. Monitor the execution of th
    from qiskit.tools.monitor import job_monitor
    job = execute(qc, backend=device, shots=shots, optimization_level=optimi
    job_monitor(job, interval = 2)
    # Get the results from the computation
    results = job.result()
    answer = results.get_counts(grover_circuit)
    plot_histogram(answer)
    return answer
```

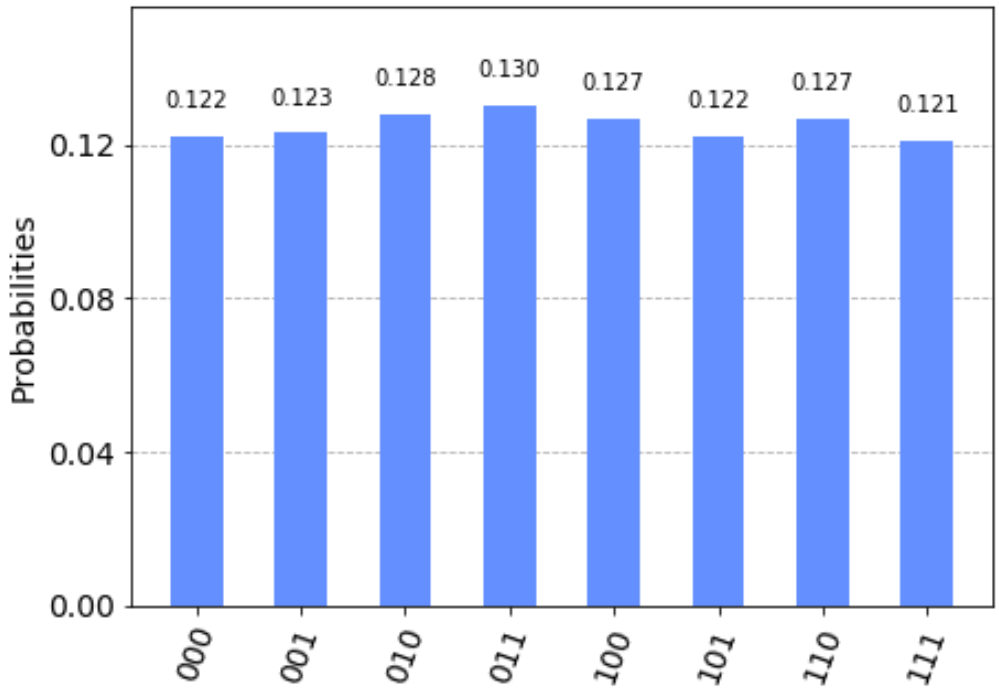
In [118]:

```
def vector_simulator(qc):
    sv_sim = Aer.get_backend('statevector_simulator')
    job_sim = execute(qc, sv_sim)
    statevec = job_sim.result().get_statevector()
    from qiskit_textbook.tools import vector2latex
    vector2latex(statevec, pretext="|\\psi\\rangle =")
```

In [94]:

```
#see for our n=3 circuit
res=simulator_result(qc)
plot_histogram(res)
```

Out[94]:



this is our element's list. You can add your feed in your own list or you can give computational basis different labels.

1.2 Amplitude Amplification: Question some elements (winners)

Given this list of elements, we want to ask if some element call it **winner** $|\omega>$ exist in this list or not? you can ask if number 6 exist? or if number 66? and so on. we embed these questions in what is called **"Oracle"** or we call it U_f.

we design this oracle such that it gives negative sign to the element of the question. I.E. if we ask about $|\omega>$, then

$$U_f|x=\omega>=-1 \quad and \quad U_f|x\neq\omega>=-1$$

and this corresponds for a diagonal matrix of negative one on the element $|\omega><\omega|$ and positive one the rest of the diagonal.

$$U_f = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & -1 & \vdots & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

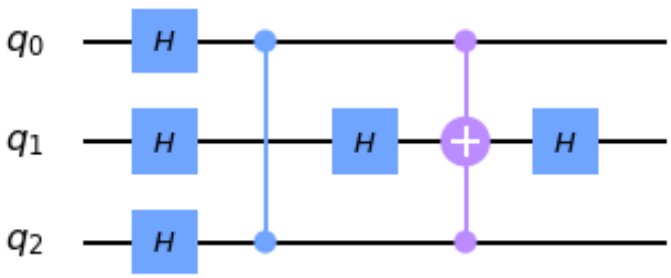
we can find the equivlent set of gates that acheive this unitary operation by optimiztion algorithms, or we can find it manually for each case.

in case of searching for number the $6^{th}\ element = |5> = |101>$ in 8 element's data, we can negative this specific state by CZ(0,2). Because you need both last qubit to be |1> to act on first qubit by Z, and Z only negative the |1> state. But this will also Negative the state |111>, we can avoid this by acting with multi qubit controlled Z gate, such that it act with Z on the middle qubit onle if the first and last qubits are in |1>.

In [180]:

```
qc=grover_initialize(3)
qc.cz(0,2)
#multi qubit z gate
qc.h(1)
qc.mct([0,2],1)
qc.h(1)
qc.draw("mpl")
```

Out[180]:



In [181]:

```
#verify that it negative the sixth element
vector_simulator(qc)
```

$$|\psi\rangle = \begin{bmatrix} 0.35355 \\ 0.35355 \\ 0.35355 \\ 0.35355 \\ 0.35355 \\ -0.35355 \\ 0.35355 \\ 0.35355 \end{bmatrix}$$

In [188]:

```
#lets make it into function, just in case we use it
def oracle_5():
    '''
    :return: quantum circuit that act -ve on |101>
    '''
    oracle_gate=QuantumCircuit(3)
    oracle_gate.cz(0,2)
    #multi qubit z gate
    oracle_gate.h(1)
    oracle_gate.mct([0,2],1)
    oracle_gate.h(1)

    oracle_gate.name="Oracle 6"
    return oracle_gate
```

for any other element in any list, all we need is to apply the equivlent gates of the diagonal matrix U_f. for this purpose I will use function "Diagonal".

In [116]:

```
from qiskit.circuit.library import Diagonal
```

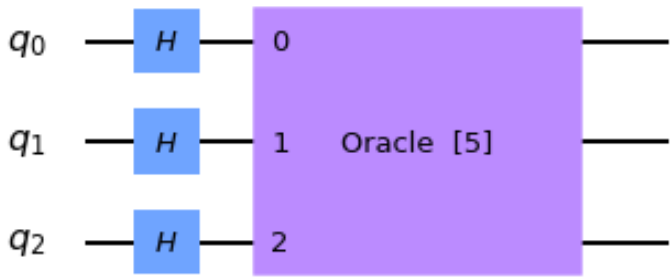
In [157]:

```
def oracle(nqubits=3, winners=[5]):
    '''
    :Param winner: the certain elements we would like to search
    :Param nqubits: the number of qubits which we susbect that the winner be
    :return: equivlent gates "oracle" that act on |winners> to be -|winners>
    '''
    diagonal_elements=np.ones(2**nqubits)
    diagonal_elements[winners]=-1
    oracle_gate = Diagonal(diagonal_elements)
    oracle_gate.name = "Oracle  %s" %winners
    return oracle_gate
```

In [186]:

```
qc=grover_initialize(3)
qc.append(oracle(3,[5]),range(3))
qc.draw("mpl")
```

Out[186]:



In [187]:

```
#verify that it negative the sixth element
vector_simulator(qc)
```

$$|\psi\rangle = \begin{bmatrix} 0.35355 \\ 0.35355 \\ 0.35355 \\ 0.35355 \\ 0.35355 \\ -0.35355 \\ 0.35355 \\ 0.35355 \end{bmatrix}$$

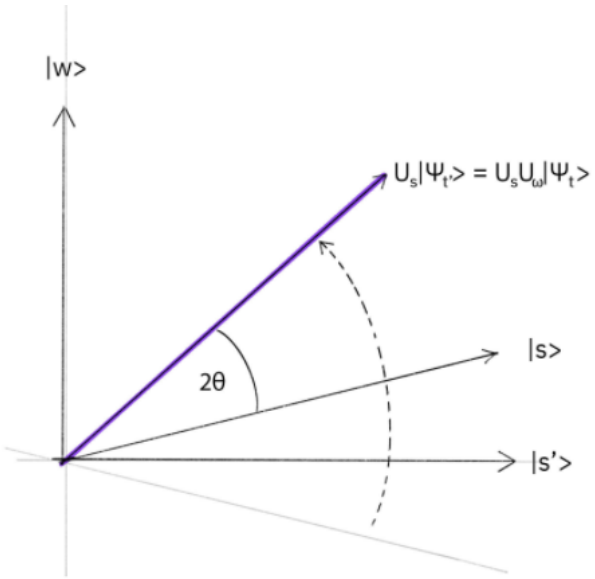
This part is the only part that depends on what you are searching for. The next step (which is general for any element) is to see if this element exist or not, because in this current state you cannot differ it by just the sign flip.

1.3. Amplitude Amplification: Boosting winners amplitudes

Given the the last state whose winner has a negative amplitude, in this step, we will try to get is amplitude positive again but higher than its peers. So that we can discriminate it by measurements. We do this job bu what is called **"Diffuser Operator"** U_s.

This action of U_s can be understood geometrically. Initially, we start with $|S>\equiv H^{\otimes n}|0>^{\otimes n}$ (or any list you want).

By acting by U_f we reflect the $|S>$ around $|\omega>$, we will design U_s such that it reflect the new state $(U_f|S>)$ around the original |S>. This precess will cause $|\omega>$ to gain more weight as in the image:



this reflection about the original elements linear combination |S> can be written as:

$$U_s = 2|S><S| - I$$

Which means that we negative any state that is orthogonal to |S>.

For the case that the list composed of all computational basis: $|S>= H^{\otimes n}|0>^{\otimes n}$, we can figure U_s easily. We want to negative any state that is orthogonal to |S>, but $H^{\otimes n}|S>= |0>^{\otimes n}$. Hence, we will act on $(U_f|S>)$ by $H^{\otimes n}$ and then negative |000..00> and then act again by by $H^{\otimes n}$. (for a global negative, this is the same as negativing whatever not |000..00>)

$$U_s = H^{\otimes n}U_0H^{\otimes n} = H^{\otimes n}X^{\otimes n}(MCZ)X^{\otimes n}H^{\otimes n}$$

Where (MCZ) is Multi-Controlled-Z gate, that act with Z on the last qubit when all other qubits are |1>.

this can be easily implemented by Qiskit, and I copied the following function directly from Qiskit Textbook.

In [328]:

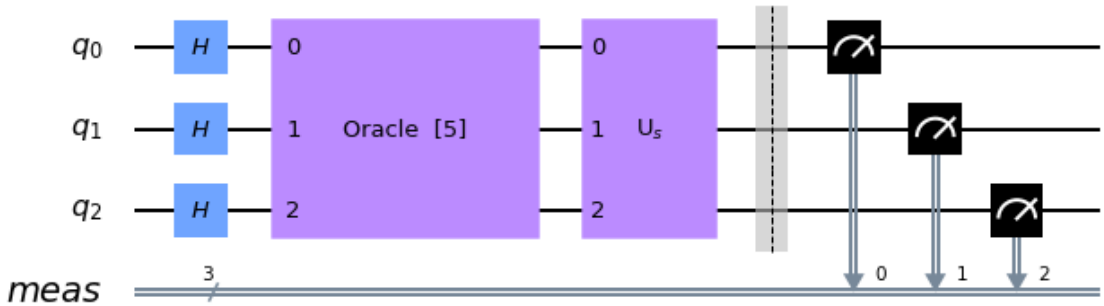
```
def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(qubit)
    # Apply transformation |00..0> -> |11..1> (X-gates)
    for qubit in range(nqubits):
        qc.x(qubit)
    # Do multi-controlled-Z gate
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)  # multi-controlled-toffoli
    qc.h(nqubits-1)
    # Apply transformation |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)
    # Apply transformation |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)
    # We will return the diffuser as a gate
    U_s = qc.to_gate()
    U_s.name = "U$_s$"
    return U_s
```

Lets see what this looks like in our example

In [329]:

```
qc=grover_initialize(3)
qc.append(oracle(3,[5]),range(3))
qc.append(diffuser(3),range(3))
qc.measure_all()
qc.draw("mpl")
```

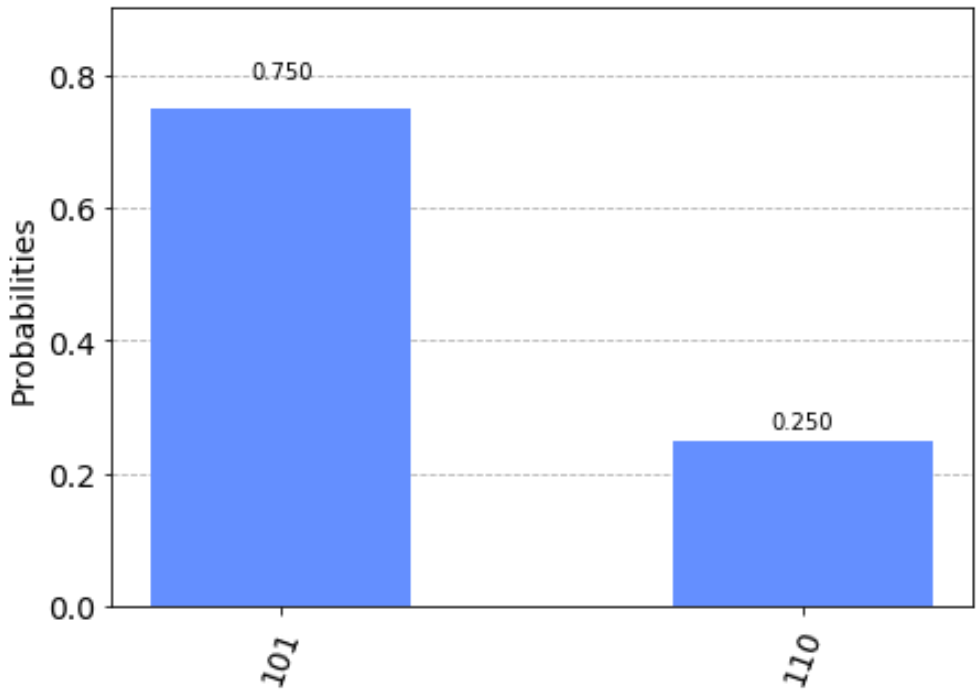
Out[329]:



In [330]:

```
res=simulator_result(qc,shots=4)
plot_histogram(res)
```

Out[330]:



We see that the probability peaks at the Sixth element |101>. I used 4 shots as it is the square root of the required classical steps, this seems not enough! But we/I promised to find the search result in quadratically less iteration. But it seems that this will fail for 4 shots!

We can increase the weight again by the same process till we reach sufficient accuracy.

$$(U_f U_s)^t |S>$$

where **t ≡ Grover's Iterates.**

It turns out that the optimized iterations is $t \approx \frac{\pi}{4}\sqrt{\frac{N}{M}}$; where N is the number of elements in the list, and M is the number of winners. This is motivated On page 157 of Kaye, Laflamme and Mosca in "An introduction to quantum computing". I will show this computationally in the next section.

For this purpose I am doing this simple function:

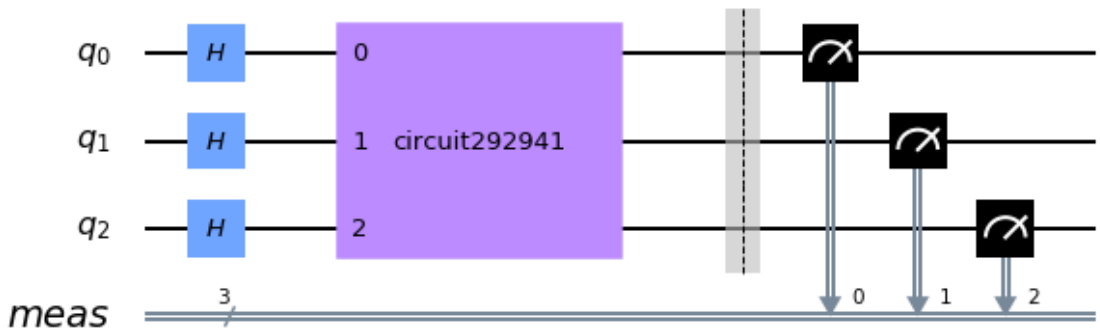
In [687]:

```
def amp_lify(nqubits=3, winners=[5], iterations=0):
    if iterations==0:
        iterations=int(np.floor(((np.pi)/4)*np.sqrt(2**nqubits/len(winners))))

    amplification_circ=QuantumCircuit(nqubits)

    for i in range(iterations):
        amplification_circ.append(oracle(nqubits, winners),range(nqubits))
        amplification_circ.append(diffuser(nqubits),range(nqubits))

    return amplification_circ.decompose()
```

In [680]:

```
n=3
w=5
qc=grover_initialize(n)
qc.append(amp_lify(n,[w],0),range(n))
qc.measure_all()
qc.draw("mpl")
```

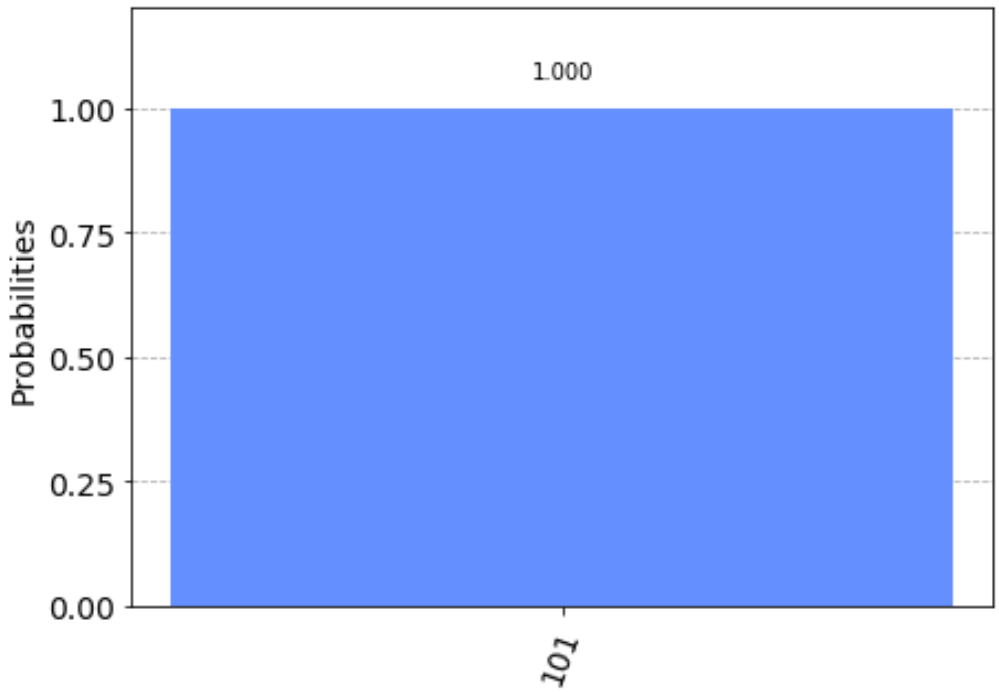
Out[680]:



In [333]:

```
res=simulator_result(qc,shots=4)
plot_histogram(res)
```

Out[333]:

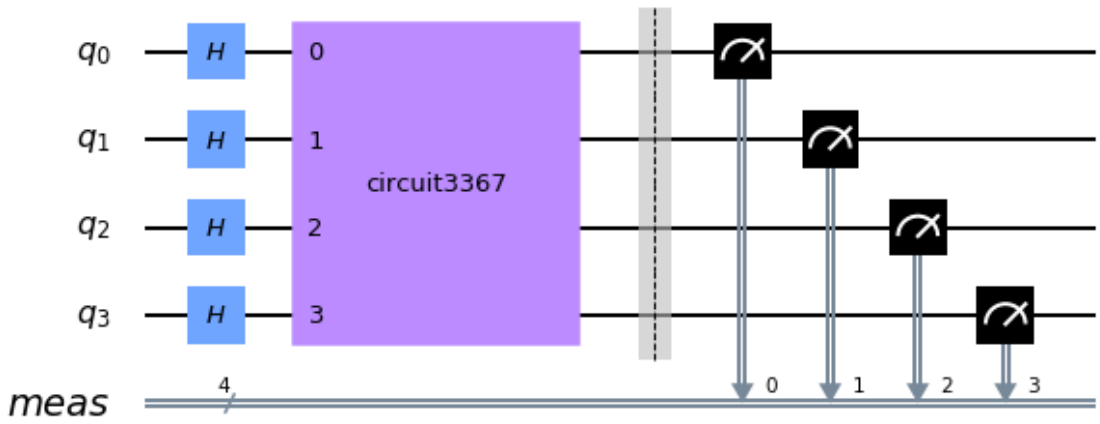


This works for any generic winner element, but it the list should be the whole computational basis of your space. Here is another example, searching for 5 in list of 16 elements.

In [334]:

```
n=4
w=5
qc=grover_initialize(n)
qc.append(amp_lify(n,[w],0),range(n))
qc.measure_all()
qc.draw("mpl")
```

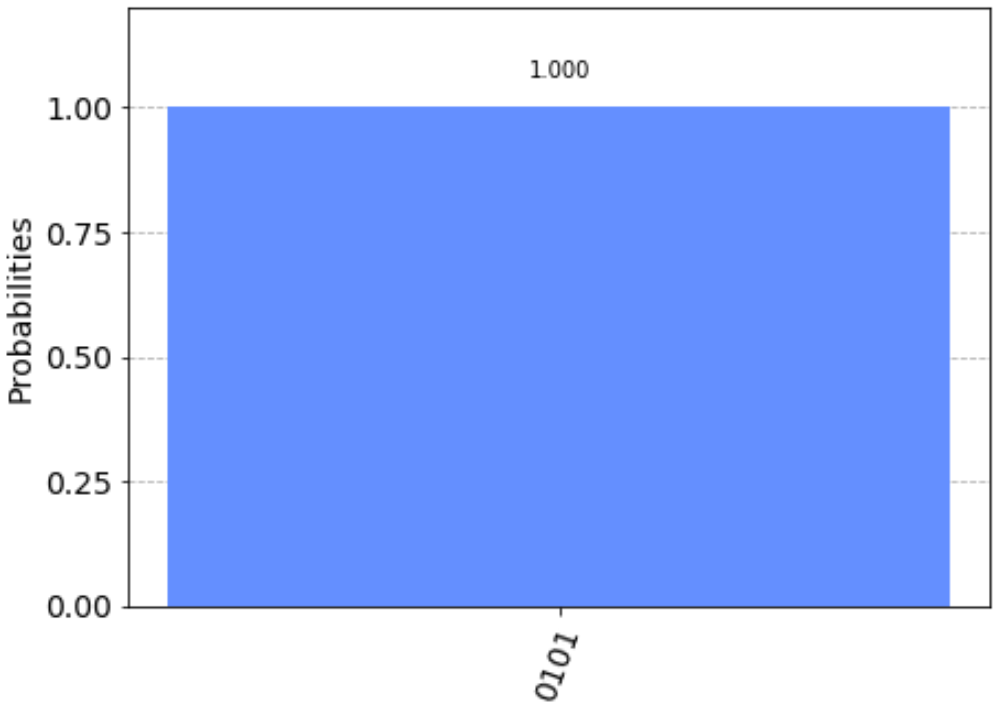
Out[334]:



In [335]:

```
res=simulator_result(qc,shots=4)
plot_histogram(res)
```

Out[335]:



Part 2: Success Probability Dependence On Grover's Iterates

2.1. At constant number of winners (M=1)

I will design a function that return the probability of getting the winners, given the number of qubits, winners, number of iterations.

In [364]:

```
#we will need to change the winner into its binary form, as this what appear
def dec_to_bin(x):
    return bin(x)[2:]
```

In [706]:

```
def win_prob(nqubits=3, winners=[5], iterations=0,shots=1024):
    #just rename our variables, to be easy to play
    n=nqubits
    w=winners
    i=iterations
    #grover circuit
    qc=grover_initialize(n)
    qc.append(amp_lify(n,w,i),range(n))
    qc.measure_all()
    #results of grover circuit
    res=simulator_result(qc,shots=shots)

    p_win=0 #I collect the probability of getting the winners here

    for j in range(len(w)): #run over the winners
        bin_win=(nqubits-len(dec_to_bin(w[j])))*'0'+dec_to_bin(w[j]) #binary
        try:
            p_win=p_win+(res[bin_win])/shots #the probability of getting the
        except:
            p_win=p_win  #in case if the winner does not exist at all
    return p_win
```

In this section I will utilize this function, searching for only one winner, and see how the success probability changes with the number of iterations at different numbers of qubits.

Let the plot begin

In [452]:

```
from tqdm import tqdm #to track the progress
```

In [453]:

```
iterations=list(range(1,44))
```

In [454]:

```
total_results=[]
#I will collect the winning probabilities here: [[winning probs with iterati
```

In [455]:

```
for nqubits in tqdm(range(3,11)):
    res=[]
    for i in iterations:
        res.append(win_prob(nqubits,iterations=i))
    print(res)  #I should save this, time is precious!
    total_results.append(res)
```

...

In [457]:

```
total_results
```

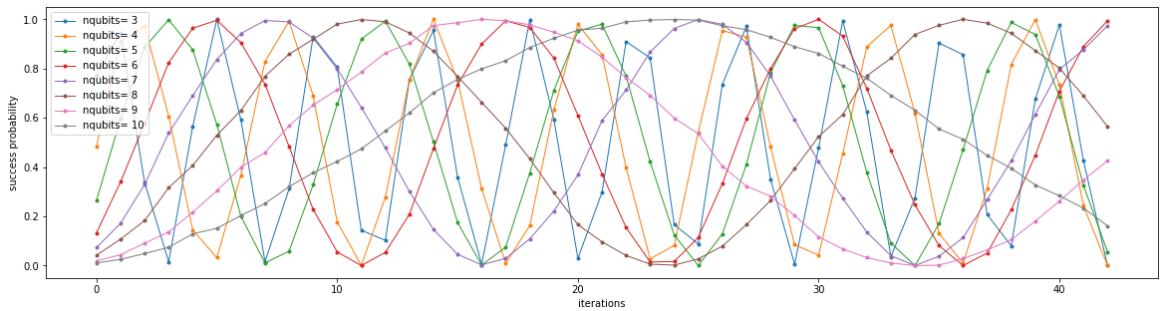
...

In [492]:

```
fig, ax = plt.subplots(figsize=(20,5))
for nqubits in range(0,8):
    ax.plot(total_results[nqubits], linewidth=1, linestyle='-', marker='.',l

ax.legend()
#ax.set_title("")
ax.set_xlabel("iterations")
ax.set_ylabel("success probability")
```

Out[492]:

```
Text(0, 0.5, 'success probability')
```



What a beauty! we see that the success probability of finding the winner oscillate harmonically with Amplitude Amplification iterations. The frequency of oscillation decrease with the number of iterations! We will call the first peak of this oscillation is the optimal iteration number.

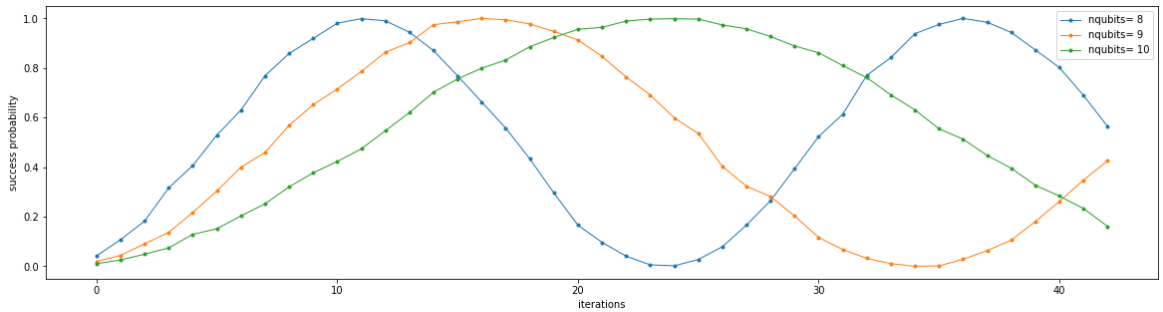
Lets see it more clearly for 8,9, and 10 qubits.

In [493]:

```
fig, ax = plt.subplots(figsize=(20,5))
for nqubits in range(5,8):
    ax.plot(total_results[nqubits], linewidth=1, linestyle='-', marker='.',l

ax.legend()
#ax.set_title("")
ax.set_xlabel("iterations")
ax.set_ylabel("success probability")
```

Out[493]:

```
Text(0, 0.5, 'success probability')
```



it is very suggestive to fit this data to:

$$sin^2(f\ t)$$

In [494]:

```
from scipy.optimize import curve_fit
```

In [578]:

```
#Fitting function
def prob(t, f):
    return (np.sin(f*t))**2
```

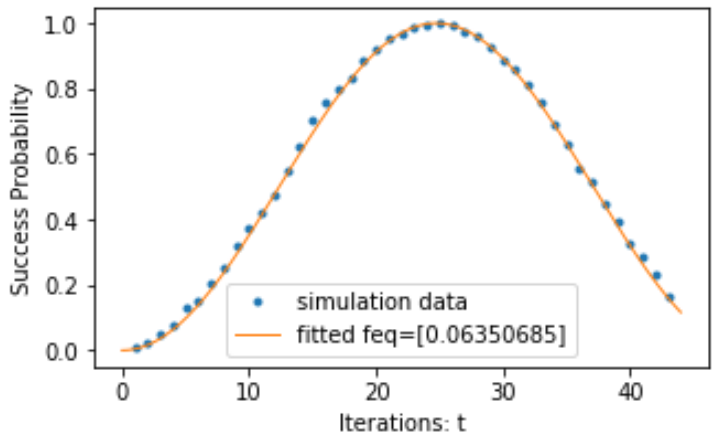
In [664]:

```
#demonstrate the fitting frequncey for 10 qubits
tFit=np.arange(0.0, 44.0, 0.01) #to make the plot of the fitted function
initial_guess=0.07
popt, pcov = curve_fit(prob, iterations, total_results[7],initial_guess)
print(popt)   #this is the fitting parameter
fig, ax = plt.subplots(figsize=(5,3))
ax.plot(iterations, total_results[7], '.', label="simulation data")
ax.plot(tFit,prob(tFit,popt), linewidth=1, label="fitted feq=%s"%popt)
plt.xlabel('Iterations: t')
plt.ylabel('Success Probability')
plt.legend()
```

```
[0.06350685]
```

Out[664]:

```
<matplotlib.legend.Legend at 0x7fa93cb88090>
```



I will find the frequencies for each number of qubits, then try to fit frequencies as a function of the number of qubits.

In [649]:

```
#for the curves P(t), I have these guesses for frequencies
f_guess=[0.7,0.5,0.35,0.25,0.2,0.125,0.1,0.05]
#lets make a list of fitted frequencies
f_fit=[]
for i in range(len(f_guess)):
    popt, pcov = curve_fit(prob, iterations, total_results[i],f_guess[i])
    f_fit.append(popt[0])
```

In [650]:

```
f_fit
```

Out[650]:

```
[0.7352950834334269,
 0.5143580712752043,
 0.3614748891635737,
 0.25526058345100144,
 0.18013617105923493,
 0.1271446138863935,
 0.08991746543084882,
 0.06350685361782697]
```

These are the frequency fits for $sin^2(f\ t)$. every value in the list above corresponds for a different number of qubits, starting from 3 to 10. Now we will fit these frequencies as a function of number of qubits. Using this Model:

$$f(n) = \frac{a}{\sqrt{b^n}}$$

This is actually the theoretically motivated dependence, with a=b=2.

In [661]:

```
#Fitting function
def f(n, a,b):
    return a/(np.sqrt(b**n))
```

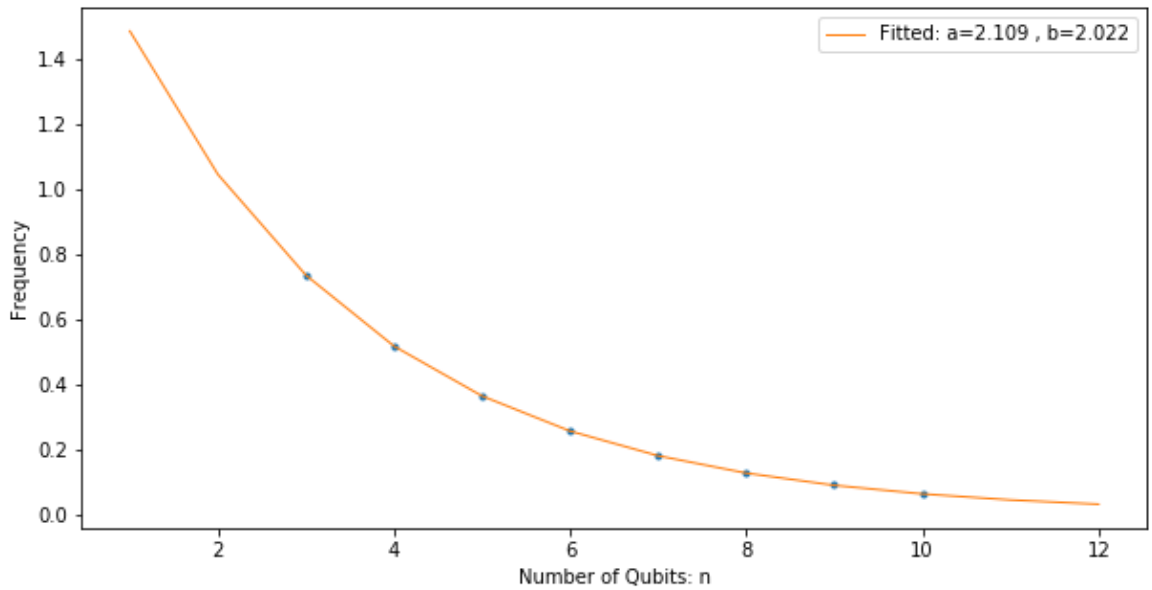
In [672]:

```
nq=list(range(3,11))
nqFit=np.arange(1, 13, 1) #to make the plot of the fitted function
popt, pcov = curve_fit(f, nq, f_fit)
print(popt)
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(nq, f_fit, '.')
ax.plot(nqFit,f(nqFit,popt[0],popt[1]), linewidth=1, label="Fitted: a=%s , b
plt.xlabel('Number of Qubits: n')
plt.ylabel('Frequency')
plt.legend()
```

```
[2.10938505 2.02166066]
```

Out[672]:

```
<matplotlib.legend.Legend at 0x7fa93980ea90>
```



This is over perfect fitting.

$$a \approx b \approx 2$$

Finally, the success probability of getting the wining item right (if we search for only one), as a function of Amplitude amplification iterations t and number of qubits n is, and list length $N = 2^n$:

$$\boxed{P_{success}(N,t) \approx sin^2\left(\frac{2}{\sqrt{N}}\ t\right)}$$

from the boxed equation we can get that the least number of iterations that can gives the maximum probability of getting the winner element is:

$$t \approx \frac{\pi}{4}\sqrt{N}$$

2.2. At constant number of Qubits (n=8, N= 28)

In [768]:

```
iterations=list(range(1,30))
```

In [769]:

```
winnerz=[] #this will include the different numbers of winners
#[[1 winner],[2 winners],[ 3 winners], ...]
```

In [770]:

```
for i in range(5,17):
    winnerz.append(list(range(5,i)))
winnerz.pop(0)
```

Out[770]:

```
[]
```

In [771]:

```
winnerz
```

Out[771]:

```
[[5],
 [5, 6],
 [5, 6, 7],
 [5, 6, 7, 8],
 [5, 6, 7, 8, 9],
 [5, 6, 7, 8, 9, 10],
 [5, 6, 7, 8, 9, 10, 11],
 [5, 6, 7, 8, 9, 10, 11, 12],
 [5, 6, 7, 8, 9, 10, 11, 12, 13],
 [5, 6, 7, 8, 9, 10, 11, 12, 13, 14],
 [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
```

In [772]:

```
total_results=[]
#I will collect the winning probabilities here:
#[[winning probs with iterations at 1 winner],[winning prob at 2 winners],..
```

In [773]:

```
for w in tqdm(range(len(winnerz))):
    res=[]
    for i in iterations:
        res.append(win_prob(nqubits=8,winners=winnerz[w],iterations=i))
    print(res)  #I should save this, time is precious!
    total_results.append(res)
```

...

In [775]:

```
total_results
```

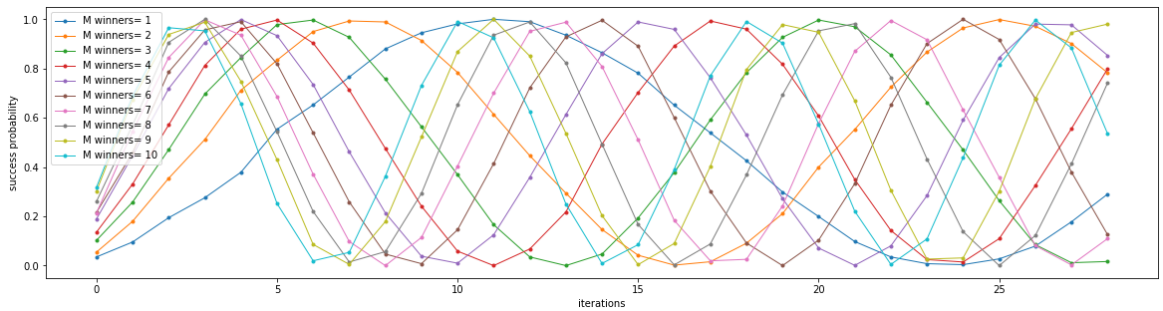
...

In [780]:

```
fig, ax = plt.subplots(figsize=(20,5))
for M in range(0,10):
    ax.plot(total_results[M], linewidth=1, linestyle='-', marker='.',label="

ax.legend()
#ax.set_title("")
ax.set_xlabel("iterations")
ax.set_ylabel("success probability")
```

Out[780]:

```
Text(0, 0.5, 'success probability')
```



this looks pretty similar to last sinusoidal graph, but this time frequency increase with number of winners.

it is very suggestive to fit this data to:

$$sin^2(f\ t)$$

In [781]:

```
#Fitting function
def prob(t, f):
    return (np.sin(f*t))**2
```

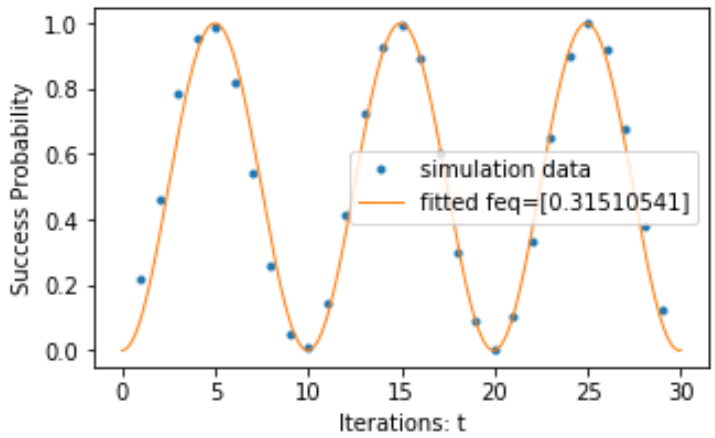
In [801]:

```
#demonstrate the fitting frequncey for 5 winners
tFit=np.arange(0.0, 30.0, 0.01) #to make the plot of the fitted function
initial_guess=0.3
popt, pcov = curve_fit(prob, iterations, total_results[5],initial_guess)
print(popt)
fig, ax = plt.subplots(figsize=(5,3))
ax.plot(iterations, total_results[5], '.', label="simulation data")
ax.plot(tFit,prob(tFit,popt), linewidth=1, label="fitted feq=%s"%popt)
plt.xlabel('Iterations: t')
plt.ylabel('Success Probability')
plt.legend()
```

```
[0.31510541]
```

Out[801]:

```
<matplotlib.legend.Legend at 0x7fa946e807d0>
```



I will find the frequencies for each number of qubits, then try to fit frequencies as a function of the number of qubits.

In [802]:

```
#for the curves P(t), I have these guesses for frequencies
f_guess=[0.13,0.18,0.23,0.26,0.29,0.32,0.34,0.36,0.39,0.41,0.42]
#lets make a list of fitted frequencies
f_fit=[]
for i in range(len(f_guess)):
    popt, pcov = curve_fit(prob, iterations, total_results[i],f_guess[i])
    f_fit.append(popt[0])
```

These are the frequency fits for $sin^2(f\ t)$. every value in the list above corresponds for a different number of winners, starting from 1 to 10. Now we will fit these frequencies as a function of number of winners. Using this Model:

$$f(M) = a\sqrt{\frac{M}{2^8}}$$

This is actually the theoretically motivated dependence for 8 qubits, with a=2.

In [817]:

```
def f(M, a):
    return a*np.sqrt(M/(2**8))
```

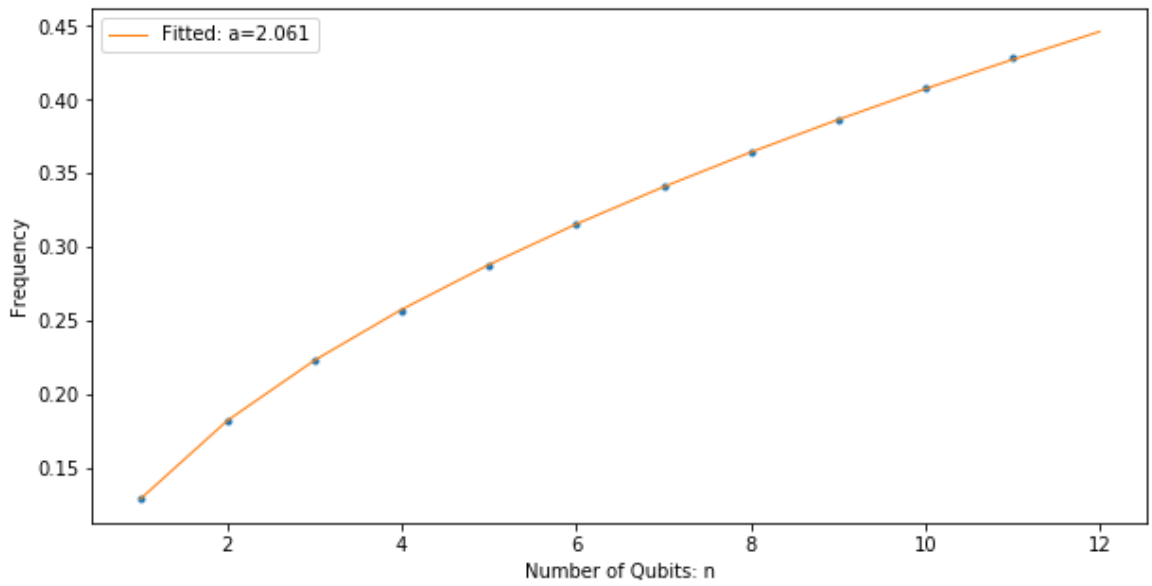
In [824]:

```
m=list(range(1,12))    #the number of winners
mFit=np.arange(1, 13, 1)  #to plot the fitted function
popt, pcov = curve_fit(f, m, f_fit)
print(popt)
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(m, f_fit, '.')
ax.plot(mFit,f(mFit,popt), linewidth=1, label="Fitted: a=%s"%(round(popt[0],
plt.xlabel('Number of Qubits: n')
plt.ylabel('Frequency')
plt.legend()
```

```
[2.06121832]
```

Out[824]:

```
<matplotlib.legend.Legend at 0x7fa944001150>
```



then $a \approx 2$, then $f(M, n = 8) \approx 2\sqrt{\frac{M}{2^8}}$

2.3. The Conclusive Equation

we just found that $f(M) \approx 2\sqrt{\frac{M}{2^8}}$ for eight qubits, as we found $f(n) \approx \frac{2}{\sqrt{2^n}}$ for one iteration.

Then we can conclude that:

$$f(M, n) \approx 2\sqrt{\frac{M}{2^n}}$$

Finally, the most conclusive equation for this work is:

$$P_{success}(N, M, t) \approx sin^2\left(2\sqrt{\frac{M}{N}}\ t\right)$$

N: Length of the list.

M: Number of winners.

t: Number of Grover's iterates. (# amplitude amplifications).