Implementation of the solution of Binary Sudoku using Grover's Algorithm

Author
Ahmed Saad El Fiky

QCIC 761 Basics of Computer Programming

Professional Master in QCIC

May 2023

Abstract

Grover's algorithm enables one to find (with probability $> 1/2$) a specific item within a randomly ordered database of N items using $O(\sqrt{N})$ operations. By contrast, a classical computer would require $O(N)$ operations to achieve this. Therefore, Grover's algorithm provides a quadratic speedup over an optimal classical algorithm. It has also been shown that Grover's algorithm is optimal in the sense that no quantum Turing machine can do this in less than $O(\sqrt{N})$ operations.

Our work here is to implement the solution of Binary Sudoku using Grover's Search Algorithm on IBM Quantum Simulator and Actual device.

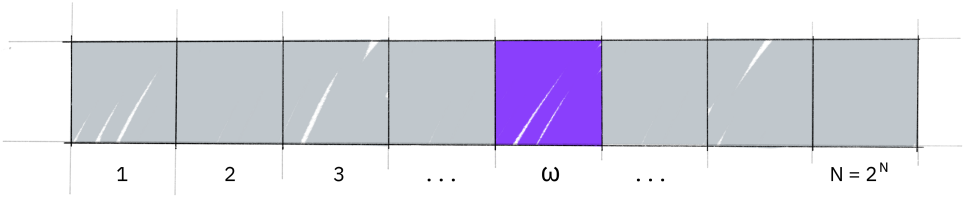
Contents

---

1. Introduction

Grover's algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms. This is called the amplitude amplification trick.

Suppose you are given a large list of N items. Among these items there is one item with a unique property that we wish to locate; we will call this one the winner w. Think of each item in the list as a box of a particular color. Say all items in the list are gray except the winner w, which is purple:



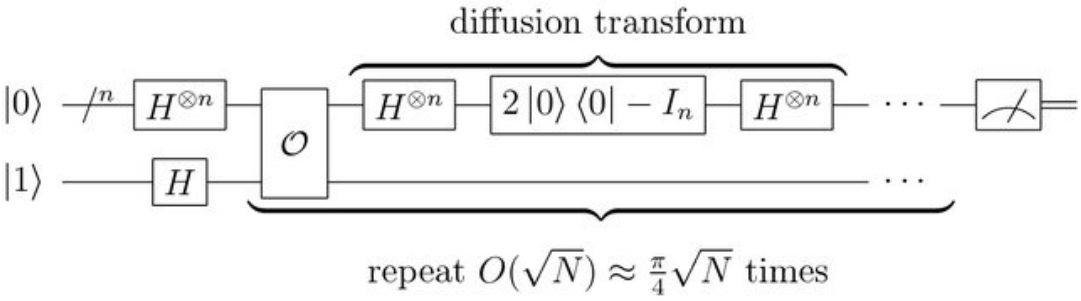
To find the purple box -- *the marked item* -- using classical computation, one would have to check on average $N/2$ of these boxes, and in the worst case, all N of them.
On a quantum computer, however, we can find the marked item in roughly steps with Grover's amplitude amplification trick. A quadratic speedup is indeed a substantial time-saver for finding marked items in long lists. Additionally, the algorithm does not use the list's internal structure, which makes it generic; this is why it immediately provides a quadratic quantum speed-up for many classical problems.

---

2. Algorithm Overview

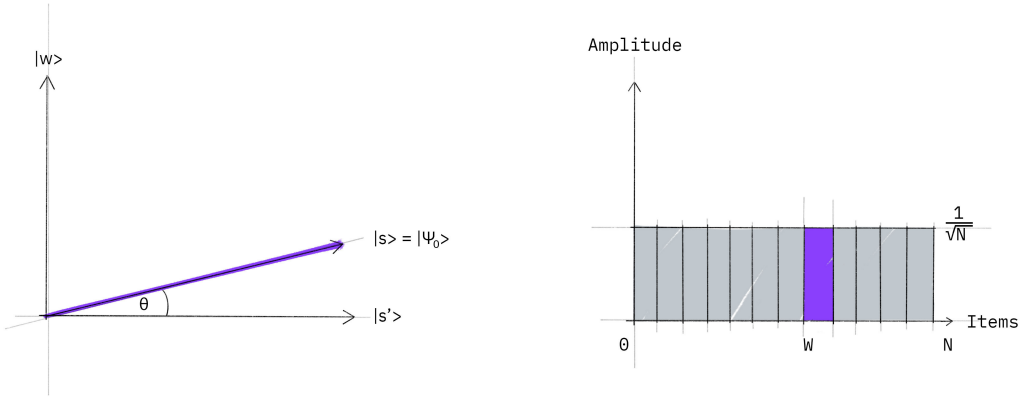
Grover's algorithm consists of three main algorithms steps: state preparation, the oracle, and the diffusion operator.

- The preparation state is where we create the search space, which is all possible cases the answer could take.
- The oracle is what marks the correct answer, or answers we are looking for.
- The diffusion operator magnifies these answers so they can stand out and be measured at the end of the algorithm.



Step 1:

The amplitude amplification procedure starts out in the uniform superposition $|s>$, which is easily constructed from $|s>= H^{\otimes n}|0>$.

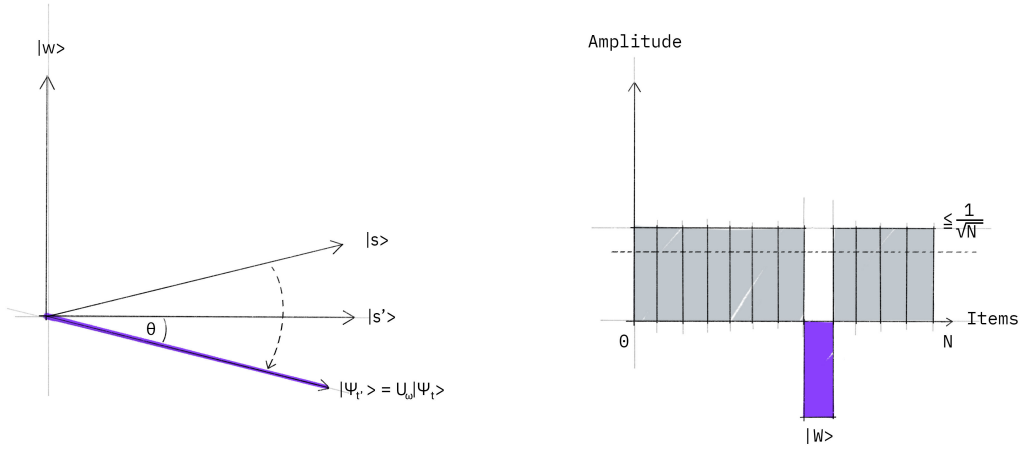


The left graphic corresponds to the two-dimensional plane spanned by perpendicular vectors $|w>$ and $|s^{'}>$ which allows to express the initial state as $|s>=sin\theta|w>+cos\theta|s^{'}>$, where $\theta = arcsin<s|w>=arcsin\frac{1}{\sqrt{N}}$.

The right graphic is a bar graph of the amplitudes of the state .

Step 2:

We apply the **Oracle** reflection $O \equiv U_f$ to the state $|s>$.



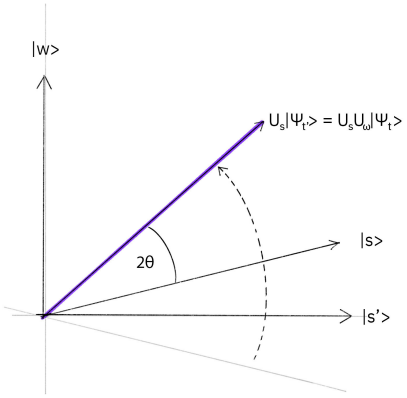
Geometrically this corresponds to a reflection of the state $|s>$ about $|s^{'}>$. This transformation means that the amplitude in front of the $|w>$ state becomes **negative**, which in turn means that the average amplitude (*indicated by a dashed line*) has been lowered.

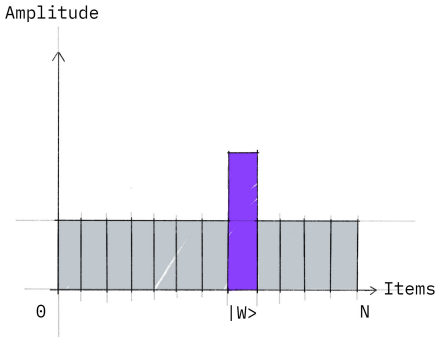
Step 3:

We now apply a **Diffusion Operator** $D \equiv (U_s)$, an additional reflection about the state $|s>$:

$$|U_s>=2|s><s|-\mathbb{1}.$$

This transformation maps the state to $U_s U_f |s>$ and completes the transformation.





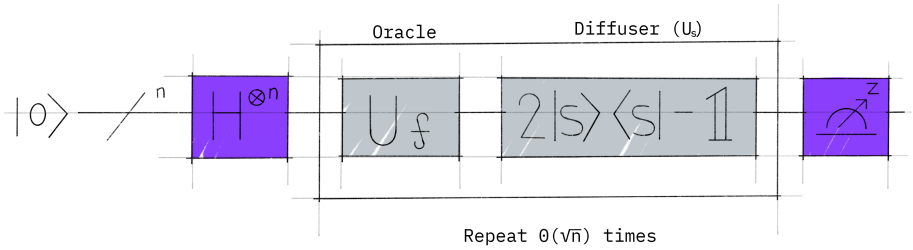
Two reflections always correspond to a rotation. The transformation $U_s U_f$ rotates the initial state $|s>$ closer towards the winner $|w>$. The action of the reflection U_s in the amplitude bar diagram can be understood as a reflection about the average amplitude. Since the average amplitude has been lowered by the first reflection, this transformation boosts the negative amplitude of $|w>$ to roughly three times its original value, while it decreases the other amplitudes.

After t steps we will be in the state $|\psi_t>$ where: $|\psi_t>= (U_s U_f)^t |s>$

To calculate the number of rotations we need to know the size of the search space and the number of answers we are looking for. The get the optimal number of iterations t, we can follow the equation:

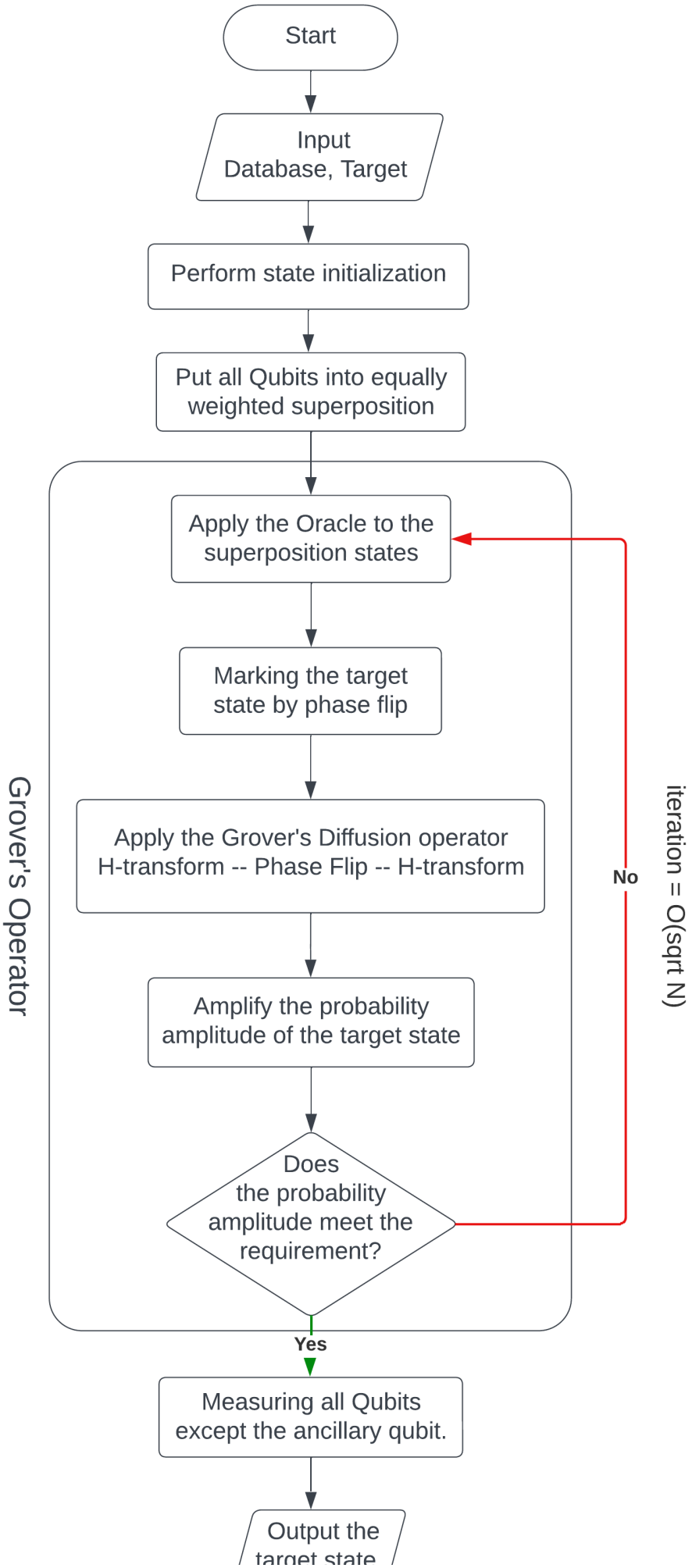
$$t = [\frac{\pi}{4}\sqrt{\frac{N}{m}}]$$

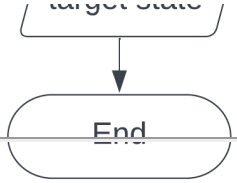
Where N is the size of the search space and m is the number of answers we want.



---

3. Flowchart of Grover's Algorithm


Start
Input
Database, Target
Perform state initialization
Put all Qubits into equally weighted superposition
Grover's Operator
Apply the Oracle to the superposition states
Marking the target state by phase flip
Apply the Grover's Diffusion operator
H-transform -- Phase Flip -- H-transform
Amplify the probability amplitude of the target state
Does the probability amplitude meet the requirement?
No
iteration = O(sqrt N)
Yes
Measuring all Qubits except the ancillary qubit.
Output the target state



4. Grover's Search Algorithm Pseudocode

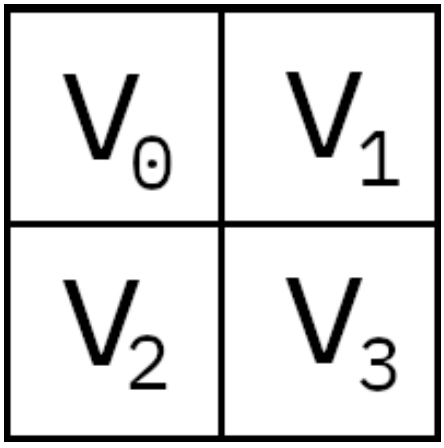
1. Input the Database and target.
2. Create the circuit and perform State initialization.
3. Put all Qubits into superposition using a Hadamard gate.
4. Implement an Oracle that will mark the state you wish to find by phase flip.
5. Implement an amplification circuit (Diffusion Operator) that further increases the marked states amplitude, while decreasing the amplitude of all other states.
6. Repeat the previous 2 steps $O(\sqrt{N})$ times.
7. Measure all qubits except the ancillary qubit.
8. Output the target state.

---

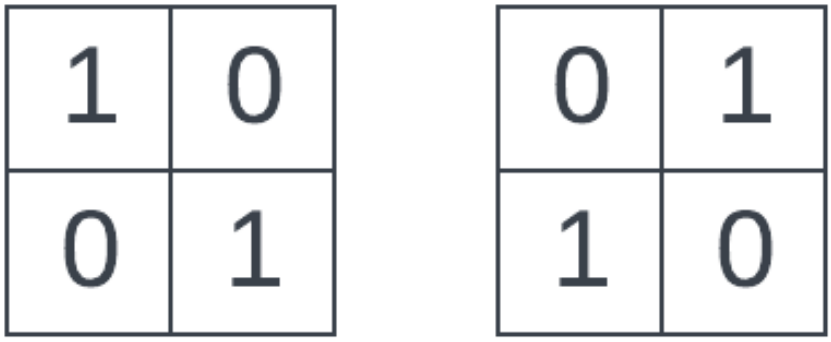
5. Solving Sudoku using Grover's Algorithm

Our problem is a 2×2 binary sudoku, which in our case has two simple rules:

- No column may contain the same value twice.
- No row may contain the same value twice.



There are two possible solutions for this problem: 1001 OR 0110



We will design an Oracle to flip the phase of all the states that correspond to a solution, |1001⟩ and |0110⟩ to -1.

We will use **4** qubits, one for each square, and this will give us $N = 16$ as input states to the circuit, where: $N = 2^n$, n is the number of qubits.

---

6. Circuit implemntation using Qiskit on IBM Quantum Lab

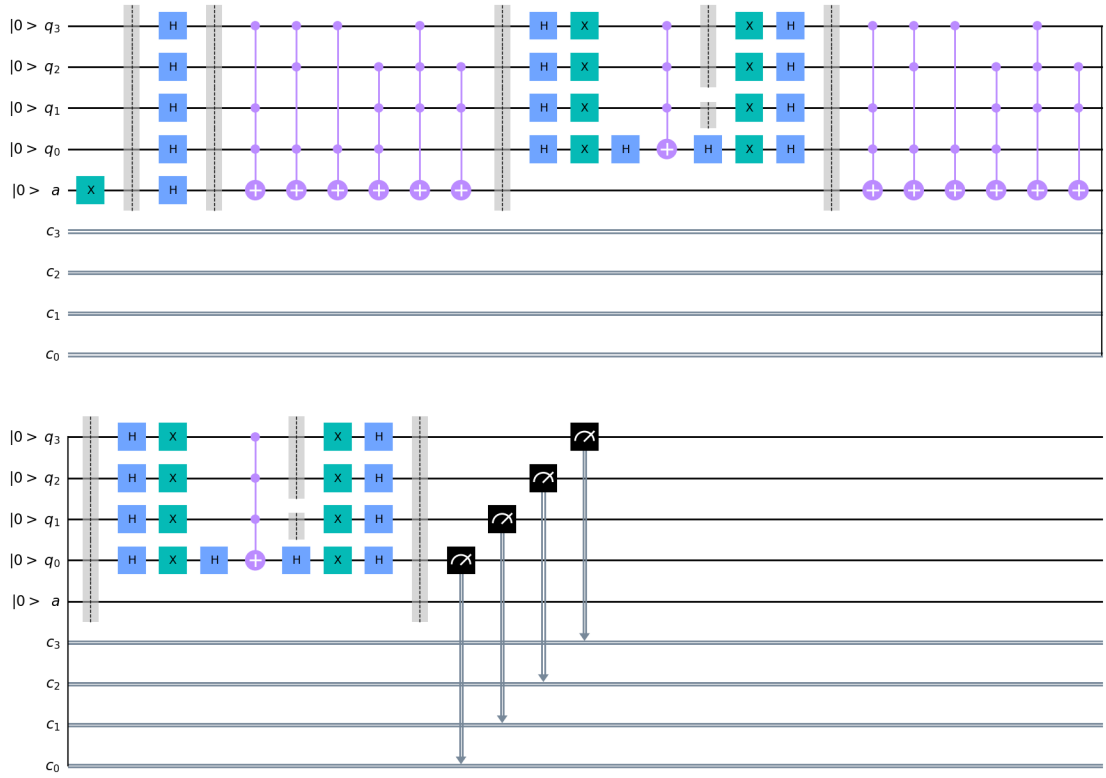
In [1]:
```
# initializing
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import datetime
import math
import os

# assign pi
pi = math.pi

# importing Qiskit
from qiskit import IBMQ, Aer, transpile, execute, assemble
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.circuit.library import MCXGate
import qiskit.providers.ibmq

from qiskit_ibm_provider import IBMProvider
from qiskit_ibm_runtime import Estimator, Session, QiskitRuntimeService,
from qiskit.quantum_info import SparsePauliOp

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

```
service = QiskitRuntimeService()
```

In [2]:

```
# The problem to find the solution of binary Sudoku
# We previously know that the winner states are 1001 and 0110
# So we will implement an oracle to phase kickback these to states

# create the registers and circuit
n = 4    # no. of qubits used
q = QuantumRegister(n, '|0> q')
a = QuantumRegister(1, '|0>  a') # ancilla qubit
c = ClassicalRegister(n, 'c')
qc = QuantumCircuit(a, q, c)

# define a function to apply H-gate on a given list of qubits:
def Htransform(qc, qubits):
    for q in qubits:
        qc.h(q)
    return qc

# define a function to apply X-gate on a given list of qubits:
def xgate(qc, qubits):
    for q in qubits:
        qc.x(q)
    return qc

# assign the ancilla qubit to be in state |1>
qc.x(a)
qc.barrier()

# initializing the superpositioning over all qubits
Htransform(qc, [q, a])
qc.barrier()


# Iteration loop
# No. of iteration = pi/4*sqrt(2**n/no. of solutions)
# Here in Binary Sudoku problem we have 2 solutions 1001 & 0110 and no.

i = 1
while i <= (pi/4)*math.sqrt(2**n/2):
    # Oracle
    qc.append(MCXGate(3), [q[0], q[1], q[3], a])
    qc.append(MCXGate(3), [q[0], q[2], q[3], a])
    qc.ccx(q[0], q[3], a)
    qc.append(MCXGate(3), [q[0], q[1], q[2], a])
    qc.append(MCXGate(3), [q[1], q[2], q[3], a])
    qc.ccx(q[1], q[2], a)

    qc.barrier()

    # Grover's Diffusion Operator
    Htransform(qc, q)    # initializing the superpositioning
    xgate(qc, q)         # flip all qubits
    qc.h(q[0])
```

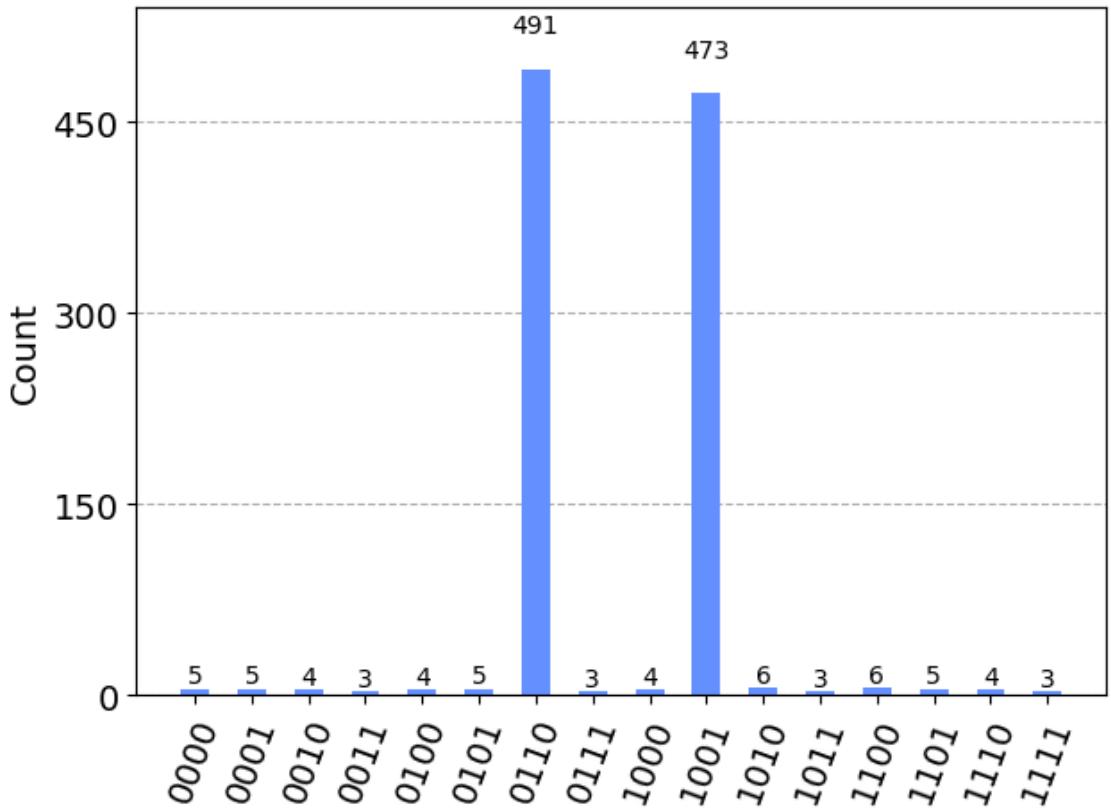
```
    qc.append(MCXGate(n-1), q[::-1])


    qc.h(q[0])
    qc.barrier(q[1:], label = '')
    xgate(qc, q)         # flip all qubits again
    Htransform(qc, q)    # return the superpositioning to its first stat

    qc.barrier()

    i += 1

# measure
qc.measure(q,c)

# Draw the circuit without barriers and reverse bit order:
display(qc.draw(output='mpl', plot_barriers=True, reverse_bits=True))
```



In [3]:
```
# execute the circuit and print the count for each state
job = execute(qc, Aer.get_backend('qasm_simulator'), shots= 1024)
counts = job.result().get_counts()
print('Results:', counts)
```

```
Results: {'1111': 3, '1000': 4, '0110': 491, '1101': 5, '1100': 6, '010
0': 4, '1010': 6, '0010': 4, '0000': 5, '1011': 3, '0111': 3, '0011':
3, '1110': 4, '1001': 473, '0101': 5, '0001': 5}
```

In [4]:
```
display(plot_histogram(counts))
```



In [8]:
```
# Save account credentials.
IBMProvider.save_account(token='0b9dc8a19b02b5f2f7b64516df38295e1520d112

# Load previously saved account credentials.
provider = IBMProvider()

# Select a backend.
backend = provider.get_backend("ibmq_quito")

# Transpile the circuit for the backend
transpiled = transpile(qc, backend=backend)

# Submit a job.
job = backend.run(transpiled)
# Get results.
print(job.result().get_counts())
```

```
{'0000': 407, '0001': 281, '0010': 253, '0011': 160, '0100': 328, '010
1': 216, '0110': 244, '0111': 144, '1000': 343, '1001': 255, '1010': 26
5, '1011': 153, '1100': 326, '1101': 219, '1110': 234, '1111': 172}
```

In [9]:
```
job.status()
```

Out[9]:
```
<JobStatus.DONE: 'job has successfully run'>
```

In [10]:
```
job.job_id()
```

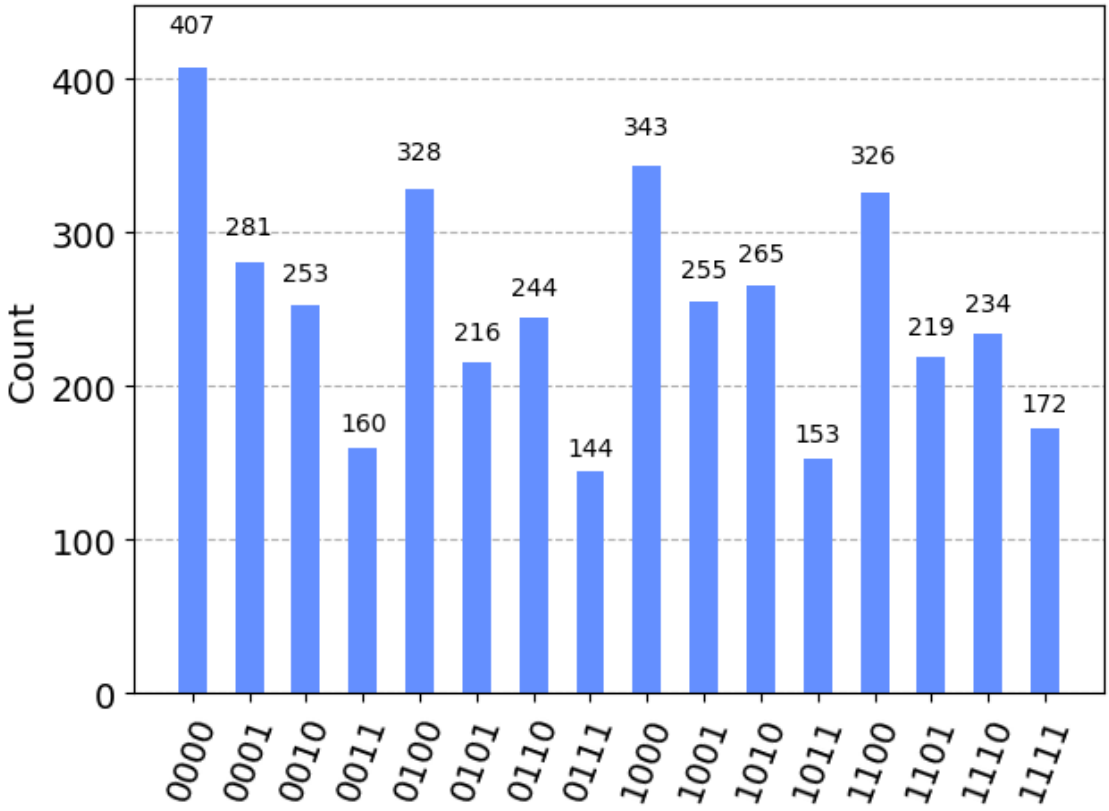
Out[10]:
```
'chkhe2pike34bjhegpfg'
```

In [13]:
```
result = job.result()
counts = result.get_counts()
```

```
print(counts)
```

```
{'0000': 407, '0001': 281, '0010': 253, '0011': 160, '0100': 328, '010
1': 216, '0110': 244, '0111': 144, '1000': 343, '1001': 255, '1010': 26
5, '1011': 153, '1100': 326, '1101': 219, '1110': 234, '1111': 172}
```

In [14]:
```
display(plot_histogram(counts))
```



---

7. Conclusion

Grover's algorithm is powerful. It can solve pretty much any problem that can be expressed as a series of logic gate checks, which is quite a lot. And, it runs in $O(\sqrt{N})$ time, a significant improvement over classical computers with $O(N)$ runtime. A quadratic speedup. Once quantum computers get bigger, Grover's algorithm might end up being used everywhere in computing.

---

8. References

1. L.K. Grover, Proceedings of the 8th Annual ACM Symposium on Theory of Computing (ACM Press, New York, USA, 1996), pp. 212–219
2. L.K. Grover, Am. j. Phys. 69, 769 (2001)
3. https://learn.qiskit.org/course/ch-algorithms/grovers-algorithm