HW6

Xichen Li, EE521 - Group 5

A) Xichen Li: I did HW6 independently. The first part of the submission is problems in Python and the second half of the submission is the end of the chapter problems.

Grover's Algorithm on 1-IN-3-SAT

So far we've been focusing on fairly simple systems designed solely for observing and understanding mysterious behavior quantum mechanics. Now, we'll put our engineering hat's on and build a circuit that will actually solve a real (albeit small scale) problem. Although the problem we'll tackle is small, it is by no means easy. In fact, let's go straight for some of the hardest (yet most important) types of problems studied, a $\mathcal{NP}$-complete problem!

First, a quick recap on complexity theory, so we can really appreciate what we're about to do. Algorithms can be compared in terms of how much space (memory) or time (number of operations) they require, in terms of their input size (denoted N). Algorithms can be classified into different sets, for example, the set of polynomial algorithms (denoted $\mathcal{P}$) comprises of all the algorithms that require on the order of a polynomial steps in the input, denoted $\mathcal{O}(N^K)$ for some K. Another important set is the set of non-deterministic polynomial algorithms (denoted $\mathcal{NP}$). A subset of $\mathcal{NP}$ problems are $\mathcal{NP}$-complete, so called because any problem in $\mathcal{NP}$ can efficiently be transformed into an $\mathcal{NP}$-complete problem.

One of the most famous unsolved problems in mathematics is "$\mathcal{P}$ vs $\mathcal{NP}$" which asks whether or not the sets $\mathcal{P}$ and $\mathcal{NP}$ are actually the same. It's hard to understate the importance of this question because if it turns out $\mathcal{P} = \mathcal{NP}$ then that would mean there exists an efficient (meaning polynomial time) algorithm to solve any $\mathcal{NP}$ problem, and many of the most interesting problems we would like to solve are in $\mathcal{NP}$. One way to prove $\mathcal{P} = \mathcal{NP}$ is to develop an efficient algorithm for any one $\mathcal{NP}$-complete problem. Currently, the only algorithms we know to solve $\mathcal{NP}$-complete problems are exponential in time ($\mathcal{O}(K^N)$ for some K). A very popular family of $\mathcal{NP}$-complete problems solve for satisfiability of boolean expressions, which is what we'll be investigating.

Now before you start thinking you have to solve "$\mathcal{P}$ vs $\mathcal{NP}$" for homework (but extra credit if you do, in addition to instant world fame, a Field's medal, and millions of dollars worth of prizes!), what we're actually going to do is apply Grover's algorithm to the 1-IN-3-SAT problem. Grover's algorithm takes advantage of quantum mechanics to find solutions to a problem in $\mathcal{O}(\sqrt{N})$, where N is the number of possibilities (eg. database size), which allows us to solve the 1-IN-3-SAT problems faster than any classical algorithm.

In [245]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from qiskit import BasicAer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute
from qiskit.circuit.library import MCMT
from qiskit.tools.visualization import plot_histogram
```

Quantum Boolean Logic

Before diving in to $\mathcal{NP}$-Complete problems and quantum circuits that are several pages long, let's get an idea for boolean logic with quantum circuits.

There are two interesting properties of quantum circuits to consider: reversibility and no cloning. Remember that all the quantum circuits (up to measurement) are in theory fully reversible. Additionally, the no-cloning theorem tells us that given a qubit in some unknown state, there's no way to duplicate the state on another qubit.

These two properties require us to use separate input and output registers (qubits), and to be careful about undoing any operations that change our input qubits for future operations. Additionally, we will employ auxiliary qubits, or "ancilla," that will store intermediate values during computation. Just as we have to be hygenic about our input register, we will have to undo our operations on the ancilla to keep their state consistent (also referred to as keeping them "clean").

With this formalism, there are a handful of common logic gates that are particularly easy to implement: `NOT`, `XOR`, and `AND`. This small set of gates is already complete, which means we can build any boolean logic circuit with these gates alone. The three functions below implement classical `AND`, `NOT`, `XOR` gates in a quantum circuit (`circ`) given the input (`a`, `b`) and output (`y`) quantum registers.

In [246]:
```
def AND(circ,out,a,b): # using a Toffoli gate
    circ.ccx(a,b,out)
def NOT(circ,y): # using an X gate (same output register as input)
    circ.x(y)
def XOR(circ,out,*inputs): # using 2 CNOT gates
    for i in inputs:
        circ.cx(i,out)

def init_superposition(circ, fin): # initialize our input states as superpositions, so we test all boolean inputs
    circ.h(fin)
```

As an example we can implement the expression V:

$$V = (x_2 \oplus x_1 \oplus x_0) \wedge \neg x_2$$

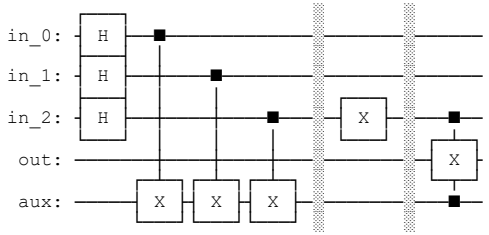
where the x_is are boolean variables, and $\oplus$ is the exclusive "or" (`XOR`) operation.

A quick calculation tells us V is true if and only if $x_2 x_1 x_0 \in \{001, 010\}$. So let's confirm this by simulating the eqivalent quantum circuit using qiskit.

In [247]:
```
f_in = QuantumRegister(3, 'in')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(1, 'aux')
qc = QuantumCircuit(f_in, f_out, aux)
init_superposition(qc, f_in)
```

In [248]:
```
XOR(qc, aux[0], f_in[0], f_in[1], f_in[2])
qc.barrier()
NOT(qc, f_in[2])
qc.barrier()
AND(qc, f_out[0], aux[0], f_in[2])
#qc.draw(output='mpl')
qc.draw()
```

Out[248]:

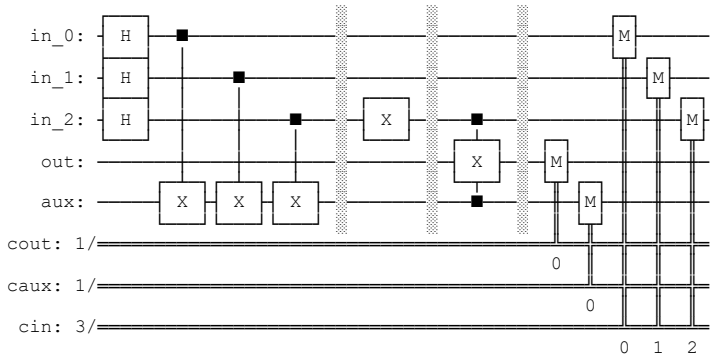


After applying all these gates, we will measure all of our qubits (including the ancilla).

In [249]:
```
def measure(circ, q, c=None):
    if c is None:
        c = ClassicalRegister(len(q), 'c{}'.format(q.name))
        circ.add_register(c)
    circ.measure(q,c)
```

In [250]:
```
qc.barrier()
measure(qc, f_out)
measure(qc, aux)
measure(qc, f_in)
#qc.draw(output='mpl')
qc.draw()
```

Out[250]:



Don't worry about `qiskit` having apparently moved some of the measurement operations before applying some of our gates. `qiskit` is just optimizing to measure each qubit as soon as possible to make our overall state space as small as possible, which will make the simulation more efficient. These optimizations will not change anything about the outcomes. Now we can simulate our circuit with 1000 trials using the `qasm_simulator`.

In [251]:
```
backend = BasicAer.get_backend('qasm_simulator')
job = execute(qc, backend=backend)
result = job.result()
print('f_in aux f_out')
result.get_counts()
```

```
f_in aux f_out
```

Out[251]:
```
{'010 0 0': 149,
 '000 1 0': 136,
 '101 1 1': 156,
 '011 1 0': 129,
 '110 1 1': 115,
 '001 0 0': 104,
 '111 0 0': 115,
 '100 0 0': 120}
```

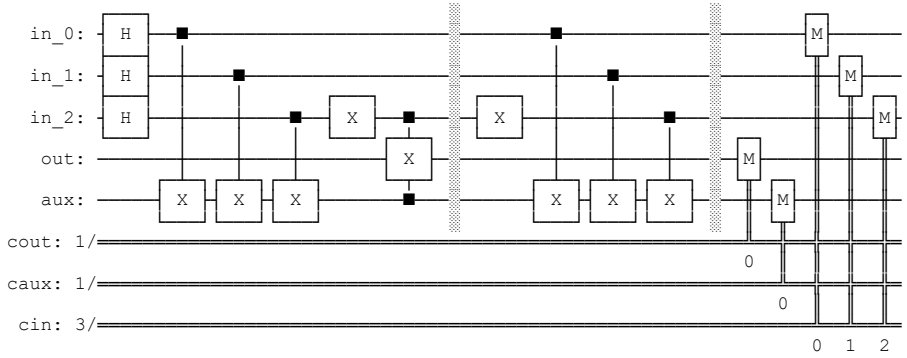
In this case we are less interested in how often we get each result, and more interested in what results we get. You'll notice that there are spaces in between the binary outcomes for each bit to separate each of the registers we defined. Given the order we defined our gates, the first group of three values correspond to our input bits (`f_in`), the second number corresponds to the ancilla (`aux`), and the final number to the output bit (`f_out`). The input bits are printed in decending order (x_2, x_1, and then x_0).

However, you will notice there are two problems: first our ancilla are not always 0 (which means they are no longer clean), and secondly, perhaps more alarmingly, we don't get the correct results. Based on the simulator our expression evaluates to true when for inputs of 101 and 110. Both issues actually have the same cause: we didn't undo our operations.

Try rebuilding the circuit, but now, adding some gates after we write our result to the output qubit.

In [252]:
```
f_in = QuantumRegister(3, 'in')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(1, 'aux')
qc = QuantumCircuit(f_in, f_out, aux)
init_superposition(qc, f_in)

XOR(qc, aux[0], f_in[0], f_in[1], f_in[2])
NOT(qc, f_in[2])
AND(qc, f_out[0], aux[0], f_in[2]) # Computation complete - result is "written" to f_out
qc.barrier()

# Undo operations to input qubits and ancilla
NOT(qc, f_in[2])
XOR(qc, aux[0], f_in[0], f_in[1], f_in[2])

qc.barrier()
measure(qc, f_out)
measure(qc, aux)
measure(qc, f_in)

qc.draw()
```

Out[252]:



In [253]:
```
backend = BasicAer.get_backend('qasm_simulator')
job = execute(qc, backend=backend)
result = job.result()
print('f_in aux f_out')
result.get_counts()
```

```
f_in aux f_out
```

Out[253]:
```
{'100 0 0': 139,
 '000 0 0': 120,
 '110 0 0': 119,
 '010 0 1': 125,
 '101 0 0': 129,
 '001 0 1': 130,
 '011 0 0': 117,
 '111 0 0': 145}
```

Note that now our ancilla is always 0, and we can recover the correct results.

The lesson here is that if want to compute an arbitrary function using a quantum circuit, we must use a separate input and output register, in addition to some number of ancilla (theoretically you only ever need a single ancilla qubit, but often the circuit can be greatly simplified by using multiple ancilla). After completing the computation of the function (meaning all necessary gates are applied to the output qubits), we have to undo the computation, by reapplying all the gates (except those that write the results to the output qubits) in reverse order.

Problem 1: An and and and

Try implementing another simple logical expression W:

$$W = x_2 \wedge \neg x_1 \wedge x_0$$

Hint 1: You should not need more than one ancilla in your circuit.

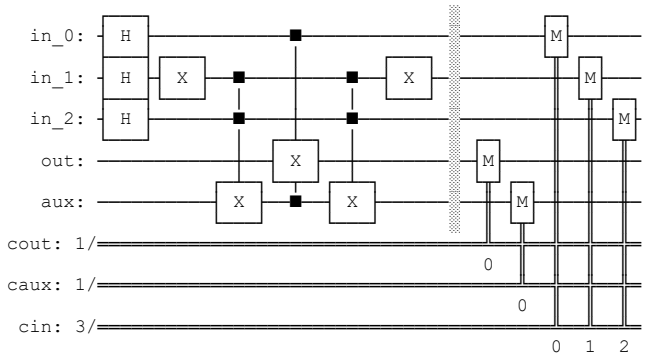
Hint 2: Remember to undo your computation to keep your input qubits and ancilla nice and clean. Your ancilla should always come out to zero, and the output should only be true when $x_2x_1x_0 = 101$.

Answer

In [254]:
```
f_in = QuantumRegister(3, 'in')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(1, 'aux')
qc = QuantumCircuit(f_in, f_out, aux)
init_superposition(qc, f_in)
```

In [255]:
```
#XOR(qc, aux[0], f_in[0], f_in[1], f_in[2])
#NOT(qc, f_in[2])
#AND(qc, f_out[0], aux[0], f_in[2]) # Computation complete - result is "written" to f_out
#qc.barrier()

# Undo operations to input qubits and ancilla
#NOT(qc, f_in[2])
#XOR(qc, aux[0], f_in[0], f_in[1], f_in[2])

NOT(qc, f_in[1])
AND(qc, aux[0], f_in[2], f_in[1])
AND(qc, f_out[0], aux[0], f_in[0]) # Computation complete - result is "written" to f_out

# Undo operations to input qubits and ancilla
AND(qc, aux[0], f_in[2], f_in[1])
NOT(qc, f_in[1])

qc.barrier()
measure(qc, f_out)
measure(qc, aux)
measure(qc, f_in)

qc.draw()
```

Out[255]:



In [256]:
```
backend = BasicAer.get_backend('qasm_simulator')
job = execute(qc, backend=backend)
result = job.result()
print('f_in aux f_out')
result.get_counts()
```

```
f_in aux f_out
```

Out[256]:
```
{'011 0 0': 146,
 '111 0 0': 127,
 '100 0 0': 111,
 '010 0 0': 130,
 '101 0 1': 132,
 '001 0 0': 133,
 '110 0 0': 117,
 '000 0 0': 128}
```

As we can see from the results above, the ancilla always come out to zero, and the output is only true when $x_2x_1x_0 = 101$.

1-IN-3-SAT

Satisfying boolean expression is a rather important problem in science and engineering. One especially interesting application, is where variables correspond to decisions and the expression might encode constraints or consequences of choices. The goal is to find an assignment of all the variables that will complete, or satisfy, the overall problem. Unfortunately, SAT, and the related 3-SAT problems are $\mathcal{NP}$-complete, so there might not exist an efficient algorithm for finding assignments to the variables which will satisfy the expression.

Just like 3-SAT, our subject, 1-IN-3-SAT, starts with a boolean expression comprising of some number of clauses using N boolean variables ($x_i \in \{0, 1\}$).

Let E be our input expression made up of M clauses, where c_j is the jth clause, then,

$$E = c_0 \wedge c_1 \wedge \ldots \wedge c_{M-2} \wedge c_{M-1}$$

Each clause c_j is a disjunction of three variables (possibly negated). For example, c_1 might contain x_0, x_1, and x_2, so that

$$c_1 = x_0 \vee \neg x_1 \vee x_2$$

where $\neg$ is the `NOT` operator, $\wedge$ is the `AND` operator, and $\vee$ is the `OR` operator.

Given expression E, containing N variables, we have to find the assignments of the variables (x_i) which will satisfy E. However, so far this is just the 3-SAT problem, so there's the additional twist that in each of our clauses c_j exactly one of the terms is true, while the other two must be false. These additional constraints can be encoded by transforming our example clause c_1 to g_1:

$$g_1 = x_0 \oplus \neg x_1 \oplus x_2 \oplus (x_0 \wedge \neg x_1 \wedge x_2)$$

Now g_1 will only be true if c_1 is true, and only one of the terms (x_0, $\neg x_1$, and x_2) are true.

Therefore, the problem in 1-IN-3-SAT is, given an expression E to find assignments to our variables x_i to satisfy the transformed expression E':

$$E' = g_0 \wedge g_1 \wedge \ldots \wedge g_{M-2} \wedge g_{M-1}$$

where each clause g_j is the result of transforming c_j as described above.

Note that our transformed clauses include only `NOT` , `AND` , and `XOR` operations, which makes this problem relatively easy to implement using common quantum gates.

Below is a function that takes in an expression E and implements the corresponding E' on circuit `circ` to be used in Grover's algorithm.

The expression (`expression`) that is passed in to the function must be a `list` of clauses (we'll limit ourselves to 3 clause expressions). Each clause is a list of terms in the form of integers. The value of each integer refers to the variable index, and the sign to whether or not the variable is negated.

For example, if:

$$E = (x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee x_1 \vee x_2)$$

Then the corresponding expression becomes `expression = [[1, 2, -3], [-1, -2, -3], [-1, 2, 3]]` .

Here $i+1$ corresponds to x_i and $-(i+1)$ corresponds to $NOT(x_i)$, where $i = 0, 1, 2$. We have changed x_i to $i+1$ only for programming convenience (see program below).

The answer to this expression is $x_0 = 1, x_1 = 0, x_2 = 1$, satisfy the 1-IN-3-SAT requirement.

$$\begin{aligned} \Rightarrow E &= (1 \vee 0 \vee \neg 1) \wedge (\neg 1 \vee \neg 0 \vee \neg 1) \wedge (\neg 1 \vee 0 \vee 1) \\ E &= (1 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 0) \wedge (0 \vee 0 \vee 1) \end{aligned}$$

Grover's Algorithm

So far when we simulated our logic expressions, we ran many trials to get an estimate of all the possible outcomes, and then we could infer which assignments of the variables would produce an output of 1. However, if our expression is large, and comprises of many variables, then it may take exponentially (in the number of variables) many trials before we will observe a satisfying assignment (ie. the output is measured to be 1).

Consequently, we want to increase the probability of measuring our output qubit to be 1. Grover's algorithm is an iterative algorithm which will do just that. In each step of Grover's algorithm, we first evaluate the input expression from a superposition of all possible states invert only those states we are interested in (this called applying the "oracle" operator), and then we will invert the amplitudes about the mean, which will increase the probability of measuring the desired outcome.

The `apply_oracle` function takes care of the first step, and below the function `invert_about_avg` executes the second step.

In [257]:
```python
def apply_oracle(circ, f_in, f_out, aux, n, expression):
    num_clauses = len(expression)
    for (k, clause) in enumerate(expression):
        # This loop ensures aux[k] is 1 if an odd number of literals are true
        for literal in clause:
            if literal > 0:
                circ.cx(f_in[literal-1], aux[k])
            else:
                circ.x(f_in[-literal-1])
                circ.cx(f_in[-literal-1], aux[k])

        # Flip aux[k] if all literals are true, using auxiliary qubit (ancilla) aux[num_clauses]
        circ.barrier(f_in)
        circ.ccx(f_in[0], f_in[1], aux[num_clauses])
        circ.ccx(f_in[2], aux[num_clauses], aux[k])
        circ.barrier(f_in)

        # Flip back to reverse state of negative literals and ancilla
        circ.ccx(f_in[0], f_in[1], aux[num_clauses])
        for literal in clause:
            if literal < 0:
                circ.x(f_in[-literal-1])
        circ.barrier()

    # The formula is satisfied if and only if all auxiliary qubits
    # except aux[num_clauses] are 1
    if (num_clauses == 1):
        circ.cx(aux[0], f_out[0])
    elif (num_clauses == 2):
        circ.ccx(aux[0], aux[1], f_out[0])
    elif (num_clauses == 3):
        circ.ccx(aux[0], aux[1], aux[num_clauses])
        circ.ccx(aux[2], aux[num_clauses], f_out[0])
        circ.ccx(aux[0], aux[1], aux[num_clauses])
    else:
        raise ValueError('We only allow at most 3 clauses')
    circ.barrier()

    # Flip back any auxiliary qubits to make sure state is consistent
    # for future executions of this routine; same loop as above.
    for (k, clause) in enumerate(expression):
        for literal in clause:
            if literal > 0:
                circ.cx(f_in[literal-1], aux[k])
            else:
                circ.x(f_in[-literal-1])
                circ.cx(f_in[-literal-1], aux[k])
        circ.barrier(f_in)
        circ.ccx(f_in[0], f_in[1], aux[num_clauses])
        circ.ccx(f_in[2], aux[num_clauses], aux[k])
        circ.barrier(f_in)
        circ.ccx(f_in[0], f_in[1], aux[num_clauses])
        for literal in clause:
            if literal < 0:
                circ.x(f_in[-literal-1])
        circ.barrier()
```

In [258]:
```
def CnX(circ, ctrls, target, clean_aux=None):
    assert len(ctrls) >= 1, 'No control qubits specified'
    if not isinstance(ctrls, list):
        ctrls = tuple(ctrls)
    if len(ctrls) == 1:
        try:
            circ.cx(ctrls[0], target[0])
        except:
            circ.cx(ctrls[0], target)
    elif len(ctrls) == 2:
        try:
            circ.ccx(ctrls[0], ctrls[1], target[0])
        except:
            circ.ccx(ctrls[0], ctrls[1], target)
    else:
        aux = clean_aux
        if aux is None or len(aux) != len(ctrls)-2:
            print('Did not receive the right number of ancillae - automatically creating new ones: got {}, need {}'.format(len(aux),len(ctrls)-2))
            aux = QuantumRegister(len(ctrls)-2, 'aux')
            circ.add_register(aux)
            aux = list(aux)
        if not isinstance(aux, list):
            aux = list(aux)
        a,b,y = ctrls.pop(), ctrls.pop(), aux.pop()
        circ.ccx(a,b,y)
        ctrls.append(y)
        CnX(circ, ctrls, target, aux)
        circ.ccx(a,b,y)

def CnZ(circ, ctrls, target, clean_aux=None):
    circ.h(target)
    CnX(circ, ctrls, target, clean_aux)
    circ.h(target)

def invert_about_avg(circ, bits, aux=None):
    try:
        bits[1:]
        gbits = bits
    except:
        gbits = tuple(bits)
    circ.h(bits)
    circ.x(bits)
    circ.barrier(bits)
    CnZ(circ, gbits[:-1], gbits[-1], aux)
    circ.barrier(bits)
    circ.x(bits)
    circ.h(bits)

def init_grover(circ, fin, fout):
    circ.h(fin)
    circ.x(fout)
    circ.h(fout)
```

Let us first understand each step of this circuit of Grover's algorithm. The example circuit below has 3 input qubits (x_0, x_1, x_2) and one clause in expression. The output qubit is out_0, the aux qubits (extra bits to help with computation) are aux_0, aux_1. We will only perform one iteration of Grover's algorithm on this example.

Remember that $x_0, x_1, x_2, out_0, aux_0, aux_1$ are all zeros at the begining.

With `experssion = [[1, 2, -3]]`, we have experssion:

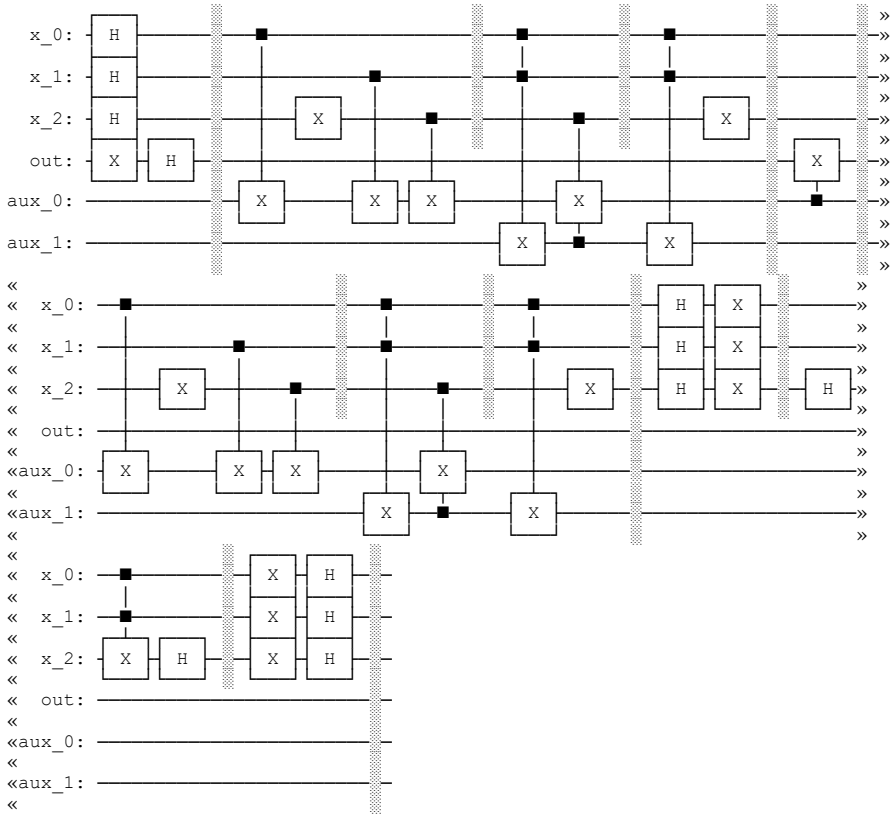
$$E = c_1 = x_0 \vee x_1 \vee \neg x_2$$

Then we have 1-IN-3-SAT transformed expression:

$$E' = g_1 = x_0 \oplus x_1 \oplus \neg x_2 \oplus (x_0 \wedge x_1 \wedge \neg x_2)$$

In [259]:
```
# In the code below x0, x1, x2 are represented by [1, 2, 3]
n = 3
expression = [[1, 2, -3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)

T = 1 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)
grover.barrier()
grover.draw()
```

Out[259]:



The following steps corresponding to each block (separate by long gray vertical bars) in the circuit diagram above:

1. Initialize the input qubits and output qubit of Grover's algorithm
2. Apply $\hat{V}$ (oracle reflection transformation) with 1st clause of expression to the input qubits. (Creating transformed expression E') The following small steps are separated by short gray vertical bars.
   A. $aux_0 = x_0 \oplus x_1 \oplus \neg x_2$
   B. $< aux_1 = x_0 \wedge x_1 >$
   $< aux_0 = aux_1 \wedge x_2 = x_0 \wedge x_1 \wedge x_2 >$
   Now we obtain the g_1 clause on aux_0: $aux_0 = aux_0 \oplus (x_0 \wedge x_1 \wedge x_2) = x_0 \oplus x_1 \oplus \neg x_2 \oplus (x_0 \wedge x_1 \wedge x_2)$
   C. The last two gates will flip back the input qubits (x_0, x_1, x_2) and aux qubit (aux_1) to make sure state is consistent. (control quibit don't need to flip back).
   `FYI, there will be 2nd, 3rd, ... clauses if we have to do them.`
3. Apply the result to the output qubit
   $out_0 = out_0 \oplus aux_0$
4. Flip back all the qubits except out_0. Apply the exact same gates as step 2.
5. Apply $\hat{W}$ (invert and average transformation) to the input qubits. See detail about this transformation at Example 6.5.7.8 in Chap 6 Lecture note (Page 47-48).

Now we can put all the steps together and build the circuit that applies Grover's algorithm to the desired expression to solve the 1-IN-3-SAT problem.
The example circuit below has 3 input qubits (x_0, x_1, x_2) and three clauses in expression. The output qubit is out_0, the aux qubits (extra bits to help with computation) are aux_0, aux_1. We will only perform one iteration of Grover's algorithm on this example.

In [260]:
```
n = 3
expression = [[1, 2, -3], [-1, -2, -3], [-1, 2, 3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)
```

Since Grover's algorithm is iterative we can choose how many steps to take before measuring. It turns out, that we actually have to be careful not to take too many steps, which will cause overrotation, which means our desired outcomes actually become less likely again.

In [261]:
```
T = 2 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)
```

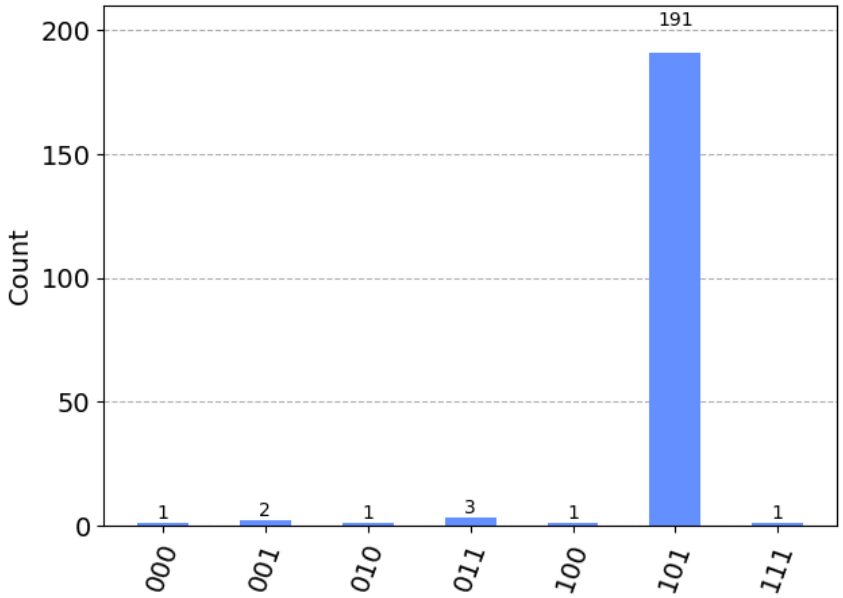
Since we are actually only interested in the assignments of the input variables, we don't even have to measure the output qubit.

In [262]:
```
grover.barrier()
measure(grover, f_in)
```

Now we can simulate our circuit, and run many trials to get an idea of how likely each outcome is. This is a fairly large circuit, so don't worry if the simulator needs a couple seconds.

In [263]:
```
# Execute circuit
backend = BasicAer.get_backend('qasm_simulator')
job = execute([grover], backend=backend, shots=200)
result = job.result()
counts = result.get_counts(grover)
plot_histogram(counts)
```

Out[263]:

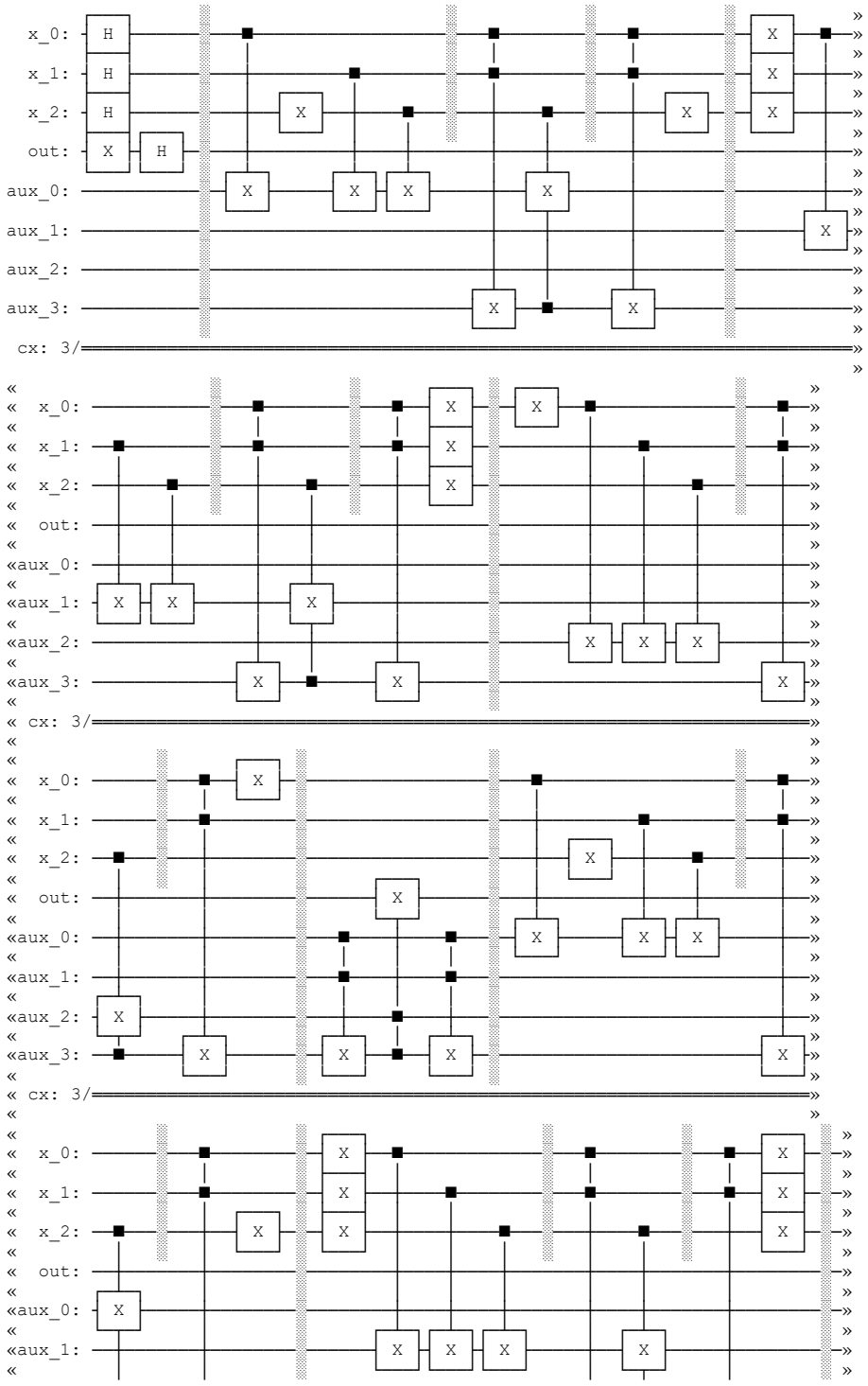


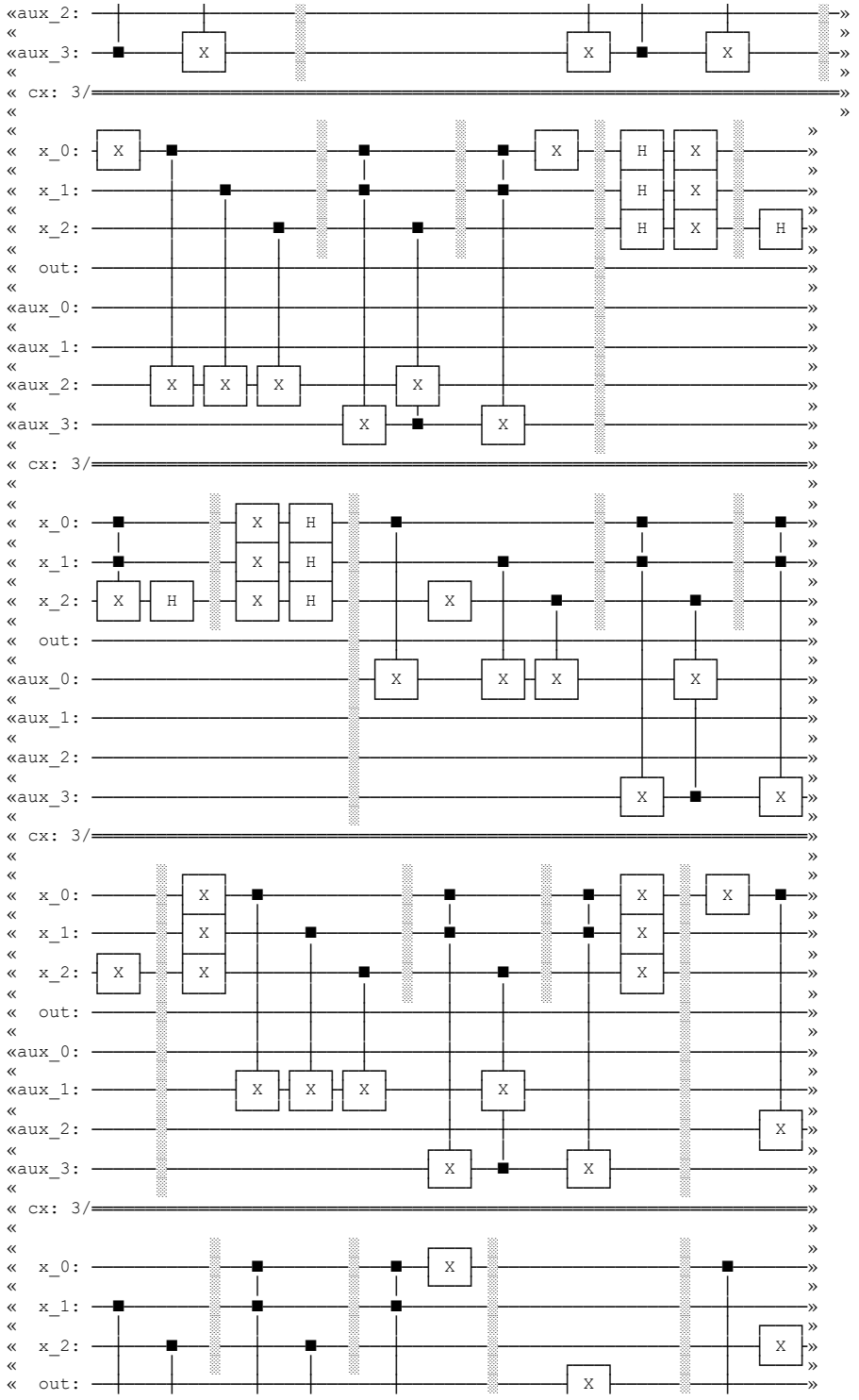
As you can see, we measure the outcome 101 more than of the time. The corresponding assignment does in fact satisfy the expression just like we wanted.

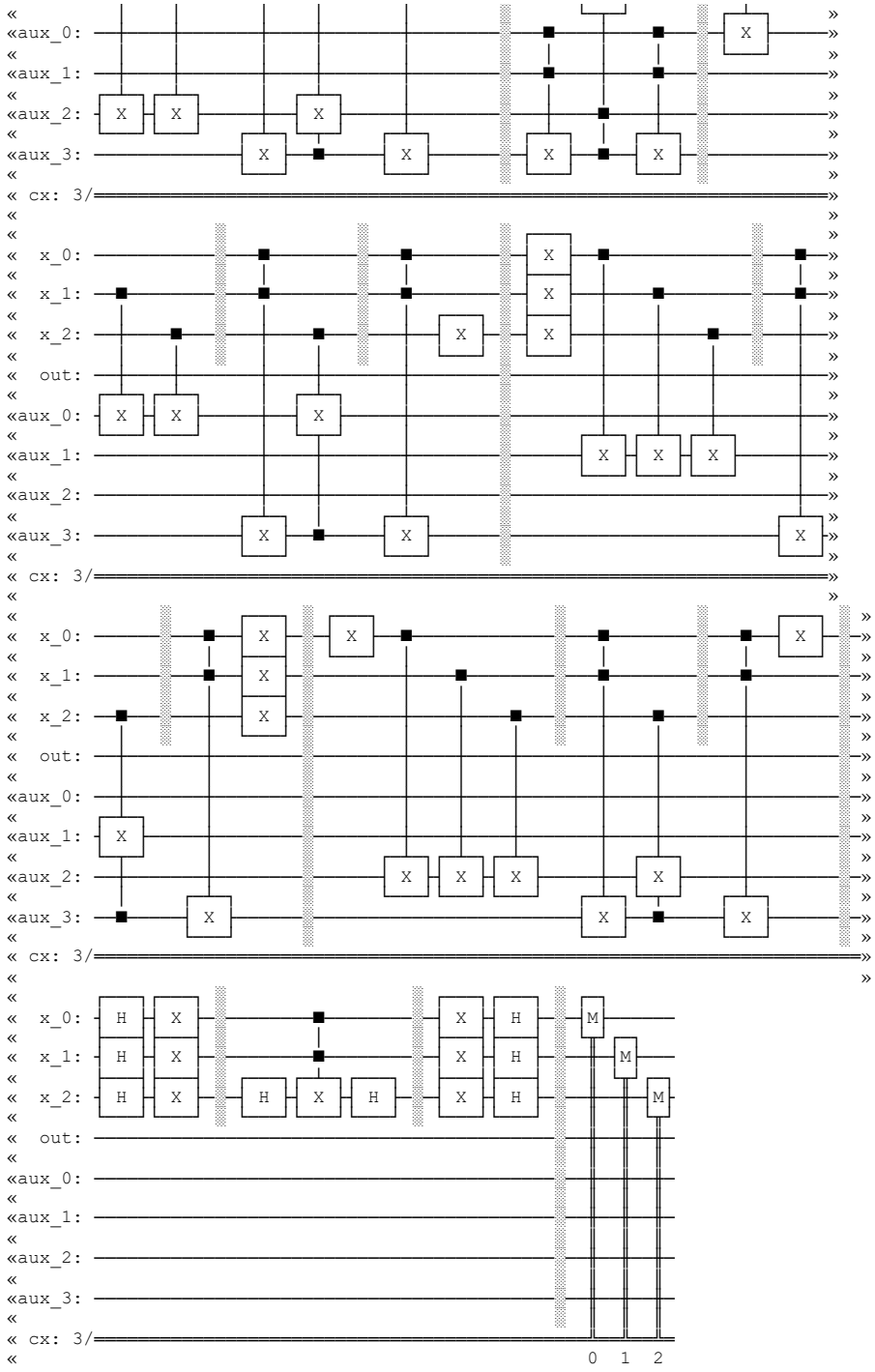
We can also see just how large our circuit actually is. Considering we have to apply our oracle gate once for each step, all the while making sure to keep our ancilla and input clean, you might not be surprised how large our gate can get.

In [264]:
```
grover.draw()
```

Out[264]:




«aux_2:
«aux_3:
« cx: 3/
« x_0:
« x_1:
« x_2:
« out:
«aux_0:
«aux_1:
«aux_2:
«aux_3:
« cx: 3/
« x_0:
« x_1:
« x_2:
« out:
«aux_0:
«aux_1:
«aux_2:
«aux_3:
« cx: 3/
« x_0:
« x_1:
« x_2:
« out:
«aux_0:
«aux_1:
«aux_2:
«aux_3:
« cx: 3/
« x_0:
« x_1:
« x_2:
« out:


aux_0:
aux_1:
aux_2:
aux_3:
cx: 3/
x_0:
x_1:
x_2:
out:
H
M
0 1 2

Problem 2: Solving 1-IN-3-SAT

Given:

$$F = (x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee x_2)$$

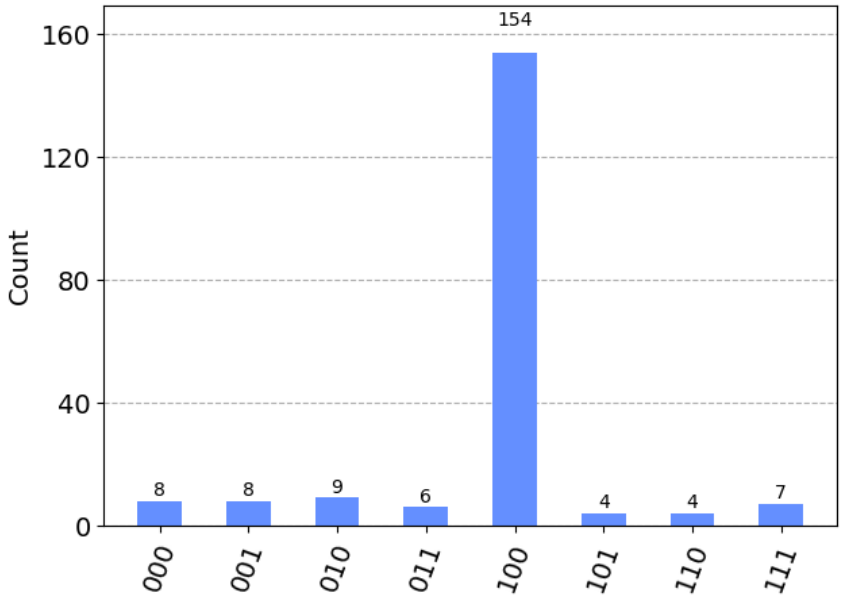
First, use Grover's algorithm to solve the 1-IN-3-SAT problem for F, that is, find an assignment of x_0, x_1, and x_2 that will satisfy the expression F, given that only a single term in each expression can be true.

Try running the circuit using 1, 2 and 4 steps, and report the probabilities of all outcomes each time. How do the results change when you change the timesteps? Why do they change in that way?

Answer

In [265]:
```
n = 3
expression = [[1, -2, -3], [-1, 2, -3], [1, 2, 3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)

T = 1 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)

grover.barrier()
measure(grover, f_in)

# Execute circuit
backend = BasicAer.get_backend('qasm_simulator')
job = execute([grover], backend=backend, shots=200)
result = job.result()
counts = result.get_counts(grover)
plot_histogram(counts)
```

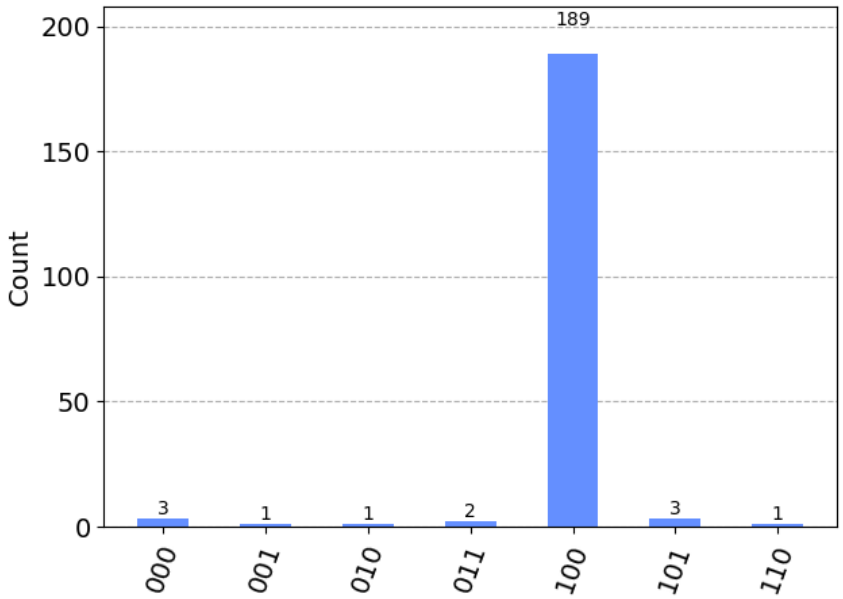
Out[265]:



The answer to this expression is $x_0 = 1, x_1 = 0, x_2 = 0$, satisfy this 1-IN-3-SAT requirement.
The code below tests different time step.

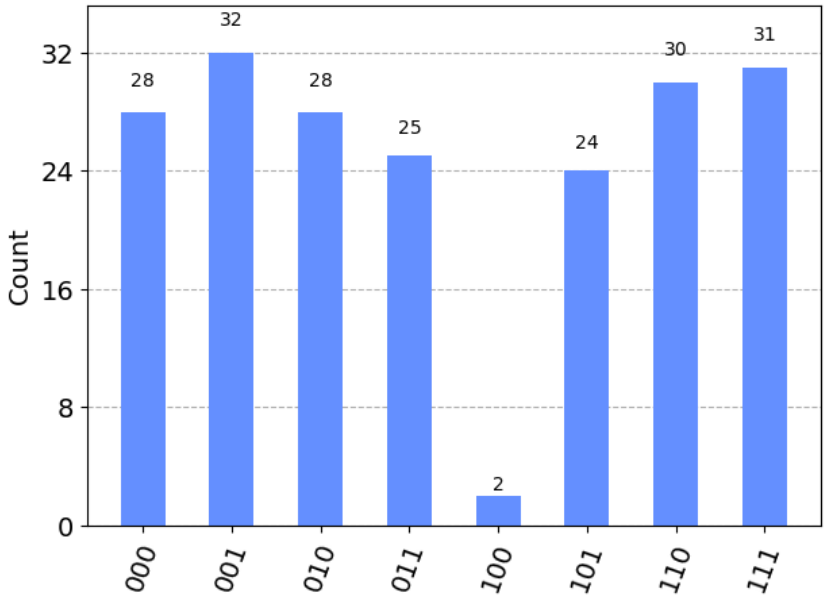
In [266]:
```
n = 3
expression = [[1, -2, -3], [-1, 2, -3], [1, 2, 3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)

T = 2 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)

grover.barrier()
measure(grover, f_in)

# Execute circuit
backend = BasicAer.get_backend('qasm_simulator')
job = execute([grover], backend=backend, shots=200)
result = job.result()
counts = result.get_counts(grover)
plot_histogram(counts)
```

Out[266]:



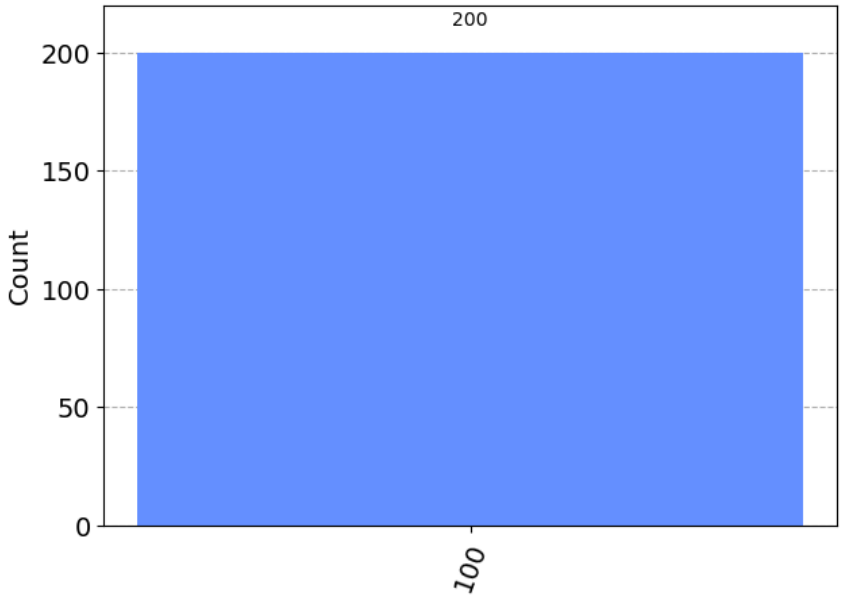
In [267]:
```
n = 3
expression = [[1, -2, -3], [-1, 2, -3], [1, 2, 3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)

T = 4 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)

grover.barrier()
measure(grover, f_in)

# Execute circuit
backend = BasicAer.get_backend('qasm_simulator')
job = execute([grover], backend=backend, shots=200)
result = job.result()
counts = result.get_counts(grover)
plot_histogram(counts)
```

Out[267]:



In [268]:
```
n = 3
expression = [[1, -2, -3], [-1, 2, -3], [1, 2, 3]]

f_in = QuantumRegister(n, 'x')
f_out = QuantumRegister(1, 'out')
aux = QuantumRegister(len(expression) + 1, 'aux') # our oracle requires n+1 ancilla
circ = QuantumCircuit(f_out, f_in, aux)

grover = QuantumCircuit()
grover.add_register(f_in)
grover.add_register(f_out)
grover.add_register(aux)

T = 6 # number of steps to take

init_grover(grover, f_in, f_out)
for t in range(T):
    grover.barrier()
    apply_oracle(grover, f_in, f_out, aux, n, expression)
    invert_about_avg(grover, f_in, aux)

grover.barrier()
measure(grover, f_in)

# Execute circuit
backend = BasicAer.get_backend('qasm_simulator')
job = execute([grover], backend=backend, shots=200)
result = job.result()
counts = result.get_counts(grover)
plot_histogram(counts)
```

Out[268]:



I am expecting to see the probablities of getting '100' increases when I increase the number of steps. When I change T from 1 to 2, the samples of '100' increases as expected. However, when I set T to 4, the samples of getting '100' decreases which mean the calculation fails. I also tried to further increase the value of T. When T equals to 6, as you can see from the figure above, the probablity of getting '100' equals to 1 which meets my expectation. I think the optimum number will be 2 in this case.

Problem 3: 6-bit Grover's Algorithm (for EE 521 only)

We define eight phone number $p_1 = 000, p_1 = 001, p_3 = 010, \ldots p_8 = 111$ and the names corresponding to the phone numbers are $n_1 = 000, n_1 = 001, n_3 = 010, \ldots n_8 = 111$ respectively. Recall that in Grover's algorithm, the phone numbers are stored in three qubits and the name are stored in three other qubits. In terms of qubit states, the phone numbers are $|p_1\rangle = |000\rangle, |p_2\rangle = |001\rangle, |p_3\rangle = |010\rangle, \ldots |p_8\rangle = |111\rangle$ and the corresponding names stored on three other qubits are $|p_1\rangle = |000\rangle, |p_2\rangle = |001\rangle, |p_3\rangle = |010\rangle, \ldots |p_8\rangle = |111\rangle$. The phone numbers are identical to the names which are stored on three other qubits. Starting out with the state $|000000\rangle = |000\rangle|000\rangle$, find the gate operations to reach the following states in the Grover's algorithm sequentially:

(a) $|init\rangle = \frac{1}{\sqrt{8}}[|000\rangle|p_1\rangle + |001\rangle|p_2\rangle + |010\rangle|p_3\rangle + \ldots + |111\rangle|p_8\rangle]$

(b) $\hat{V}|init\rangle = -\frac{1}{\sqrt{8}}|110, p_7\rangle + \frac{\sqrt{7}}{\sqrt{8}}[\frac{1}{\sqrt{7}}(|000, p_1\rangle + |001, p_2\rangle + |010, p_3\rangle + \ldots)]$

(c) $\hat{W}\hat{V}|init\rangle = \frac{5}{4\sqrt{2}}|110\rangle|p_7\rangle + \frac{\sqrt{7}}{4\sqrt{2}}[\frac{1}{\sqrt{7}}(|000, p_1\rangle + |001, p_2\rangle + |010, p_3\rangle + \ldots)]$

(d) repeat above operations

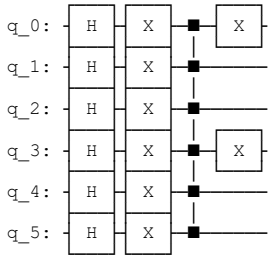
Implement the above gates in qiskit and confirm that your realization is correct. What is the optimum number of rotations for this search problem in term of operations by $\hat{W}\hat{V}$?

Hint: You can refer to Lecture note 7.5.2 section

In [269]:
```
n=6
gc = QuantumRegister(n, 'q')
circ = QuantumCircuit(gc)

circ.h(gc[0:n])
circ.x(gc[0:n])
circ += MCMT('z',5,1, label=None)
circ.x(gc[0])
circ.x(gc[3])
circ.draw()
```

```
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\3682985156.py:7: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potential
ly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
```

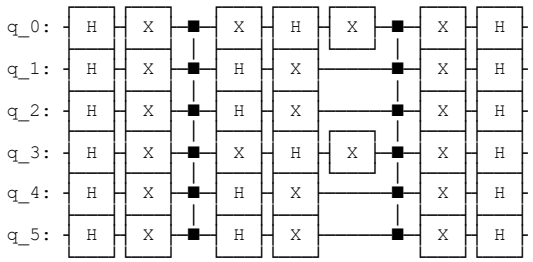
Out[269]:



In [270]:
```
circ.h(gc[0:n])
circ.x(gc[0:n])
circ += MCMT('z',5,1, label=None)
circ.x(gc[0:n])
circ.h(gc[0:n])
circ.draw()
```

```
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\4048097771.py:3: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potential
ly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
```

Out[270]:



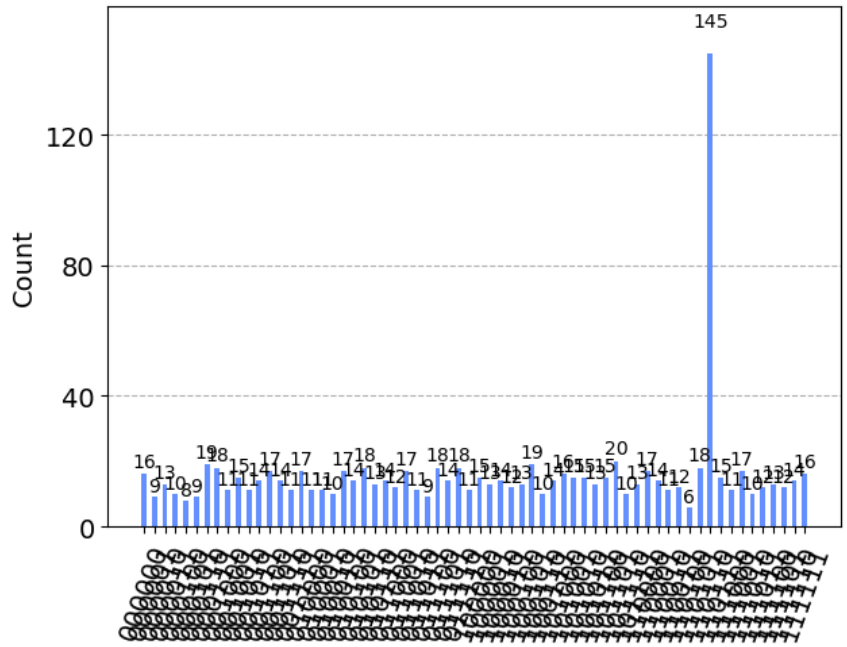
In [271]:
```
backend = BasicAer.get_backend('statevector_simulator')
job = execute(circ, backend)
result = job.result()
gcstate = result.get_statevector(circ)
print('Statevector:', gcstate)
```

```
Statevector: [-0.1171875-3.32294887e-16j -0.1171875-3.53263551e-16j
 -0.1171875-3.57812351e-16j -0.1171875-3.83833851e-16j
 -0.1171875-3.54455961e-16j -0.1171875-3.69630194e-16j
 -0.1171875-3.71518501e-16j -0.1171875-3.94613036e-16j
 -0.1171875-3.40646632e-16j -0.1171875-3.49672916e-16j
 -0.1171875-3.78749472e-16j -0.1171875-3.99507994e-16j
 -0.1171875-3.62443665e-16j -0.1171875-3.78131743e-16j
 -0.1171875-3.85545547e-16j -0.1171875-3.99042242e-16j
 -0.1171875-3.23644144e-16j -0.1171875-3.44930937e-16j
 -0.1171875-3.56576503e-16j -0.1171875-3.79402791e-16j
 -0.1171875-3.48613474e-16j -0.1171875-3.71791570e-16j
 -0.1171875-6.08414022e-16j -0.1171875-4.17136112e-16j
 -0.1171875-3.33542563e-16j -0.1171875-3.44772304e-16j
 -0.1171875-3.68519611e-16j -0.1171875-3.97835511e-16j
 -0.1171875-3.65982975e-16j -0.1171875-3.83127668e-16j
 -0.1171875-3.86648601e-16j -0.1171875-4.04443349e-16j
 -0.1171875-3.36588736e-16j -0.1171875-3.43558533e-16j
 -0.1171875-3.65019813e-16j -0.1171875-3.77000889e-16j
 -0.1171875-3.60397834e-16j -0.1171875-3.71427339e-16j
 -0.1171875-4.10586774e-16j -0.1171875-3.90215320e-16j
 -0.1171875-3.57254760e-16j -0.1171875-3.52825869e-16j
 -0.1171875-3.92462547e-16j -0.1171875-3.96213618e-16j
 -0.1171875-3.80560537e-16j -0.1171875-3.83347640e-16j
 -0.1171875-4.02258409e-16j -0.1171875-4.07843008e-16j
 -0.1171875-2.82770510e-16j -0.1171875-2.68643344e-16j
 -0.1171875-3.31244674e-16j -0.1171875-3.53992891e-16j
 -0.1171875-3.24002140e-16j -0.1171875-1.96398433e-16j
 -0.3671875-9.74499366e-16j -0.1171875-4.01194288e-16j
 -0.1171875-3.29961630e-16j -0.1171875-3.22734741e-16j
 -0.1171875-3.62270190e-16j -0.1171875-3.82797167e-16j
 -0.1171875-3.59537583e-16j -0.1171875-3.68079785e-16j
 -0.1171875-3.94931611e-16j -0.1171875-3.80910070e-16j]
```

In [272]:
```
c1 = ClassicalRegister(n, 'c1')
circ.add_register(c1)
circ.measure(gc[0], c1[0])
circ.measure(gc[1], c1[1])
circ.measure(gc[2], c1[2])
circ.measure(gc[3], c1[3])
circ.measure(gc[4], c1[4])
circ.measure(gc[5], c1[5])
circ.draw()

backend_gc = BasicAer.get_backend('qasm_simulator')
job_gc = execute(circ, backend_gc, shots=1000)
result_gc = job_gc.result()
counts_gc = result_gc.get_counts()
print(counts_gc)
plot_histogram(counts_gc)
```

```
{'110000': 17, '110111': 15, '110110': 145, '111001': 17, '001100': 17, '001101': 14, '101000': 16, '000000': 16, '101110': 10, '100101': 19, '111000': 11, '110011': 12, '0
00101': 9, '000011': 10, '111100': 13, '100011': 12, '111010': 10, '100001': 13, '000110': 19, '001000': 11, '111011': 12, '111110': 14, '100110': 10, '101001': 15, '10111
1': 13, '001011': 14, '001111': 17, '011000': 12, '010101': 18, '101010': 15, '011111': 11, '111101': 12, '011100': 18, '011110': 18, '110101': 18, '111111': 16, '000111':
18, '100100': 13, '011101': 14, '001001': 15, '010000': 11, '010111': 14, '100010': 14, '101100': 15, '101011': 13, '011011': 9, '000001': 9, '010100': 14, '011001': 17, '0
11010': 11, '100000': 15, '110010': 11, '010011': 17, '001110': 11, '100111': 14, '101101': 20, '000010': 13, '001010': 11, '110100': 6, '110001': 14, '000100': 8, '01011
0': 13, '010010': 10, '010001': 11}
```

Out[272]:



Above is the result after one operation, the probablity is around $0.367^2 = 0.134$. Then I will repeat this for multiple of times.

In [273]:
```
n=6
gc = QuantumRegister(n, 'q')
circ = QuantumCircuit(gc)

T = 2 # number of steps to take

for t in range(T):
    circ.h(gc[0:n])
    circ.x(gc[0:n])
    circ += MCMT('z',5,1, label=None)
    circ.x(gc[0])
    circ.x(gc[3])
    circ.h(gc[0:n])
    circ.x(gc[0:n])
    circ += MCMT('z',5,1, label=None)
    circ.x(gc[0:n])
    circ.h(gc[0:n])
```

```
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\1329721474.py:10: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potentia
lly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\1329721474.py:15: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potentia
lly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
```

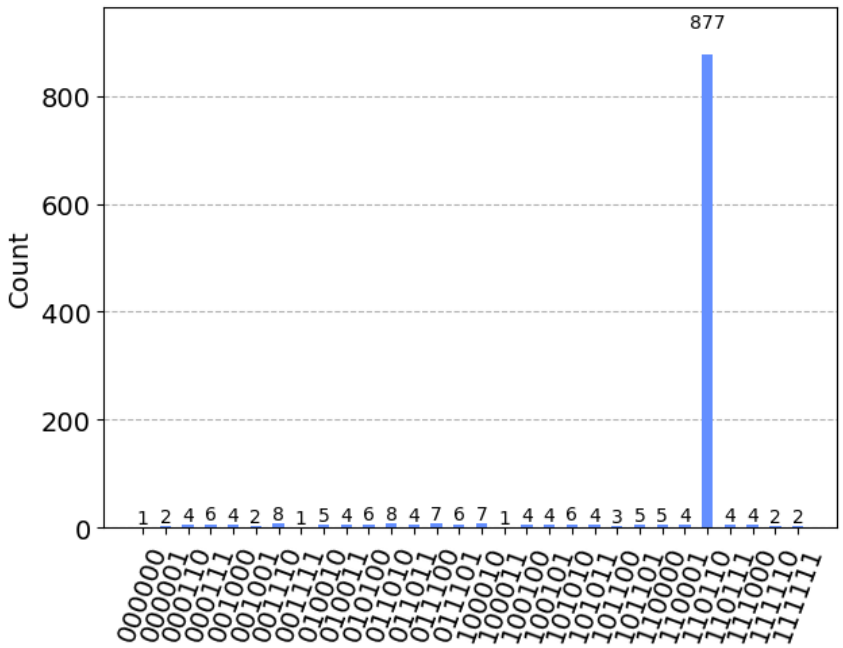
In [274]:
```
backend = BasicAer.get_backend('statevector_simulator')
job = execute(circ, backend)
result = job.result()
gcstate = result.get_statevector(circ)
print('Statevector:', gcstate)
```

```
Statevector: [-0.0625-2.11428595e-16j -0.0625-1.72203802e-16j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
 -0.0625-2.00290925e-16j -0.0625-2.24107474e-16j -0.0625-2.12058007e-16j
 -0.0625-1.64635497e-16j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j -0.0625-2.20728466e-16j
 -0.0625-2.18415082e-16j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
 -0.0625-1.95486506e-16j -0.0625-2.32381340e-16j -0.0625-2.14042926e-16j
 -0.0625-1.93875600e-16j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j -0.0625-2.20627435e-16j
 -0.0625-2.35863349e-16j -0.0625-2.17051415e-16j -0.0625-1.83928218e-16j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j -0.0625-2.93112543e-16j -0.0625-3.32209353e-16j
 -0.0625-3.37307393e-16j -0.0625-2.61993651e-16j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
 -0.0625-2.99754011e-16j -0.0625-3.22098438e-16j -0.0625-3.50517311e-16j
 -0.0625-2.72202500e-16j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
 -0.0625-2.84350851e-16j -0.0625-2.84592532e-16j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
  0.9375+4.93321261e-15j -0.0625-2.32083063e-16j -0.0625-3.37426660e-16j
 -0.0625-2.95701491e-16j  0.    +0.00000000e+00j  0.    +0.00000000e+00j
  0.    +0.00000000e+00j  0.    +0.00000000e+00j -0.0625-1.98626032e-16j
 -0.0625-3.14806200e-16j]
```

In [275]:
```
c1 = ClassicalRegister(n, 'c1')
circ.add_register(c1)
circ.measure(gc[0], c1[0])
circ.measure(gc[1], c1[1])
circ.measure(gc[2], c1[2])
circ.measure(gc[3], c1[3])
circ.measure(gc[4], c1[4])
circ.measure(gc[5], c1[5])
circ.draw()

backend_gc = BasicAer.get_backend('qasm_simulator')
job_gc = execute(circ, backend_gc, shots=1000)
result_gc = job_gc.result()
counts_gc = result_gc.get_counts()
print(counts_gc)
plot_histogram(counts_gc)
```

```
{'110110': 877, '100010': 7, '111111': 2, '010010': 5, '011101': 6, '000111': 6, '110111': 4, '001110': 8, '001111': 1, '000110': 4, '010011': 4, '111000': 4, '100100': 4,
'101010': 6, '011100': 7, '011010': 8, '110000': 5, '001000': 4, '000001': 2, '010100': 6, '011011': 4, '000000': 1, '101100': 3, '101011': 4, '100101': 4, '101101': 5, '00
1001': 2, '110001': 4, '111110': 2, '100011': 1}
```

Out[275]:



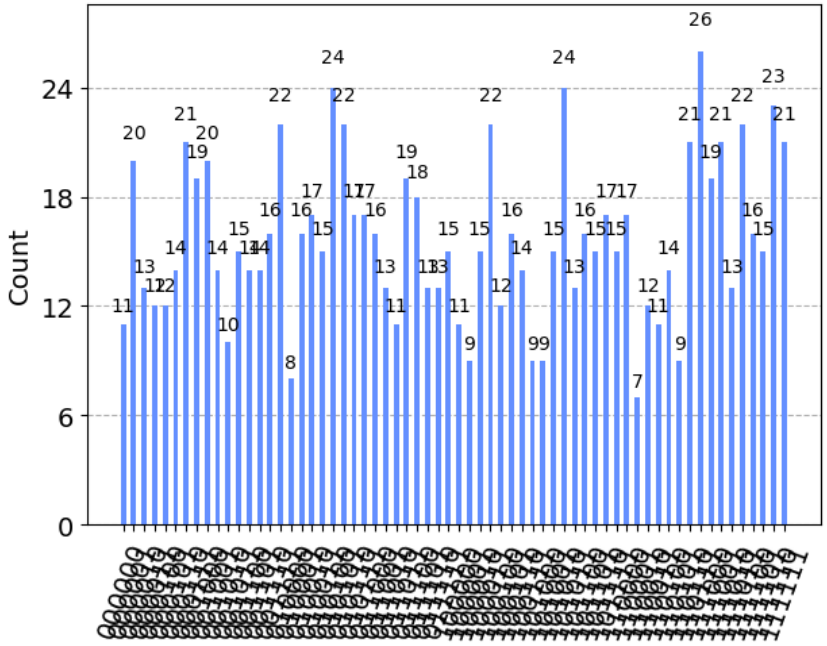
Above is the result after two operations, the probablity is around $0.9375^2 = 0.879$. Then I will repeat this for multiple of times.

In [276]:
```
n=6
gc = QuantumRegister(n, 'q')
circ = QuantumCircuit(gc)

T = 3 # number of steps to take

for t in range(T):
    circ.h(gc[0:n])
    circ.x(gc[0:n])
    circ += MCMT('z',5,1, label=None)
    circ.x(gc[0])
    circ.x(gc[3])
    circ.h(gc[0:n])
    circ.x(gc[0:n])
    circ += MCMT('z',5,1, label=None)
    circ.x(gc[0:n])
    circ.h(gc[0:n])

backend = BasicAer.get_backend('statevector_simulator')
job = execute(circ, backend)
result = job.result()
gcstate = result.get_statevector(circ)
print('Statevector:', gcstate)


c1 = ClassicalRegister(n, 'c1')
circ.add_register(c1)
circ.measure(gc[0], c1[0])
circ.measure(gc[1], c1[1])
circ.measure(gc[2], c1[2])
circ.measure(gc[3], c1[3])
circ.measure(gc[4], c1[4])
circ.measure(gc[5], c1[5])
circ.draw()

backend_gc = BasicAer.get_backend('qasm_simulator')
job_gc = execute(circ, backend_gc, shots=1000)
result_gc = job_gc.result()
counts_gc = result_gc.get_counts()
print(counts_gc)
plot_histogram(counts_gc)
```

```
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\1513814759.py:10: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potentia
lly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
C:\Users\lixic\AppData\Local\Temp\ipykernel_4904\1513814759.py:15: DeprecationWarning: The QuantumCircuit.__iadd__() method is being deprecated. Use the compose() (potentia
lly with the inplace=True argument) and tensor() methods which are more flexible w.r.t circuit register compatibility.
  circ += MCMT('z',5,1, label=None)
```

```
Statevector: [-0.1171875+1.14626429e-16j  0.1328125+6.29149549e-16j
 -0.1171875-6.40019904e-16j -0.1171875-7.67432152e-16j
 -0.1171875-7.34324030e-16j -0.1171875-6.68357536e-16j
  0.1328125+6.71025475e-16j  0.1328125+7.12167027e-16j
  0.1328125+6.61964187e-16j  0.1328125+6.99761112e-16j
 -0.1171875-6.17783171e-16j -0.1171875-6.23569112e-16j
 -0.1171875-6.64405669e-16j -0.1171875-6.52623926e-16j
  0.1328125+6.43263808e-16j  0.1328125+6.58830371e-16j
 -0.1171875-6.94381198e-16j -0.1171875-6.80761633e-16j
  0.1328125+6.45908201e-16j  0.1328125+6.54668095e-16j
  0.1328125+6.98609038e-16j  0.1328125+7.18844253e-16j
 -0.1171875-2.25103309e-16j -0.1171875-6.56944374e-16j
 -0.1171875-6.69807268e-16j -0.1171875-6.54161221e-16j
  0.1328125+6.57385430e-16j  0.1328125+6.65308590e-16j
  0.1328125+7.00580459e-16j  0.1328125+6.93454064e-16j
 -0.1171875-6.64485291e-16j -0.1171875-6.56230591e-16j
 -0.1171875-5.23223858e-16j -0.1171875-6.27047579e-16j
  0.1328125+6.16105186e-16j  0.1328125+6.40456118e-16j
  0.1328125+6.44734852e-16j  0.1328125+6.44352516e-16j
 -0.1171875-6.37237679e-16j -0.1171875-6.32817264e-16j
 -0.1171875-6.39819221e-16j -0.1171875-6.51702925e-16j
  0.1328125+5.54702946e-16j  0.1328125+6.38974631e-16j
  0.1328125+6.55297003e-16j  0.1328125+6.56992292e-16j
 -0.1171875-6.54127157e-16j -0.1171875-6.62331638e-16j
  0.1328125+6.23679431e-16j  0.1328125+6.16618840e-16j
 -0.1171875-5.62654365e-16j -0.1171875-5.74911072e-16j
 -0.1171875-6.59697725e-16j -0.1171875-6.73240944e-16j
  0.1328125+1.03459743e-15j  0.1328125+6.01651475e-16j
  0.1328125+6.49124123e-16j  0.1328125+6.56226278e-16j
 -0.1171875-5.73119604e-16j -0.1171875-6.25594836e-16j
 -0.1171875-6.55263874e-16j -0.1171875-6.73337986e-16j
  0.1328125+6.41673612e-16j  0.1328125+6.80807183e-16j]
{'010110': 17, '010100': 24, '000101': 14, '011011': 19, '100001': 9, '111011': 22, '010000': 8, '100101': 16, '011101': 13, '100100': 12, '101010': 24, '111111': 21, '0100
11': 15, '001101': 14, '111110': 23, '000001': 20, '100011': 22, '011111': 15, '101100': 16, '000110': 21, '111000': 19, '110111': 26, '001011': 15, '010111': 17, '111010':
13, '110110': 21, '101001': 15, '000000': 11, '111001': 21, '101111': 15, '110000': 17, '111101': 15, '010101': 22, '000011': 12, '111100': 16, '101110': 17, '001110': 16,
'110010': 12, '000111': 19, '110100': 14, '001000': 20, '011100': 18, '001001': 14, '000100': 12, '011000': 16, '100111': 9, '110011': 11, '001100': 14, '100000': 11, '0011
11': 22, '110001': 7, '110101': 9, '011010': 11, '100110': 14, '001010': 10, '101101': 15, '010010': 17, '100010': 15, '011110': 13, '010001': 16, '101011': 13, '011001': 1
3, '101000': 9, '000010': 13}
```

Out[276]:



Above is the result after two operations, the probablity is around $0.133^2 = 0.017$. Then I will repeat this for multiple of times. Thus the optimum number of operation is **two**.

Should we alert the press?

Quantum algorithms can provide an exponential speedup, compared to any known classical algorithm, for some problems. The problem we investigated here, 1-IN-3-SAT, is an $\mathcal{NP}$-complete problem, which means if we can efficiently solve it, then we can efficiently solve any $\mathcal{NP}$ problem (which includes a lot of very important, very interesting problems).

We don't know of any classical algorithm that can solve the 1-IN-3-SAT problem in faster than in exponential time ($\mathcal{O}(K^N)$) where N is the number of variables.

Meanwhile, Grover's algorithm lets us search a space of size N samples in on the order of $\mathcal{O}(\sqrt{N})$ time. This means we can run Grover's algorithm in polynomial time (with respect to N).

This begs the question: Did we just find an algorithm that can solve an $\mathcal{NP}$-complete problem in polynomial time? If so, we have to alert the press right away - we just made history!!

Ignore this: Fast, but is it fast enough?

Just what kind of speedup did applying Grover's algorithm give us (assuming we ran our circuit on a real quantum computer rather than just a simulator)? Did we find a polynomial time algorithm to solve 1-IN-3-SAT?

Hint: Remind yourself what the definitions of N is for each of the running times above, how do they relate?

End of Chapter Problems

Chapter-9 Problem-3:

Consider a two-level system with energy level spacing equal to $2eV$. At time $t = 0$, the electron is in state 1. Plot the probablity to find the electron as a function of time when the frequency of the applied ac potential is (a) $2.05eV$, (b) $2eV$ and (c) $1.95eV$. Assume that the matrix element $u_{12} = 6.5 * 10^{-6} meV$

Answer:

The expressions for $P_1(t)$ and $P_2(t)$ are:

$$P_1(t) = cos^2(\Omega' t) + \frac{\Delta^2}{4\Omega'^2} sin^2(\Omega' t)$$

$$P_2(t) = \frac{\Omega^2}{\Omega'^2} sin^2(\Omega' t)$$

where

$$\Omega' = \sqrt{\Omega^2 + \frac{\Delta^2}{4}}$$

$$\Omega = \frac{|u_{21}|}{\hbar}$$

$$\Delta = \omega - \omega_{21}$$

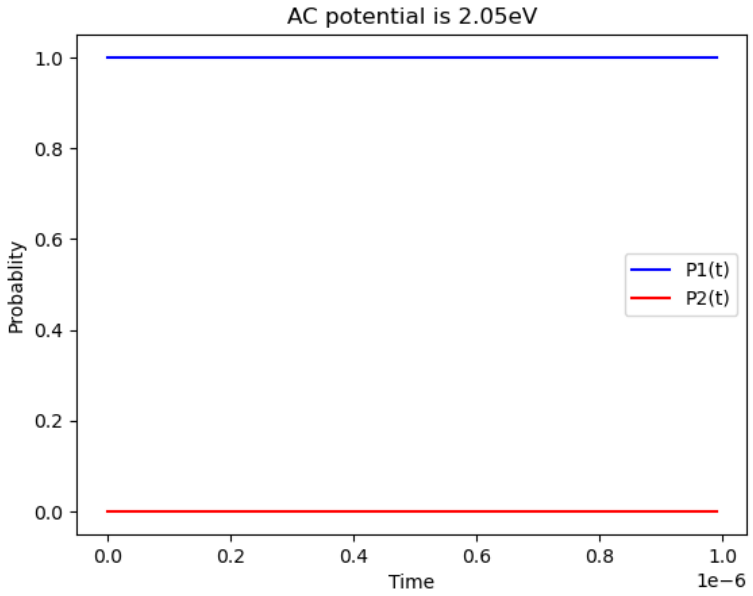
Also we have $\hbar\omega_{21} = 2eV$ and $\hbar\omega = 2.05, 2, 1.95eV$ for part (a), (b), (c), respectively.

Part (a) Applied AC poetntial is $2.05eV$

In [277]:
```
eta = 6.63e-34/2/np.pi #Reduced Plank constant in eV.s
q = 1.6e-19
u21=6.5e-6*1e-3*q
w21=2*q/eta
w=2.05*q/eta
delta=w-w21
omega=abs(u21)/eta
omegap=np.sqrt(omega**2+delta**2/4)

Tend=1e-6
dt=1e-8
t = np.arange(0,Tend,dt)
P1=(np.cos(omegap*t))**2 + delta**2/4/omegap**2*(np.sin(omegap*t))**2
P2=omega**2/omegap**2*(np.sin(omegap*t))**2

print(omega)
print(omegap)

plt.figure()
plt.plot(t, P1, color='b', ls='-')
plt.plot(t, P2, color='r', ls='-')
#plt.plot(t, P1+P2, color='g', ls='-')
plt.legend(['P1(t)', 'P2(t)','P1(t)+P2(t)'])
plt.title('AC potential is 2.05eV')
plt.xlabel('Time')
plt.ylabel('Probablity')
```

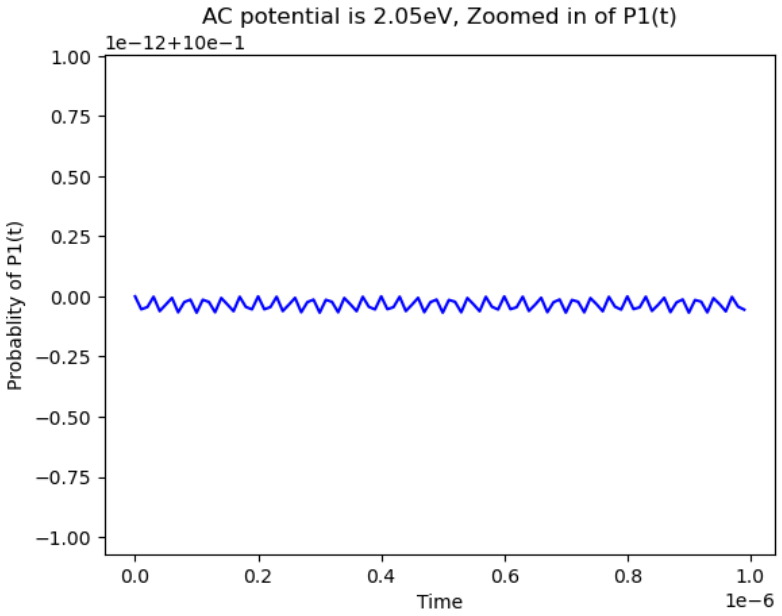
```
9855976.952438565
37907603663226.53
```

Out[277]: Text(0, 0.5, 'Probablity')



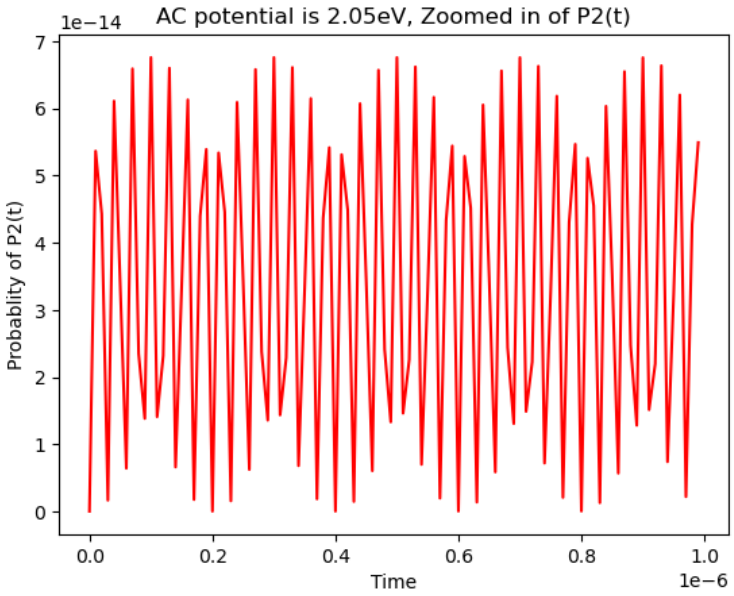
In [278]:
```
plt.figure()
plt.plot(t, P1, color='b', ls='-')
#plt.plot(t, P1+P2, color='g', ls='-')
plt.title('AC potential is 2.05eV, Zoomed in of P1(t)')
plt.xlabel('Time')
plt.ylabel('Probablity of P1(t)')
```

Out[278]:
```
Text(0, 0.5, 'Probablity of P1(t)')
```



In [279]:
```
plt.figure()
plt.plot(t, P2, color='r', ls='-')
#plt.plot(t, P1+P2, color='g', ls='-')
plt.title('AC potential is 2.05eV, Zoomed in of P2(t)')
plt.xlabel('Time')
plt.ylabel('Probablity of P2(t)')
```

Out[279]:
```
Text(0, 0.5, 'Probablity of P2(t)')
```



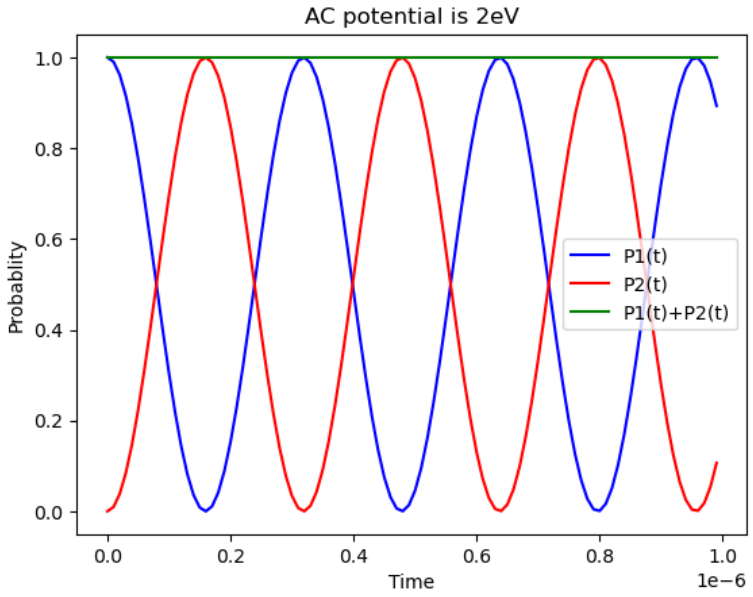
The probablity of p1 is almost 1 and the probability of p2 is almost 0.

Part (b) Input AC poetntial is $2eV$

In [280]:
```
w=2*q/eta
delta=w-w21
omega=abs(u21)/eta
omegap=np.sqrt(omega**2+delta**2/4)

Tend=1e-6
dt=1e-8
t = np.arange(0,Tend,dt)
P1=(np.cos(omegap*t))**2 + delta**2/4/omegap**2*(np.sin(omegap*t))**2
P2=omega**2/omegap**2*(np.sin(omegap*t))**2

print(delta)
print(omega)
print(omegap)

plt.figure()
plt.plot(t, P1, color='b', ls='-')
plt.plot(t, P2, color='r', ls='-')
plt.plot(t, P1+P2, color='g', ls='-')
plt.legend(['P1(t)', 'P2(t)','P1(t)+P2(t)'])
plt.title('AC potential is 2eV')
plt.xlabel('Time')
plt.ylabel('Probablity')
```

```
0.0
9855976.952438565
9855976.952438565
```

Out[280]: Text(0, 0.5, 'Probablity')

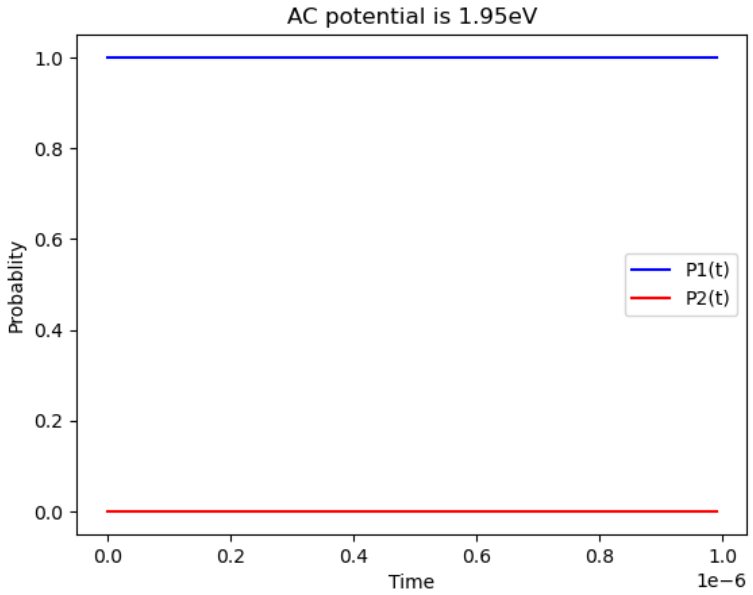


Part (c) Input AC poetntial is $1.95eV$

In [281]:
```
w=1.95*q/eta
delta=w-w21
omega=abs(u21)/eta
omegap=np.sqrt(omega**2+delta**2/4)

Tend=1e-6
dt=1e-8
t = np.arange(0,Tend,dt)
P1=(np.cos(omegap**2*t))**2 + delta**2/4/omegap**2*(np.sin(omegap**2*t))**2
P2=omega**2/omegap**2*(np.sin(omegap**2*t))**2

print(delta)
print(omega)
print(omegap)

plt.figure()
plt.plot(t, P1, color='b', ls='-')
plt.plot(t, P2, color='r', ls='-')
#plt.plot(t, P1+P2, color='g', ls='-')
plt.legend(['P1(t)', 'P2(t)','P1(t)+P2(t)'])
plt.title('AC potential is 1.95eV')
plt.xlabel('Time')
plt.ylabel('Probablity')
```

```
-75815207326451.0
9855976.952438565
37907603663226.78
```

Out[281]: Text(0, 0.5, 'Probablity')



The probablity of p1 is almost 1 and the probability of p2 is almost 0.

In [ ]:

In [ ]: