Project

Name: Chinmay Patwardhan.

Class: MscComputerScience 1st year 2nd Sem

Subjec: Project

Project Title: Solving specific Problems using Grover's Quantum Search algorithm.

Aim: To solve numerous Quantum Search Problems using Grover's Quantum Search algorithm in a simulated quantum state on a quantum computer.

Procedure to follow:

1. Get access to Quantum Computer.

2. Test run "Hello World " quantum program to test if access to quantum computer is available or not.

3. Run simulations on various problems related to and solvable by Grover's Quantum Search Algorithm.

Test Running "Hello World" Simulation on IBM's Quantum Computer using qiskit library on python

Code:

```
from qiskit import *
```

Next, we will initialize a quantum register having 2 qubits. A qubit is the fundamental unit of a quantum circuit, the analogue of a bit in classical computing! We will also initialize a classical register having 2 qubits. Most circuits will be a combination of classical and quantum registers, as the quantum registers will be used to perform quantum mechanical operations on qubits, and classical registers will be used to perform classical operations on the measurements obtained.

```
qr=QuantumRegister(2)
cr=ClassicalRegister(2)
circuit=QuantumCircuit(qr,cr)
```

In Qiskit, we have the option to visualize how the quantum circuit looks by running the following command.

```
%matplotlib inline
circuit.draw()
```

We will see two quantum bits and two classical bits.

We will see the effect of applying a quantum gate on a single bit, the Hadamard gate, which is a gate used for creating superposition

We will do this by running the following command to the first qubit in the quantum register. Note that indexing starts from 0 in Python.

```
circuit.h(qr[0])
```

To visualize the circuit, we will run the command

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

Let us explore more fundamental gates, like the cx gate which is a controlled-x gate, and the gate performs a NOT on the target qubit if the control qubit is in state 1. By default, all qubits are initialized to state 0.

```
circuit.cx(qr[0],qr[1])
```

The control qubit is the first parameter or qr[0] in this case and the target parameter is qr[1]

In the following circuit, the target qubit is a circle with an addition sign and the control is a dot.

This circuit is used to implement entanglement between two qubits. After initializing a quantum circuit and performing computation by implementing quantum gates, we will extract the outputs by using .measure() function in Qiskit. In this example, the classical bits will be used to store the outputs.

```
circuit.measure(qr,cr)
```

The arrows point from the qubits, to the classical bits which are used to store the extracted outputs.

Now, the fun part is executing the circuit we built on a quantum computer. There are many quantum devices that Qiskit has. In order to use these devices, we will first have to load an IBMQ account (Note : follow this link https://quantum-computing.ibm.com/account to create an account)

```
IBMQ.load_account()
```

Next, we will have to give details of the IBMQ provider and quantum computer that we choose to execute our circuit on.

```
provider=IBMQ.get_provider('ibm-q')
quantum_computer=provider.get_backend('ibmq_vigo')
```

We use the execute() function to run our quantum circuit using ibmq_vigo as our backend.

```
execute_circuit=execute(circuit,backend=quantum_computer)
```

To see the results, simply run the following command.

```
result=execute_circuit.result()
```

To visualize the results, run the following command.

```
plot_histogram(result.get_counts(circuit))
```

Theoretically, we can compare the results we get on a quantum computer vs the results simulated on a quantum simulator.

Run the following command to get the simulator backend(we will use the qasm_simulator)

```
simulator=Aer.get_backend('qasm_simulator')
```

Next, execute the circuit.

```
result=execute(circuit,backend=simulator).result()
```

Plot the histogram of results to see the probabilities of getting each state in the output.

```
plot_histogram(result.get_counts(circuit))
```

Comparing the results, we see that on the quantum computer we get a small probability for obtaining states other than 00 and 11. This means that the quantum computer does not give 100% accurate results, but gives us an approximation. This is because quantum computers are not prone from noise, and in our next tutorial we will explore quantum noise and how to mitigate noise.

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
from qiskit import execute
from qiskit import IBMQ
from qiskit.tools.jupyter import *
from qiskit.providers.ibmq import least_busy
# import basic plot tools
from qiskit.tools.visualization import plot_histogram, circuit_drawer
```

In [2]:

```
IBMQ.load_account()
```

In [3]:

```
IBMQ.backends()
```

Out[3]:

```
[<IBMQBackend('ibmqx4') from IBMQ()>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ()>,
 <IBMQBackend('ibmq_qasm_simulator') from IBMQ()>]
```

In [4]:

```
backend = least_busy(IBMQ.backends(simulator=False))
print("The least busy backend is " + backend.name())
The least busy backend is ibmqx4
```

In [5]:

```
%%qiskit_job_status
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q, c)
qc.h(q[0])
qc.cx(q[0], q[1])
qc.measure(q, c)
job_exp = execute(qc, backend=backend, shots=1024, max_credits=3)
VBox(children=(HTML(value="<p style='font-size:16px;'>Job Status : job is being initialized
</p>"),))
```

In [6]:

```
plot_histogram(job_exp.result().get_counts(qc))
print('You have made entanglement!')
You have made entanglement!
```

The circuit that was run on the machine is

In [7]:

```
qc.draw()
```

Out[7]:

```
         ┌───┐          ┌─┐
q0_0: |0>┤ H ├──■───────┤M├─
         └───┘┌─┴─┐┌─┐ └╥┘
q0_1: |0>─────┤ X ├┤M├──╫──
              └───┘└╥┘  ║
 c0_0: 0 ═══════════╬═══╩══
                    ║
 c0_1: 0 ═══════════╩══════
```

Simulating nature is something that is really hard something like modeling atomic bombing an orbital overlap.

Insterad of writing out sumations over many terms in quantum computing you actually mimic a system you are trying to simulate directly on a quantum computer.

Quantum computer creates new possibilities and new ways to approach problems that classical computers have difficulty doing.

What are Quantum Computers?

- Quantum computers are machines that use the properties of quantum mechanics to store data and perform computations.

- This can be extremly usefull for certain tasks where they could vastly outperform even our best supercomputers.

- Classical computers which include smart phones and laptops encode information in binary “bits” that can either be 0s or 1s. --------> In Quantum computing the basic unit of memory is quantum bit or qbit.

- Qbits are made using physical systems such as the spin of an electron or the orientation of a photon.

- These systems can be in many different arrangements all at once quantum superposition.

- Qbits can also be linked together un a way that is impossible to disentangle or seperate using a phenomenon called quantum entanglement.

- The result is that a series of qbits can represent different thing simultaniously.

- For example: In quantum computer 8 bit is enough to represent any number between 0 to 255. But 8 Qbits is enough for a quantum computer to represent any number between 0-255 at the same time. A few hundred entangled qbits would be enough to represent more numbers than there are atoms in the univers.

This is where Quantum Computers get their edge over classical computers. In situations where there are large number of possible combinations, quantum computers can consider them simultaniously.

Examples:

Trying to find prime factors of a very large number or the best route between two places.

A quantum computer rather than using transistors it is using spins.

Different states can be acheived using superpositions of spins.

In this case more combinations means more memory.

Superposition

Entanglement

Interference

We can have constructive interference and distructive interference.

If constructive wave amplitudes that add to the signal gets larger and if destructive interference the amplitudes cancel each other by using a property like interference quantum computers can contro quantum states and amplify the kinds of signals that are towards the right answer and cancel the types of signals that are leading to wrong answers.

How are Quantum Computers Built?

Need to be able to have an object to call it a qbit or quantum bit that can actually be put into superpositions of states.

Those two qbits states that we can physically entangle with each other

Atoms, Ions, Superconducting Qbits.

When we store a bit into our classical hard drives there is a magnetic domain and we have a magnetic polarization we can change the magnatisation to pointing up or pointing down.

Quantum Systems were still manipulating a device and changing the quantum state of that device we can imagine if its a spin that we can have spin ip and spin down.

But if we isolate it wnough we can have superposition of up and down.

We encode the problems we are going to solve into a computer's quantum state and then we manipulate that state to drive it towards what will eventually represent a solution.

We need a chip with a qbit, each q-bit is a carrier of quantum information and the way we control the state of theat q-bit is using microwave pulses.

We calibrate these microwave pulses so that we know exactly what kind of pulse with what kind of frequency and also to how much duration will put these two qbits into superposition or we will flip the state of the qbit from 0 to 1 or if we aply macrowave pulse between two qbits we can entangle them.

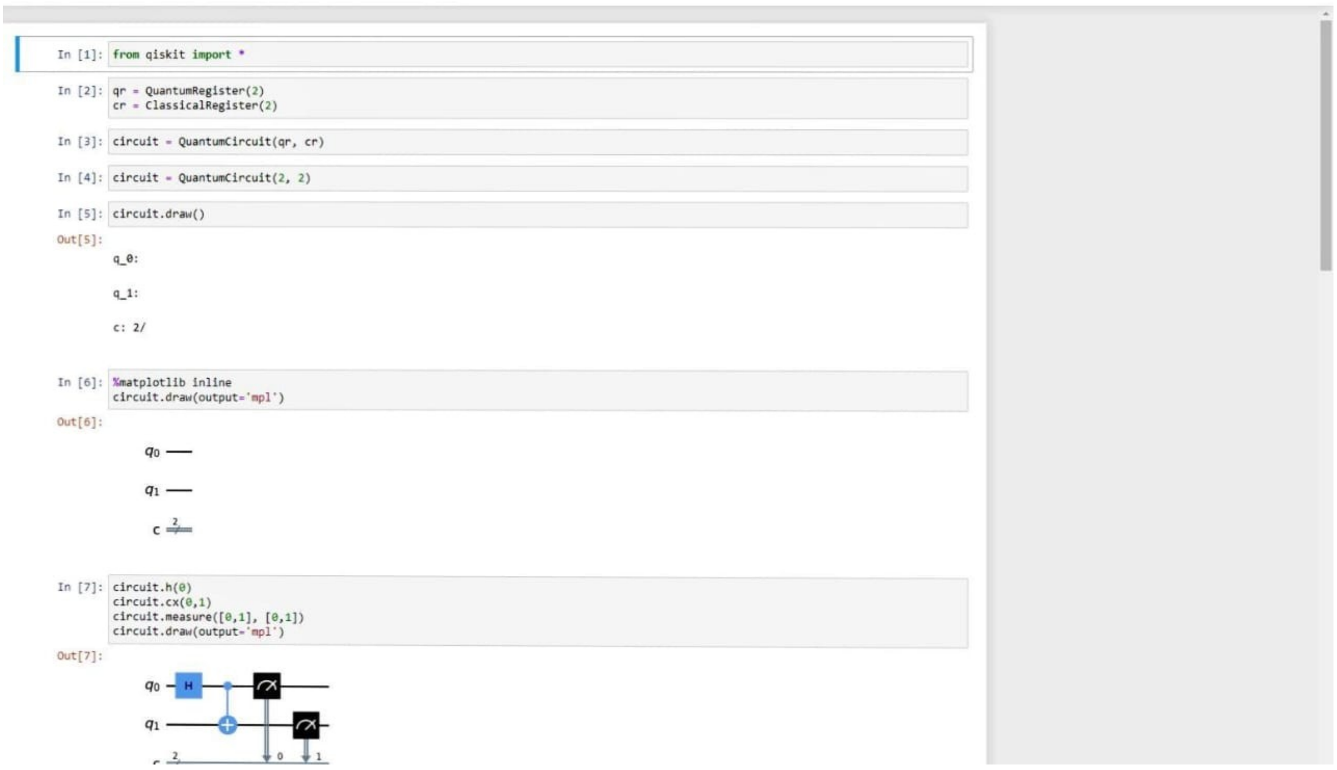
We can then measure results with microwave signals. The key is to come up with algorithms where the result is deterministic.

How those algorithms look like?

There are two main classes of quantum algorithms.
1. Algorithms whcih were developed for decads
----> Shor’s algorithm which is for factoring,
----> Grover’s algorithm which is for unstructured search

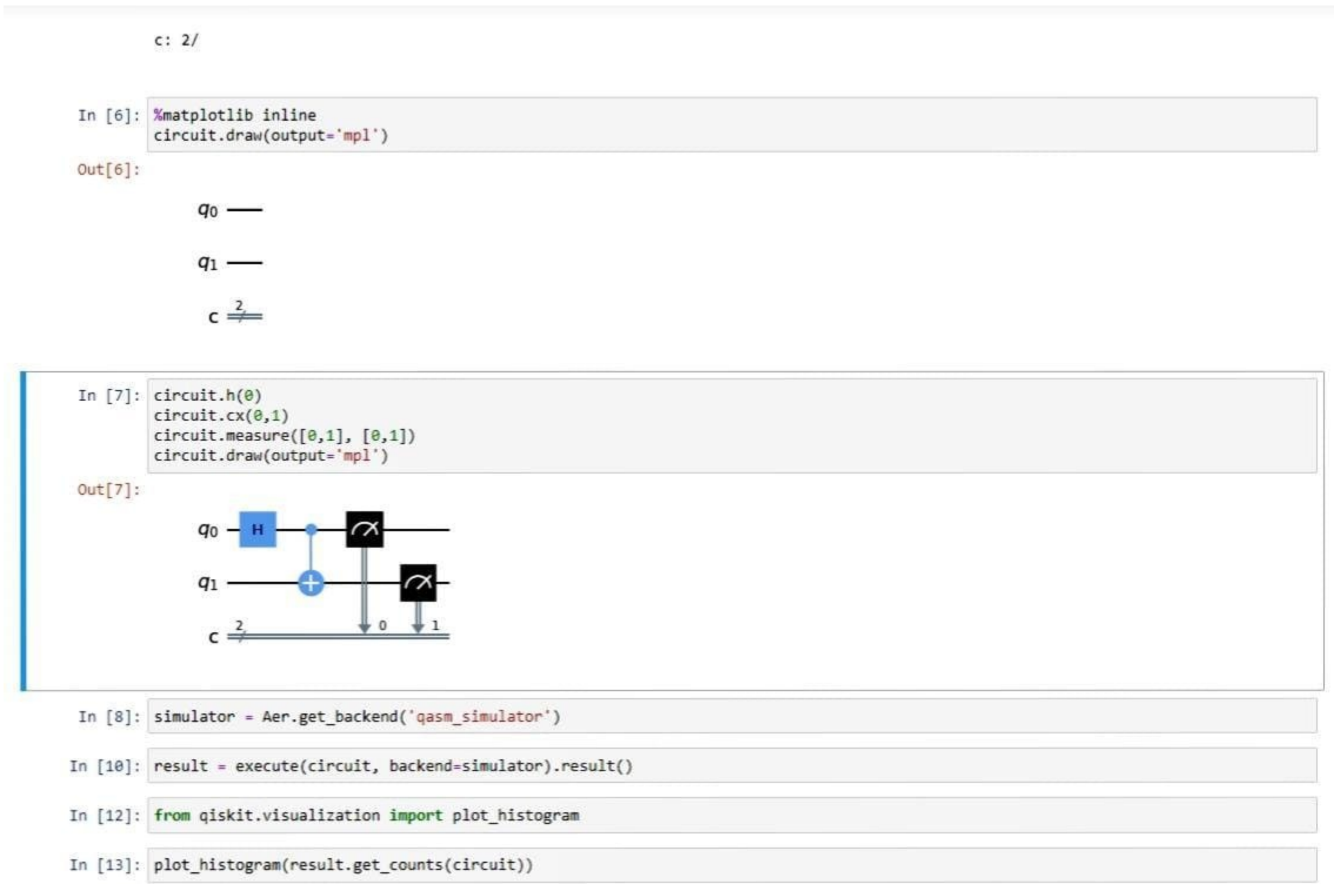
and these algorithms were designed assuming we had a perfect fault-tolerant Quantum Computer

In [ ]:



```
In [1]: from qiskit import *

In [2]: qr = QuantumRegister(2)
        cr = ClassicalRegister(2)

In [3]: circuit = QuantumCircuit(qr, cr)

In [4]: circuit = QuantumCircuit(2, 2)

In [5]: circuit.draw()
Out[5]:
        q_0:

        q_1:

        c: 2/

In [6]: %matplotlib inline
        circuit.draw(output='mpl')
Out[6]:
```

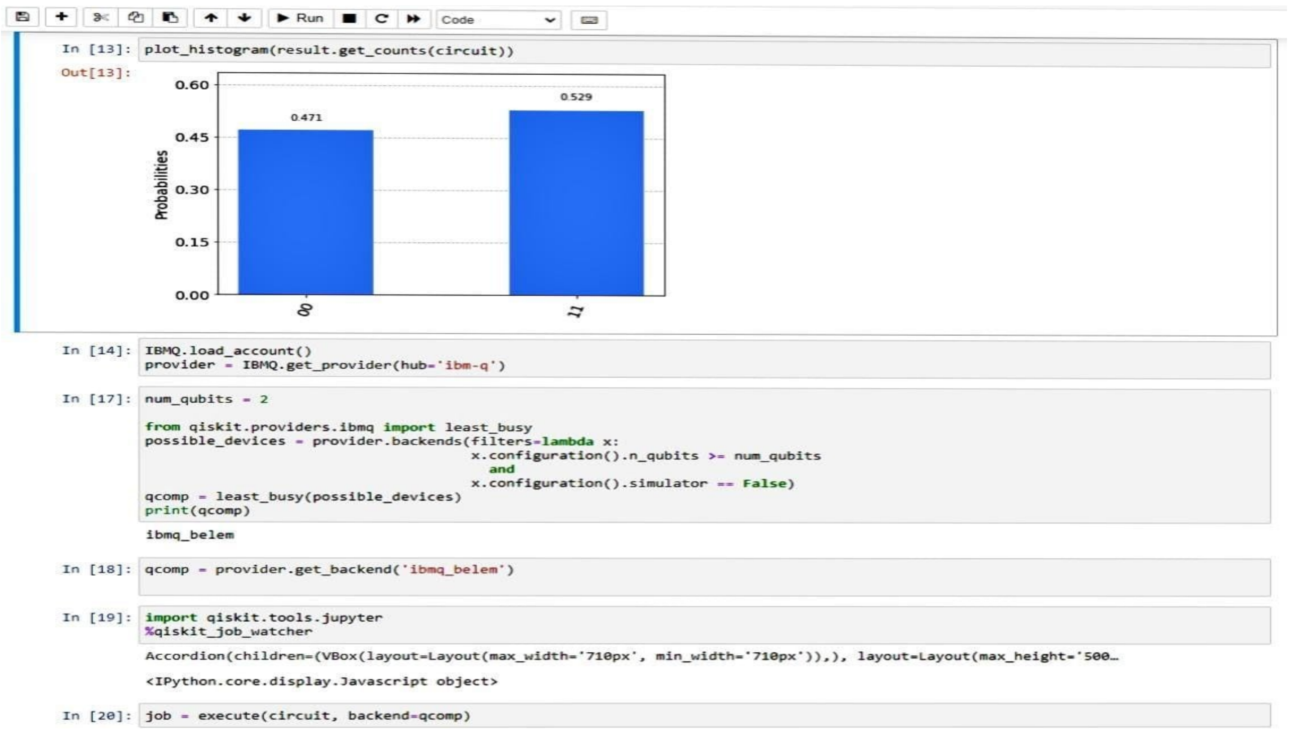
```
In [7]: circuit.h(0)
        circuit.cx(0,1)
        circuit.measure([0,1], [0,1])
        circuit.draw(output='mpl')
Out[7]:
```



```
        c: 2/

In [6]: %matplotlib inline
        circuit.draw(output='mpl')
Out[6]:
```

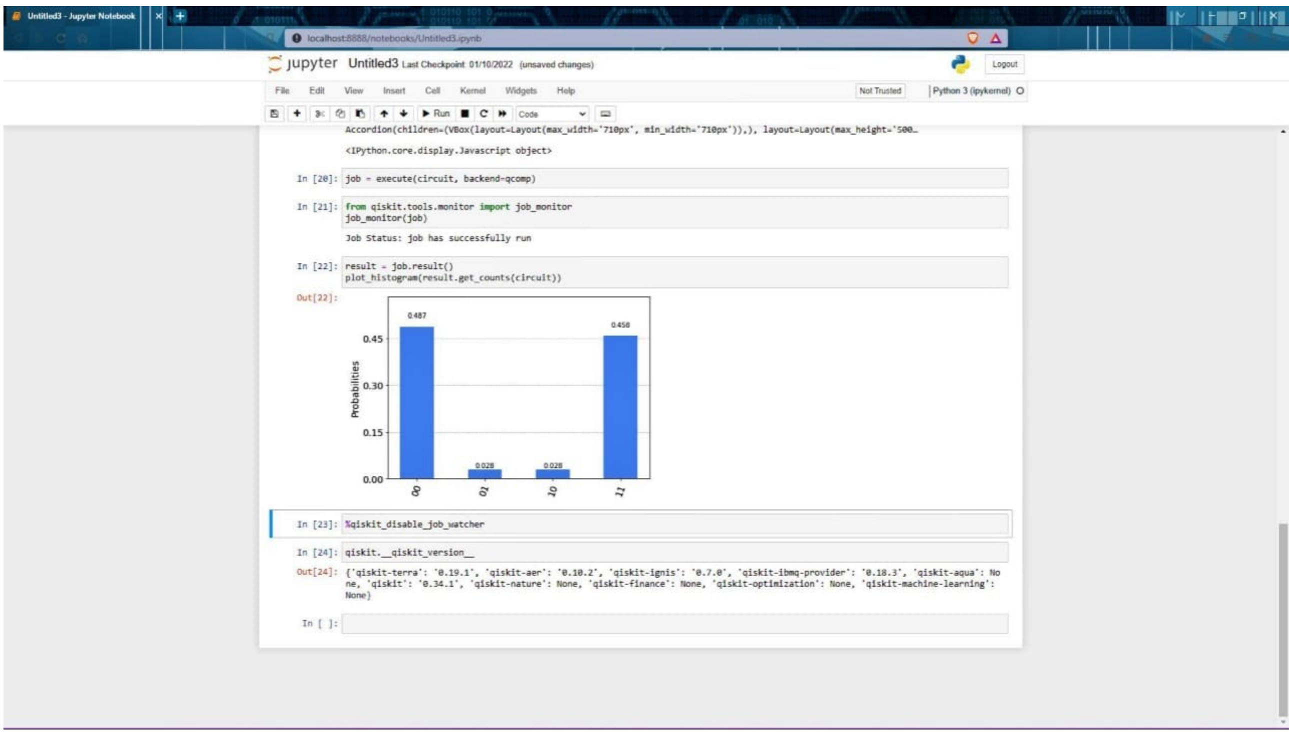
```
In [7]: circuit.h(0)
        circuit.cx(0,1)
        circuit.measure([0,1], [0,1])
        circuit.draw(output='mpl')
Out[7]:
```

```
In [8]: simulator = Aer.get_backend('qasm_simulator')

In [10]: result = execute(circuit, backend=simulator).result()

In [12]: from qiskit.visualization import plot_histogram

In [13]: plot_histogram(result.get_counts(circuit))
```



```
In [13]: plot_histogram(result.get_counts(circuit))
Out[13]:
```

```
In [14]: IBMQ.load_account()
         provider = IBMQ.get_provider(hub='ibm-q')

In [17]: num_qubits = 2

         from qiskit.providers.ibmq import least_busy
         possible_devices = provider.backends(filters=lambda x:
                                               x.configuration().n_qubits >= num_qubits
                                                 and
                                               x.configuration().simulator == False)
         qcomp = least_busy(possible_devices)
         print(qcomp)

         ibmq_belem

In [18]: qcomp = provider.get_backend('ibmq_belem')

In [19]: import qiskit.tools.jupyter
         %qiskit_job_watcher

         Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),), layout=Layout(max_height='500…

         <IPython.core.display.Javascript object>

In [20]: job = execute(circuit, backend=qcomp)
```



Untitled3 - Jupyter Notebook
localhost:8888/notebooks/Untitled3.ipynb
jupyter Untitled3 Last Checkpoint: 01/10/2022 (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
         Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),), layout=Layout(max_height='500…

         <IPython.core.display.Javascript object>

In [20]: job = execute(circuit, backend=qcomp)

In [21]: from qiskit.tools.monitor import job_monitor
         job_monitor(job)

         Job Status: job has successfully run

In [22]: result = job.result()
         plot_histogram(result.get_counts(circuit))
Out[22]:
```

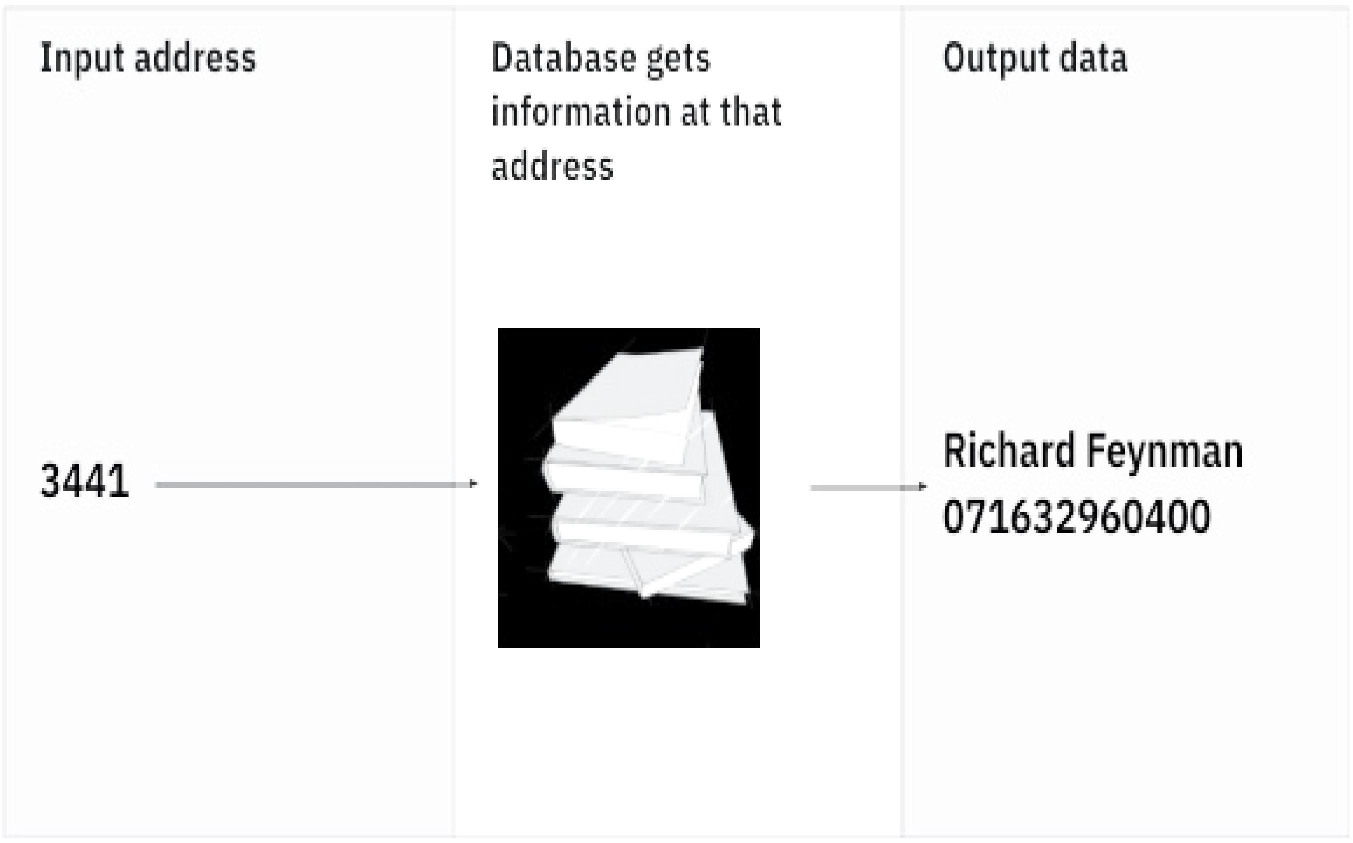
```
In [23]: %qiskit_disable_job_watcher

In [24]: qiskit.__qiskit_version__
Out[24]: {'qiskit-terra': '0.19.1', 'qiskit-aer': '0.10.2', 'qiskit-ignis': '0.7.0', 'qiskit-ibmq-provider': '0.18.3', 'qiskit-aqua': No
         ne, 'qiskit': '0.34.1', 'qiskit-nature': None, 'qiskit-finance': None, 'qiskit-optimization': None, 'qiskit-machine-learning':
         None}

In [ ]:
```

Grover's search algorithm

Search problems

Grover's algorithm. It is a continuous simulation that integrates Schrodinger's equation over time using Runge-Kutta method. It is not a discrete, unitary simulation.

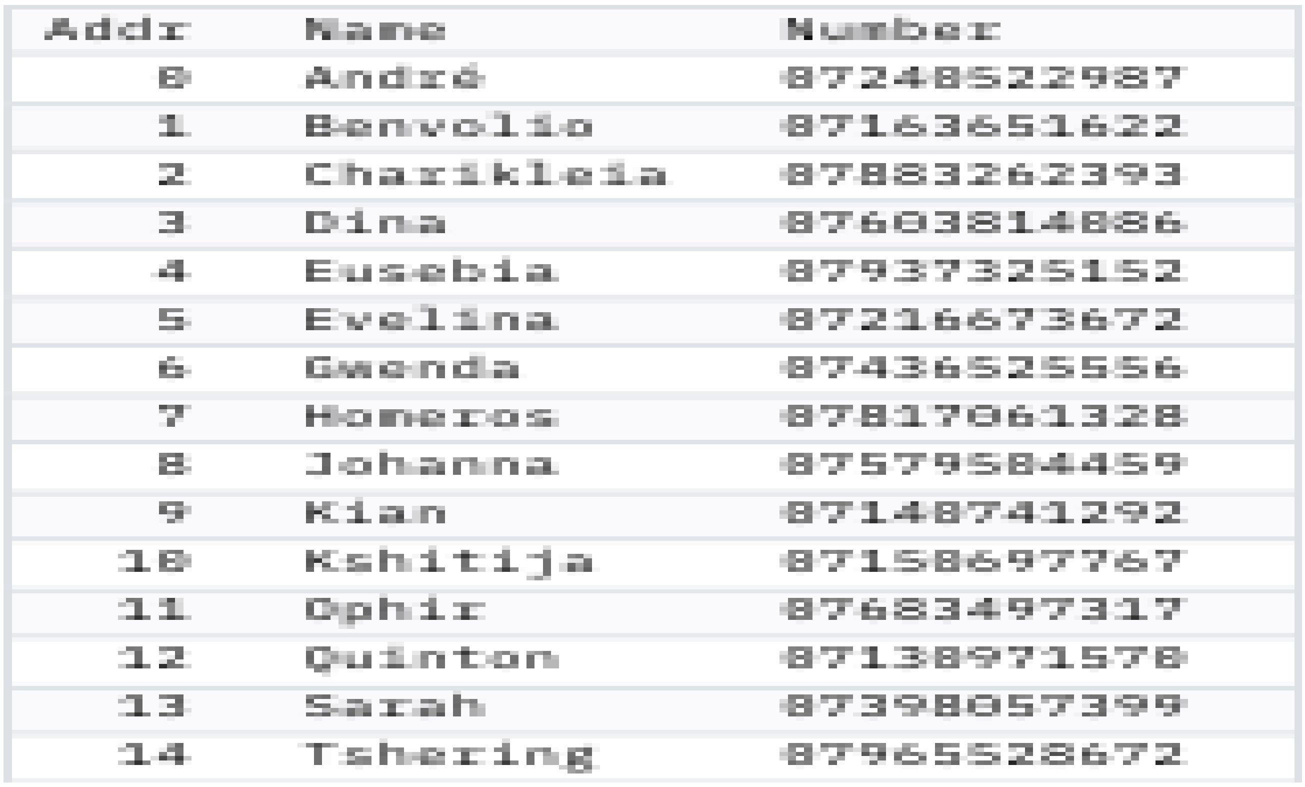
A lot of the problems that computers solve are types of *search problems*. You’ve probably already searched the web using a search engine, which is a program that builds a database from websites and allows you to search through it. We can think of a database as a program that takes an address as input, and outputs the data at that address. A phone book is one example of a database; each entry in the book contains a name and number. For example, we might ask the database to give us the data in at the 3441st address, and it will return the 3441st name and number in the book.



We call this process of providing an input and reading the output "querying the database". Often in computer science, we consider databases to be black boxes, which means we're not allowed to see how they work; we’ll just assume they're magical processes that do exactly as they promise. We call magical processes like these "oracles".

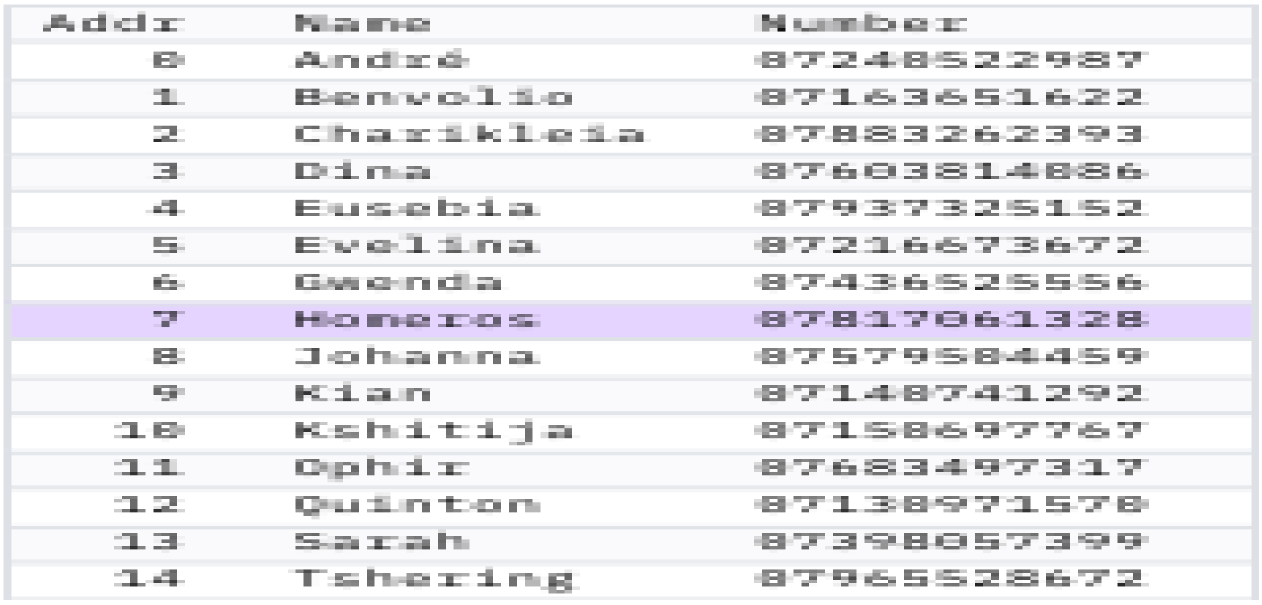
If we have someone's name and we’re trying to find their phone number, this is easy if the book is sorted alphabetically by name. We can use an algorithm called *binary search*.

Example: Binary search

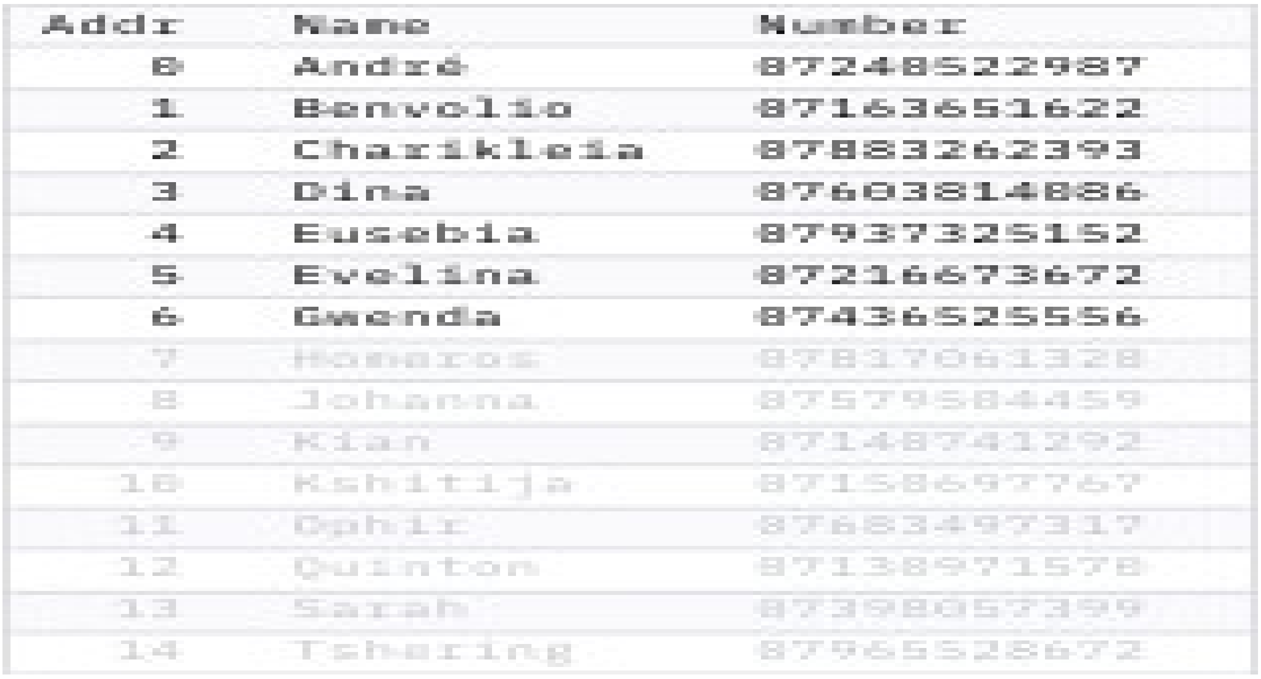


Addr	Name	Number
0	André	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87603814886
4	Eusebia	87937325152
5	Evelina	87216673672
6	Gwenda	87436525556
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672

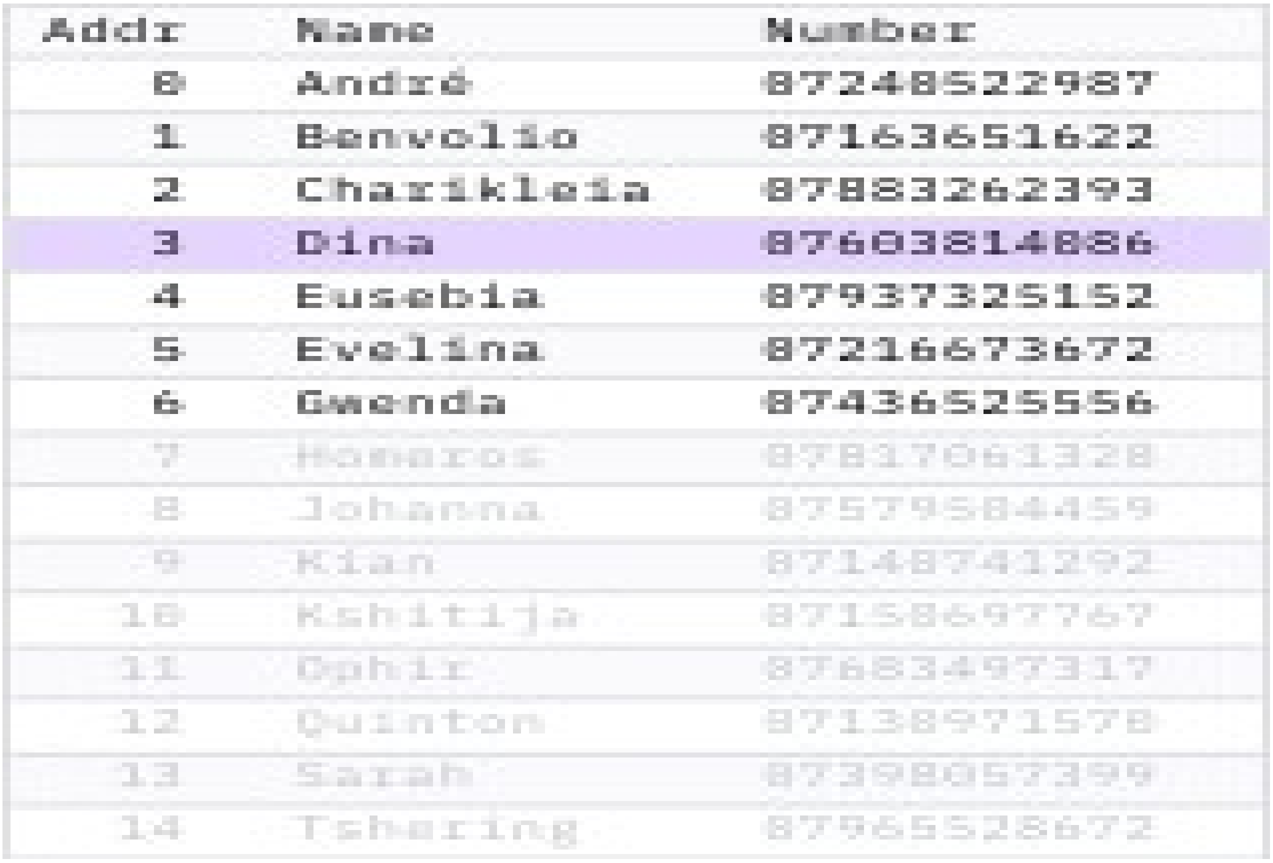
Binary search is a very efficient classical algorithm for searching sorted databases. You’ve probably used something similar when searching for a specific page in a book (or even using a physical phone book). Let’s say we want to find Evelina's phone number.



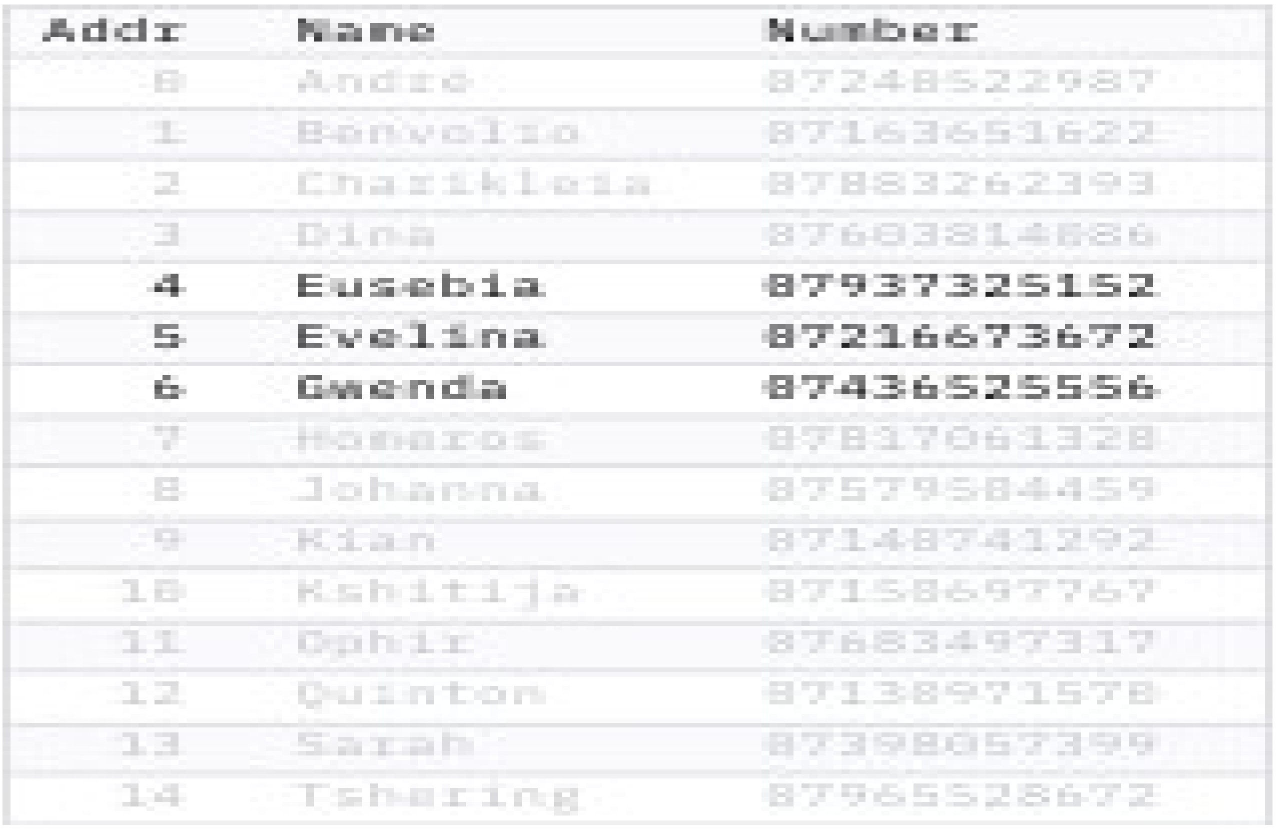
Addr	Name	Number
0	André	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87603814886
4	Eusebia	87937325152
5	Evelina	87216673672
6	Gwenda	87436525556
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672



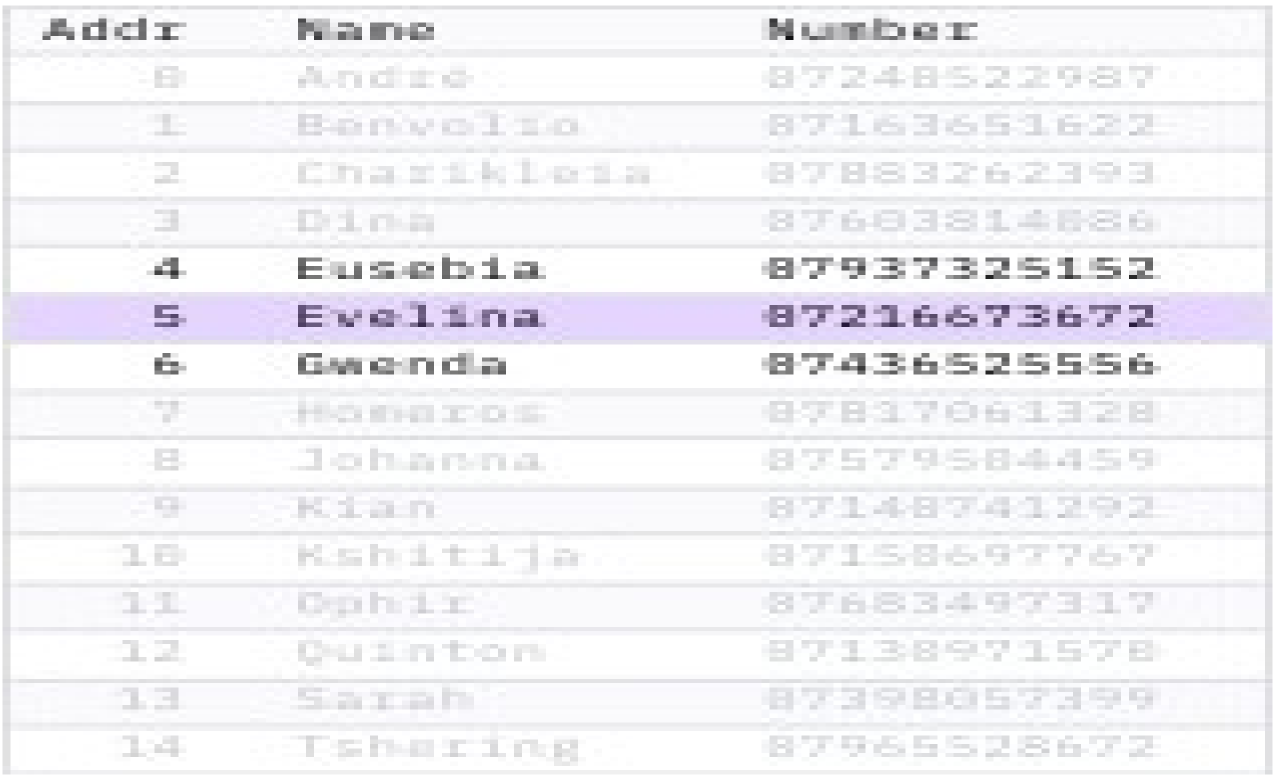
Addr	Name	Number
0	André	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87603814886
4	Eusebia	87937325152
5	Evelina	87216673672
6	Gwenda	87436525556
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672



Addr	Name	Number
0	André	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87603814886
4	Eusebia	87937325152
5	Evelina	87216673672
6	Gwenda	87436525556
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672



Addr	Name	Number
0	Andre	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87683814886
4	**Eusebia**	**87937325152**
5	**Evelina**	**87216673672**
6	**Gwenda**	**87436525556**
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672



Addr	Name	Number
0	Andre	87248522987
1	Benvolio	87163651622
2	Charikleia	87883262393
3	Dina	87683814886
4	**Eusebia**	**87937325152**
5	**Evelina**	**87216673672**
6	**Gwenda**	**87436525556**
7	Homeros	87817061328
8	Johanna	87579584459
9	Kian	87148741292
10	Kshitija	87158697767
11	Ophir	87683497317
12	Quinton	87138971578
13	Sarah	87398057399
14	Tshering	87965528672

-

Since binary search grows logarithmically

This is the inverse of exponential. This grows very slowly, so logarithmic algorithms are generally considered very efficient.

with the size of the database, there isn’t much room for improvement from a quantum computer. But we don’t always have the convenience of searching sorted lists. What if we were instead given a phone number, and we wanted to find the name associated with that number?

This is a lot more difficult, as phone books aren't usually sorted by number. If we assume the phone numbers are ordered randomly in the list, there’s no way of homing in on our target as we did last time. The best we can do with a classical computer is randomly pick an input address, see if it contains the phone number we’re looking for, and repeat until we happen upon the correct entry. For this reason, a lot of work goes into indexing

To make searching easier, we could have two copies of the phone book: One sorted by name and the other sorted by number. This will obviously take a while to create and will take up more space on the disk, but it could be worth it if we’re searching it a lot.

databases to improve search times.

When the database is completely disordered like this, we say it's *unstructured*. And the quantum algorithm we'll learn about on this page is an algorithm for unstructured search.

Unstructured search

If we search an unstructured database by randomly choosing inputs, how many inputs would we need to check on average before we find the entry we're looking for?

1. Half the possible inputs.

Not quite

3. Three-quarters of the possible inputs.

Not quite

2. All the possible inputs.

Not quite

Using random guessing, how does the average number of database queries needed grow with the number of entries in the database?

2. Logarithmically.

Not quite

1. Linearly.

Not quite

3. Quadratically.

Not quite

4. Exponentially.

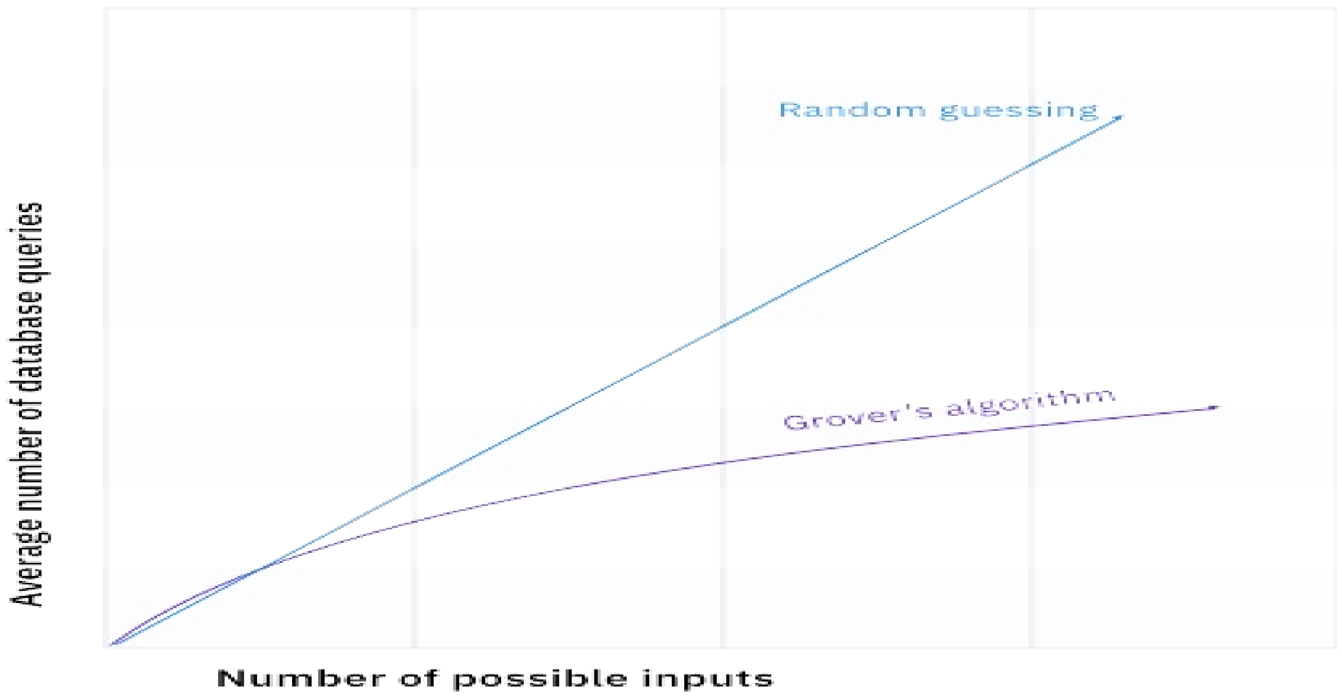
Not quite

It may seem that we can't possibly do better than random guessing here; we don't have any idea where the correct entry will be in the database, and each incorrect query only rules out one entry.

For classical computers, our intuition is correct, but if our database can input and output quantum superpositions, it turns out we can do better than random guessing! On this page we will learn about our first quantum algorithm: Grover's quantum search algorithm. When searching any database (structured or unstructured), Grover's algorithm grows with the *square root* of the number of inputs, which for unstructured search is a quadratic

'Quadratic' is anything to do with squares (i.e., multiplying a number by itself).

improvement over the best classical algorithm.



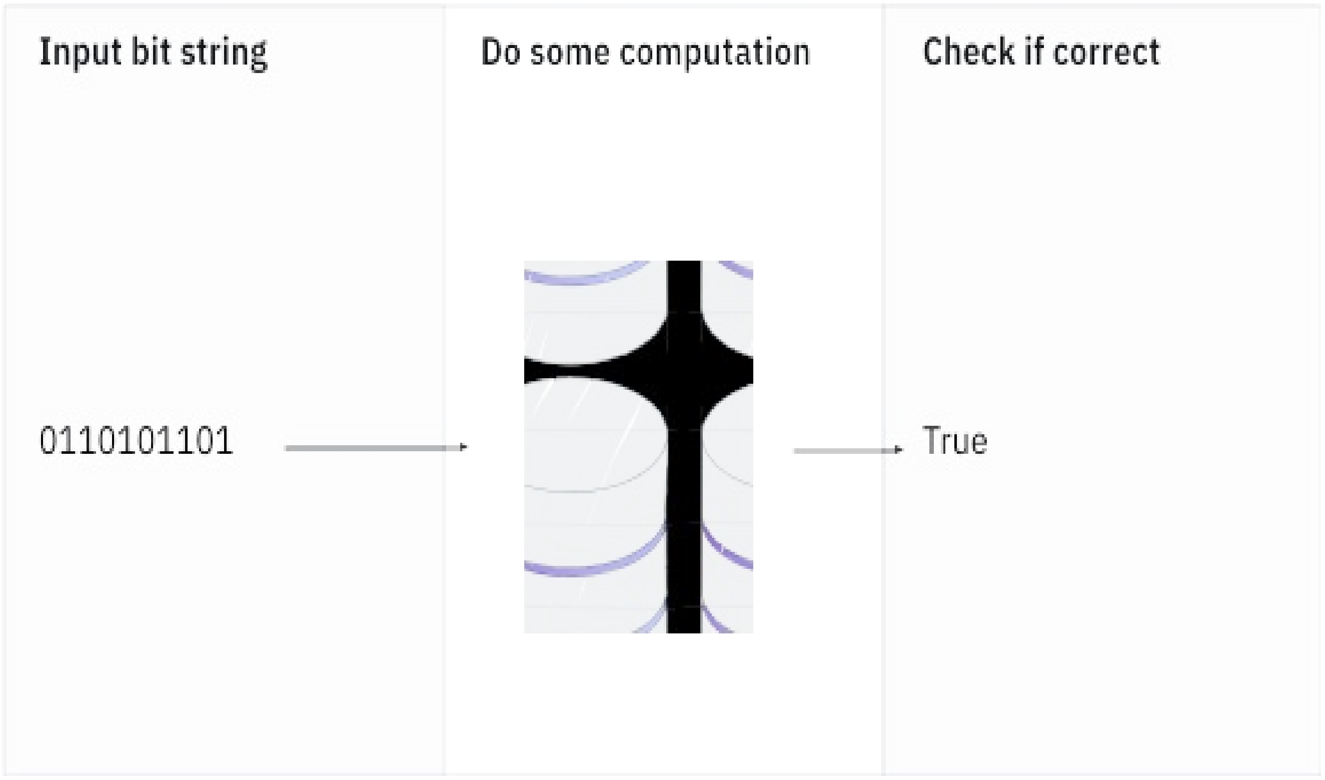
Beyond black boxes

Search algorithms can search databases of collected information such as phone books, but they can also do more than that. If we can make a problem *look* like a database search problem, then we can use a search algorithm to solve it. For example, let’s consider the problem of solving a sudoku

A sudoku is a type of puzzle. To solve it, you need to fill in numbers in a grid according to certain constraints.

. If someone claims to have solved a sudoku, you can check if it’s solved pretty quickly: You check along each row, check along each column, check each square, and you’re finished. In this sense, *you* are the database, and the person that gave you the solution is querying you. They are trying to find the input that returns the information “yes this is a valid solution”.

In fact, we can present a lot of computational problems as "find the input that results in a certain output".



One example of a problem we can solve like this is the Boolean satisfiability problem (known as 'SAT').

SAT problems

SAT problems are widely studied in computer science, and lots of other computing problems can be converted to SAT problems. In this page we will use Grover’s algorithm to solve a simple SAT problem, and you can use the skills you learn here to apply quantum search algorithms to other problems.

A solution to a SAT problem is a string of bits, which makes it easy to map to a quantum circuit. The problem itself is essentially a bunch of conditions (we call them clauses) that rule out different combinations of bit values. For example, if we had three bits, one of the clauses might be "You can’t have the zeroth bit `ON` *and* the first bit `OFF`", which would rule out the combinations `101` and `001` as valid solutions.

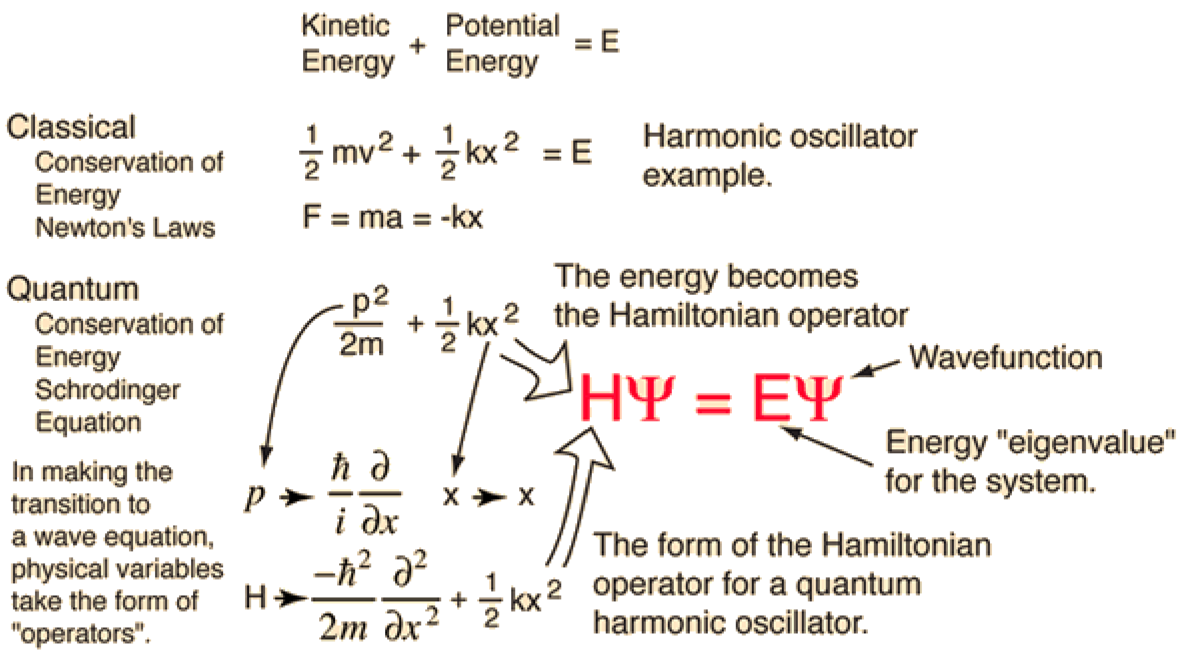
Here's a file that encodes a *"3-SAT*

Similar to sudoku, 3-SAT is a problem where you need to find a string of bits according to certain constraints.

" problem, which is a SAT problem where every clause refers to exactly 3 bits, and one of these bit conditions in each clause must be satisfied.

Schrodinger Equation

The Schrodinger equation plays the role of Newton's laws and conservation of energy in classical mechanics - i.e., it predicts the future behavior of a dynamic system. It is a wave equation in terms of the wavefunction which predicts analytically and precisely the probability of events or outcome. The detailed outcome is not strictly determined, but given a large number of events, the Schrodinger equation will predict the distribution of results.


Kinetic Energy + Potential Energy = E
Classical
Conservation of Energy
Newton's Laws
$\frac{1}{2}mv^2 + \frac{1}{2}kx^2 = E$
Harmonic oscillator example.
$F = ma = -kx$
Quantum
Conservation of Energy
Schrodinger Equation
$\frac{p^2}{2m} + \frac{1}{2}kx^2$
The energy becomes the Hamiltonian operator
$H\Psi = E\Psi$
Wavefunction
Energy "eigenvalue" for the system.
In making the transition to a wave equation, physical variables take the form of "operators".
$p \rightarrow \frac{\hbar}{i}\frac{\partial}{\partial x}$
$x \rightarrow x$
$H \rightarrow \frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}kx^2$
The form of the Hamiltonian operator for a quantum harmonic oscillator.

The kinetic and potential energies are transformed into the Hamiltonian which acts upon the wavefunction to generate the evolution of the wavefunction in time and space. The Schrodinger equation gives the quantized energies of the system and gives the form of the wavefunction so that other properties may be calculated.

Example 3-SAT problem

Here is an examples of a 3-SAT problem, stored in a file format called "DIMACS CNF". These files are very simple and are just one way of storing SAT problems.

Like with the sudoku, it’s easy to check if a bit string is a valid solution to a SAT problem; we just look at each clause in turn and see if our string disobeys any of them. In this course, we won’t worry about how we do this in a quantum circuit. Just remember we have efficient classical algorithms for checking SAT solutions, and for now we’ll just use Qiskit’s built-in tools to build a circuit that does this for us.

We've saved this file under `examples/3sat.dimacs` (relative to the code we're running).

1

with open('examples/3sat.dimacs', 'r', encoding='utf8') as f:

```
    dimacs = f.read()
print(dimacs)  # let's check the file is as promised
```

```
 c example DIMACS-CNF 3-SAT
p cnf 3 5
-1 -2 -3 0
1 -2 3 0
1 2 -3 0
1 -2 -3 0
-1 2 3 0
```

And we can use Qiskit's circuit library to build a circuit that does the job of the oracle we described above (we'll keep calling this circuit the 'oracle' even though it's no longer magic and all-powerful).

```
from qiskit.circuit.library import PhaseOracle
oracle = PhaseOracle.from_dimacs_file('examples/3sat.dimacs')
oracle.draw()
```

This circuit above acts similarly to the databases we described before. The input to this circuit is a string of 3 bits, and the output given depends on whether the input string is a solution to the SAT problem or not.

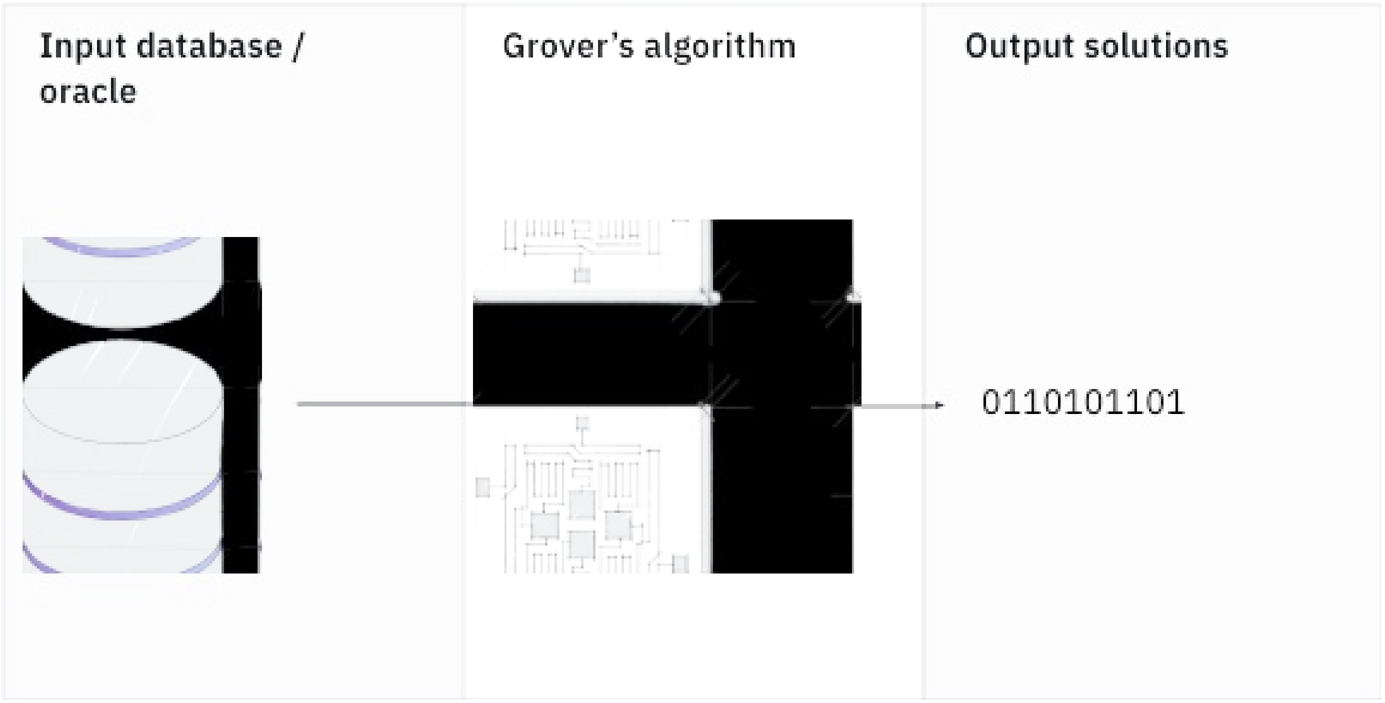
The result of this checking computation will still be either `True` or `False`, but the behaviour of this circuit is slightly different to how you might expect. To use this circuit with Grover's algorithm, we want the oracle to change the phase of the output state by 180° (i.e. multiply by -1) if the state is a solution. This is why Qiskit calls the class '`PhaseOracle`'.

For example, the only solutions to this problem are `000`, `011`, and `101`, so the circuit above has this matrix:

To summarise:

1. There are problems for which it's easy to check if a proposed solution is correct.
2. We can convert an algorithm that checks solutions into a quantum circuit that changes the phase of solution states
3. We can then use Grover's algorithm to work out which states have their phases changed.

In this sense, the database or oracle *is the problem* to be solved.



Overview of Grover's algorithm

Now we understand the problem, we finally come to Grover's algorithm. Grover's algorithm has three steps:

1. The first step is to create an equal superposition of every possible input to the oracle. If our qubits all start in the state , we can create this superposition by applying a H-gate to each qubit. We'll call this equal superposition state ".
2. The next step is to run the oracle circuit () on these qubits. On this page, we'll use the circuit (`oracle`) Qiskit created for us above, but we could use any circuit or hardware that changes the phases of solution states.
3. The final step is to run a circuit called the 'diffusion operator' or 'diffuser' () on the qubits. We'll go over this circuit when we explore Grover's algorithm in the next section, but it's a remarkably simple circuit that is the same for any oracle.

We then need to repeat steps 2 & 3 a few times depending on the size of the circuit. Note that step #2 is the step in which we query the oracle, so the number of times we do this is roughly proportional to the square root of the size of the number of possible inputs. If we repeat 2 & 3 enough times, then when we measure, we'll have a high chance of measuring a solution to the oracle.

```
from qiskit import QuantumCircuit
init = QuantumCircuit(3)
```

```
init.h([0,1,2])
init.draw()
```

Next, we can again use Qiskit's tools to create a circuit that does steps 2 & 3 for us.

```
# steps 2 & 3 of Grover's algorithm
from qiskit.circuit.library import GroverOperator
grover_operator = GroverOperator(oracle)
```

And we can combine this into a circuit that performs Grover's algorithm. Here, we won't repeat steps 2 & 3 as this is a small problem and doing them once is enough.

```
qc = init.compose(grover_operator)
qc.measure_all()
qc.draw()
```

Finally, let's run this on a simulator and see what results we get:

```
# Simulate the circuit
from qiskit import Aer, transpile
sim = Aer.get_backend('aer_simulator')
t_qc = transpile(qc, sim)
counts = sim.run(t_qc).result().get_counts()
```

```

# plot the results
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

We have a high probability of measuring one of the 3 solutions to the SAT problem.

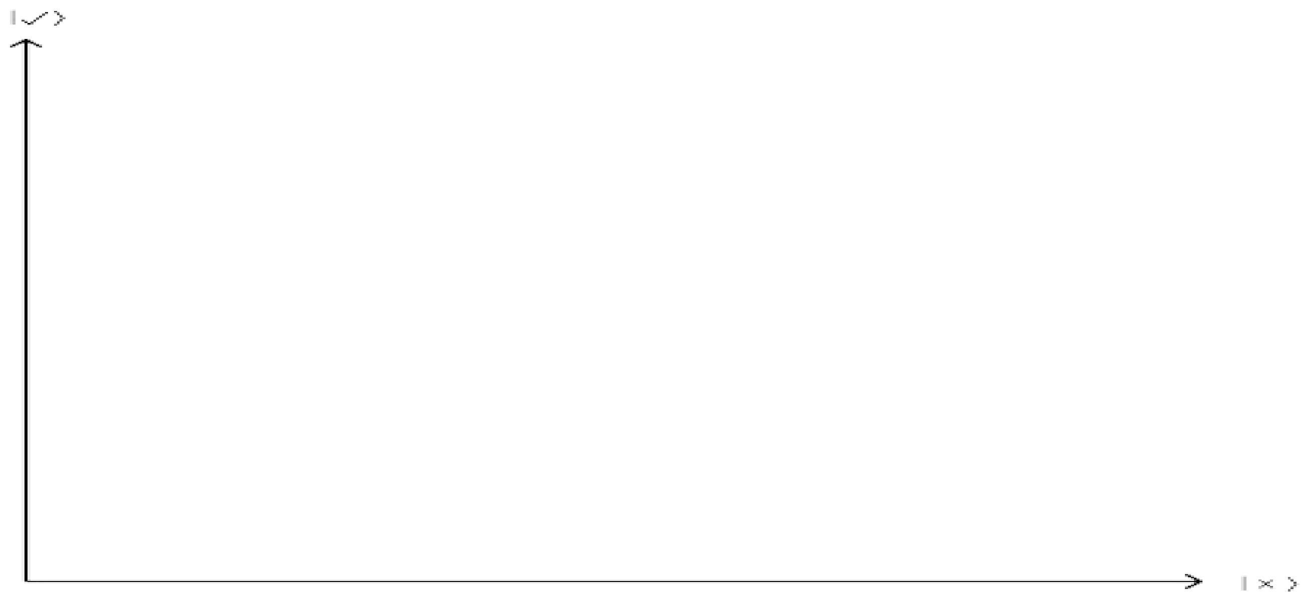
How does Grover's algorithm work?

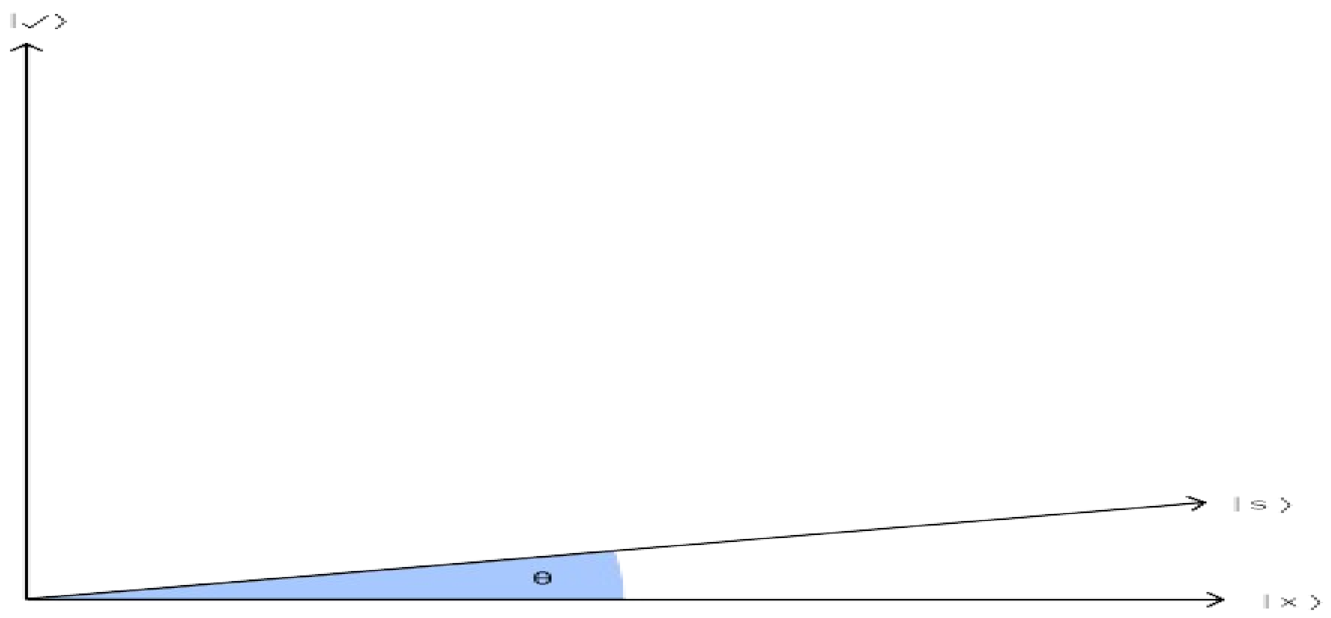
We’ve learnt about search problems, and seen Grover’s algorithm used to solve one. But how, and why, does this work?

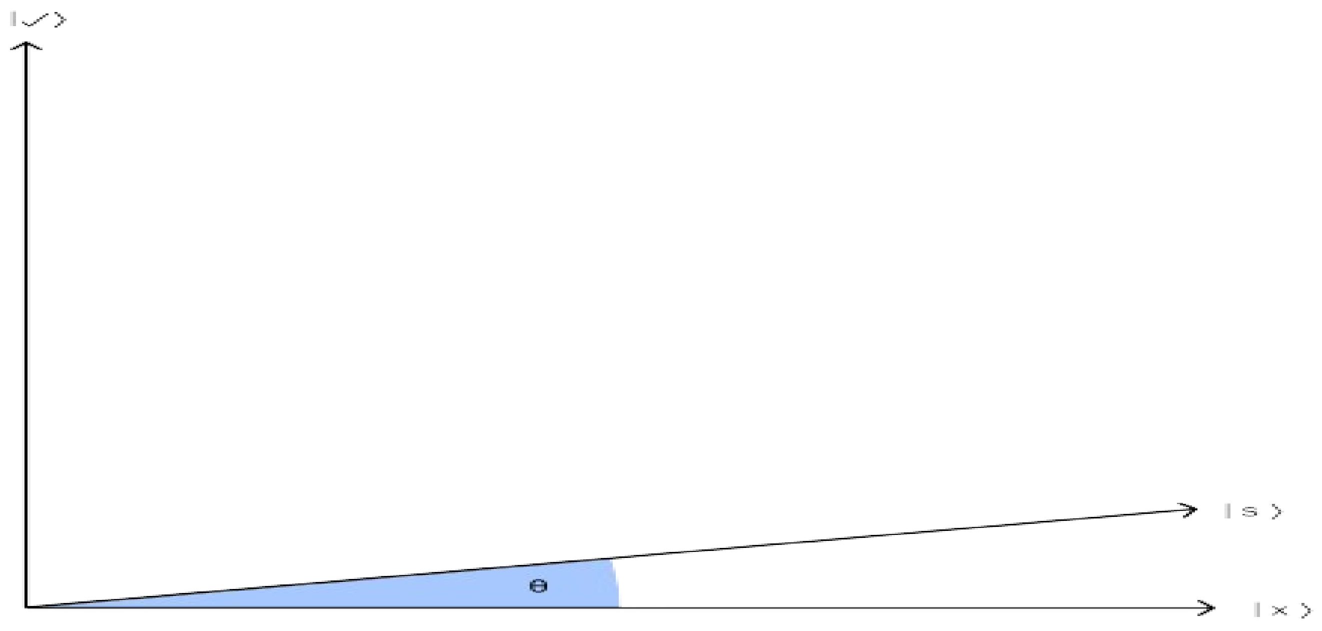
Visualising Grover's algorithm

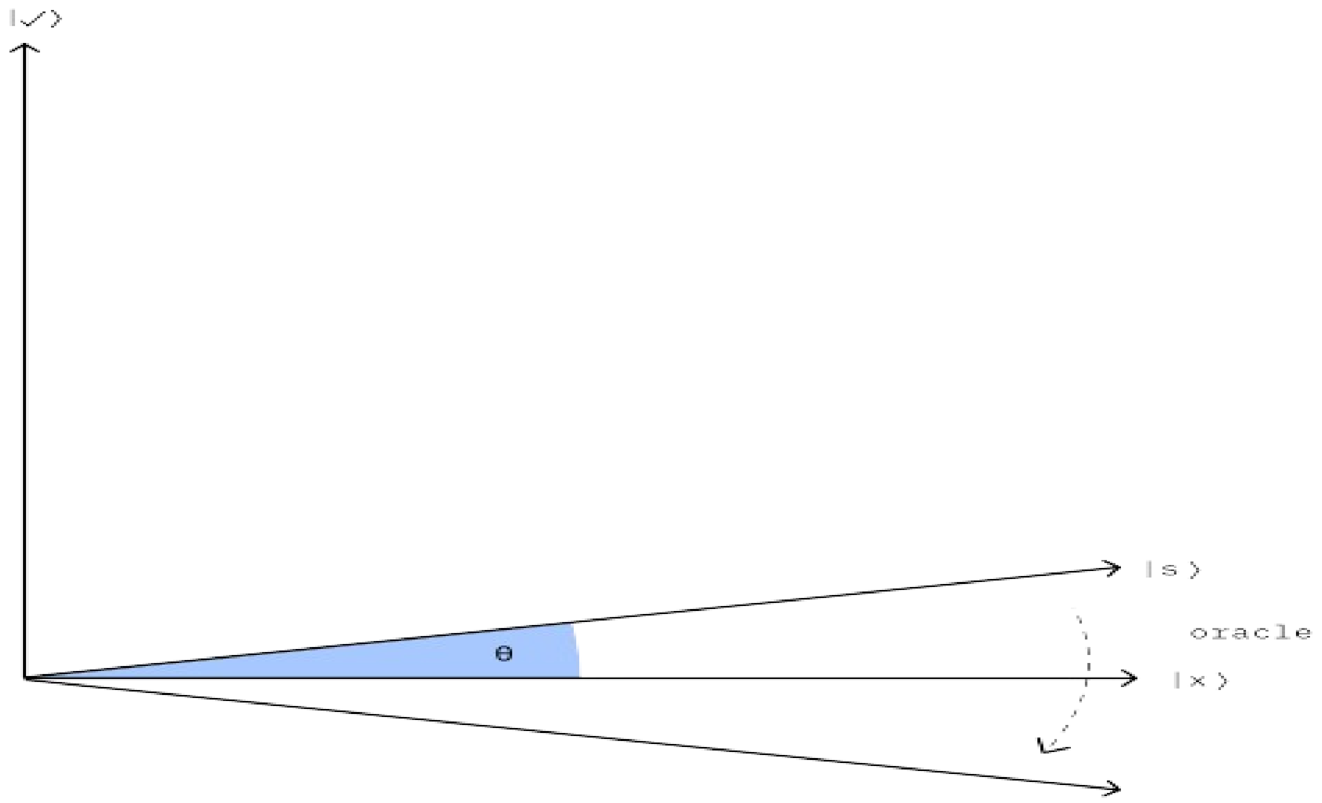
Grover’s algorithm has a nice geometric explanation. We’ve seen that we can represent quantum states through vectors. With search problems like these, there are only two vectors we care about: The solutions, and everything else. We'll call the superposition of all solution states ", so for the SAT problem above:

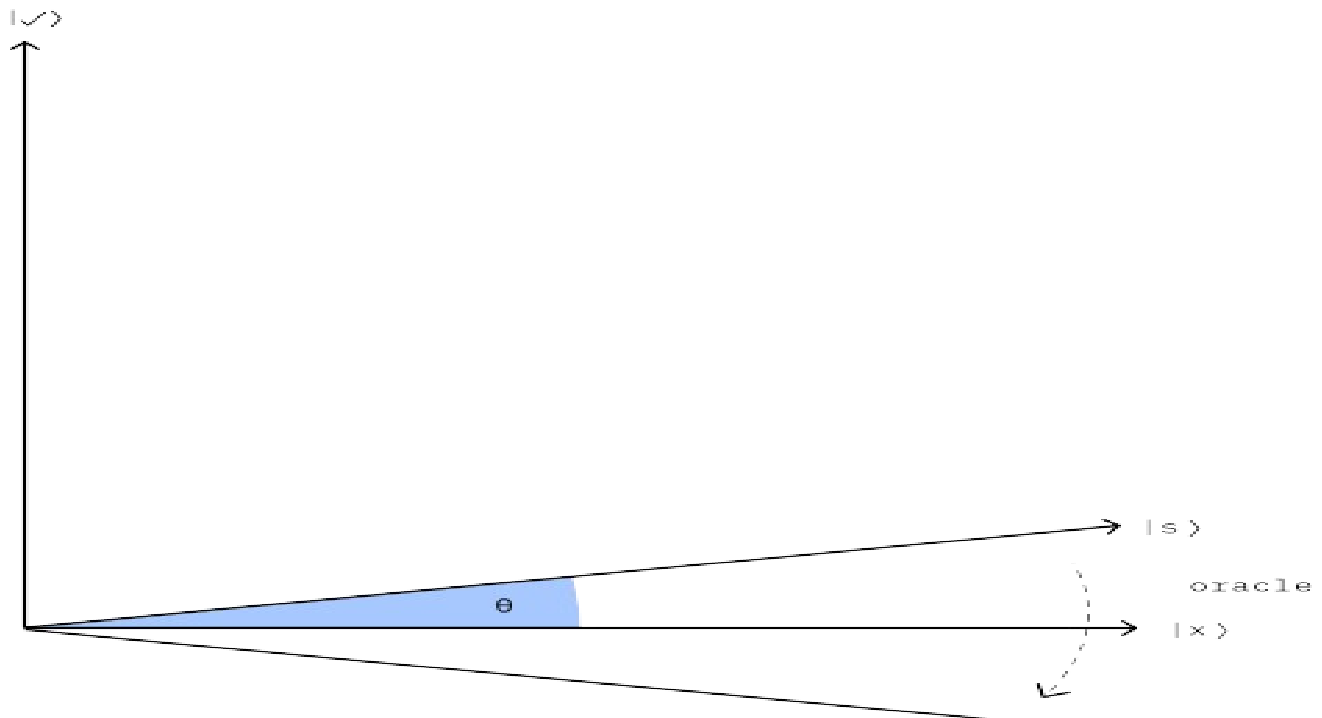
and we'll call the superposition of every other state ":

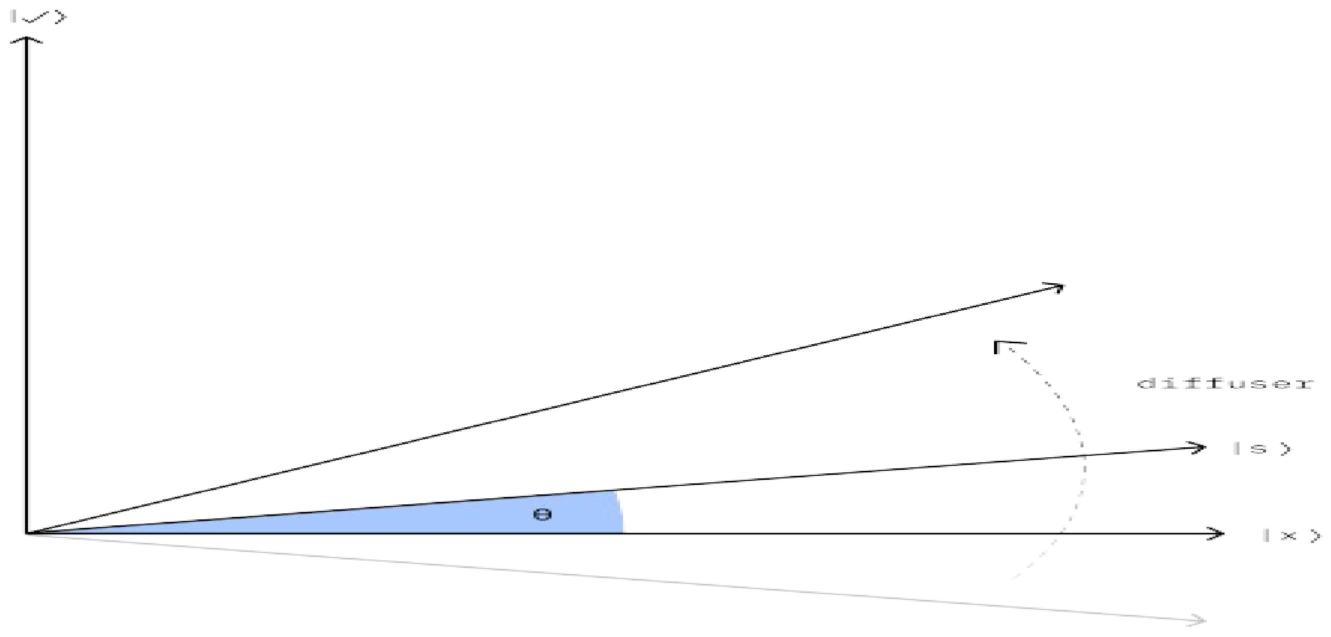



|✓⟩
|s⟩
θ
|x⟩


|✓⟩
|s⟩
θ
|x⟩

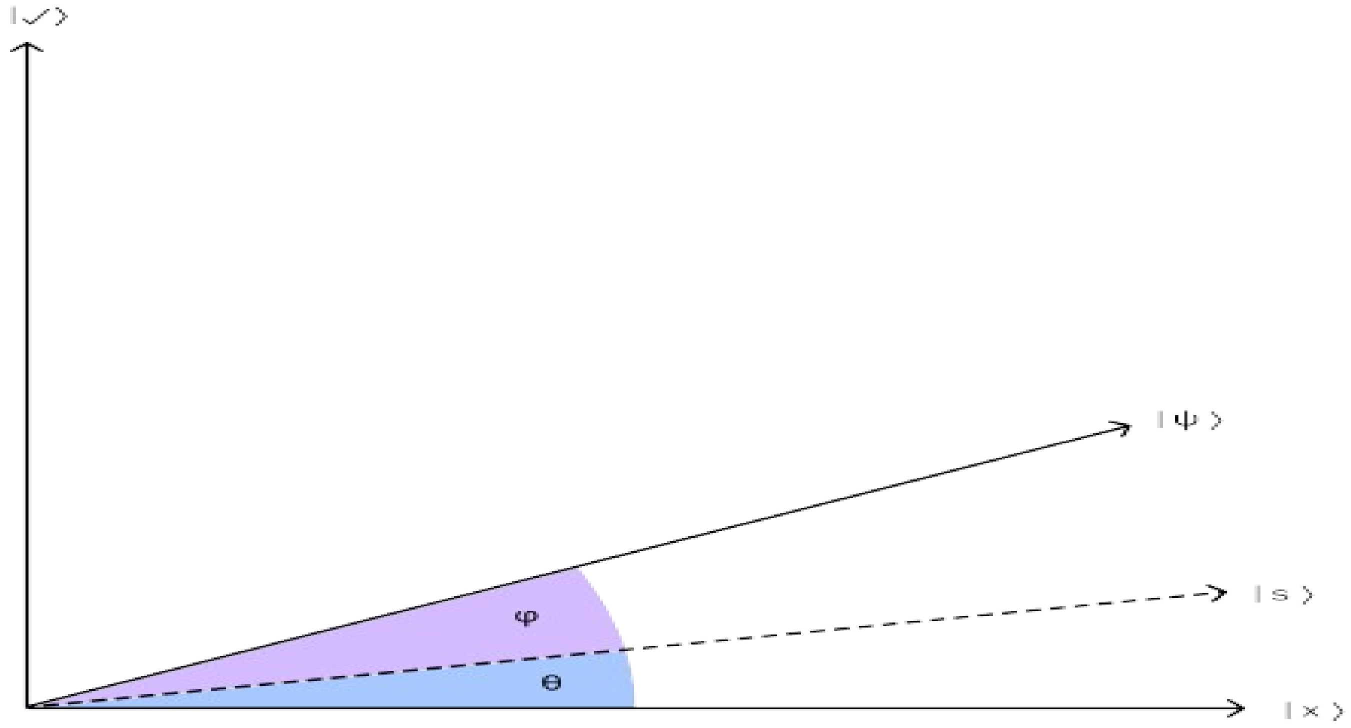

|✓⟩
|s⟩
θ
oracle
|x⟩


|✓⟩
|s⟩
θ
oracle
|x⟩

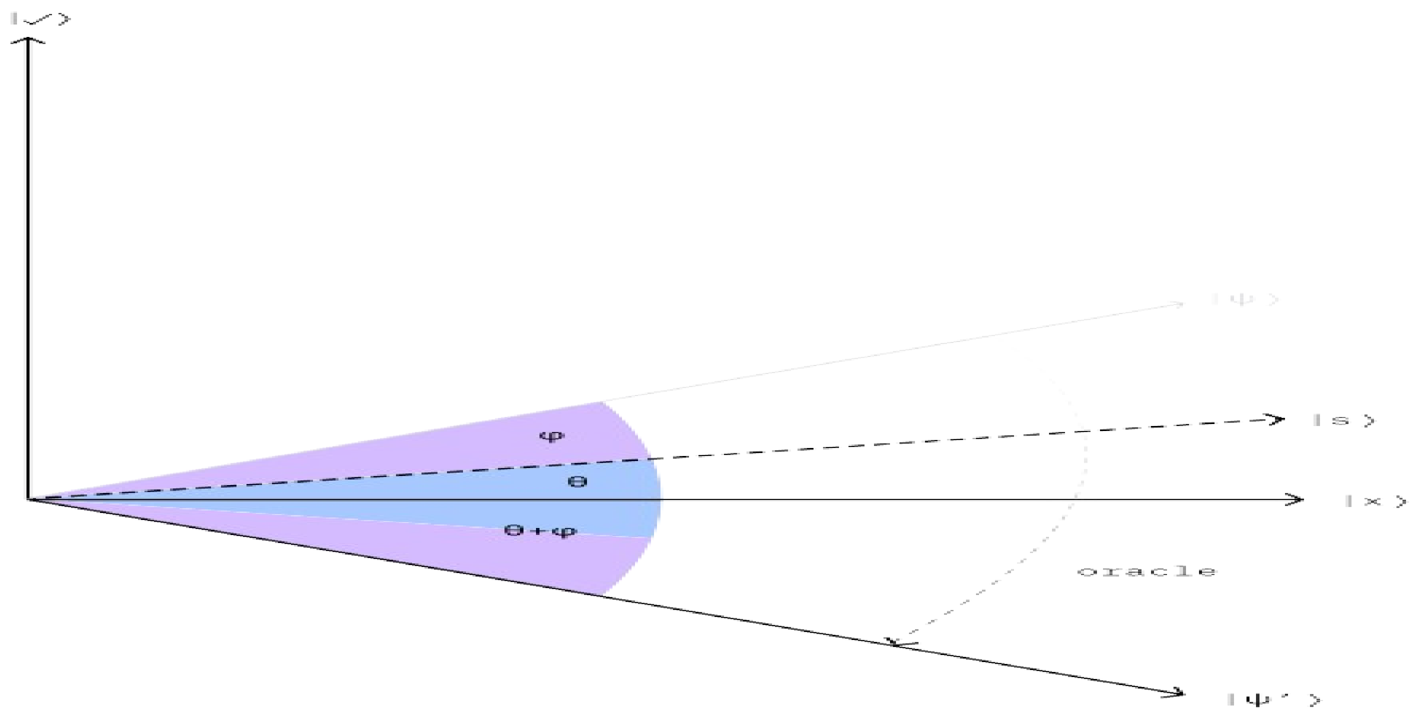


•

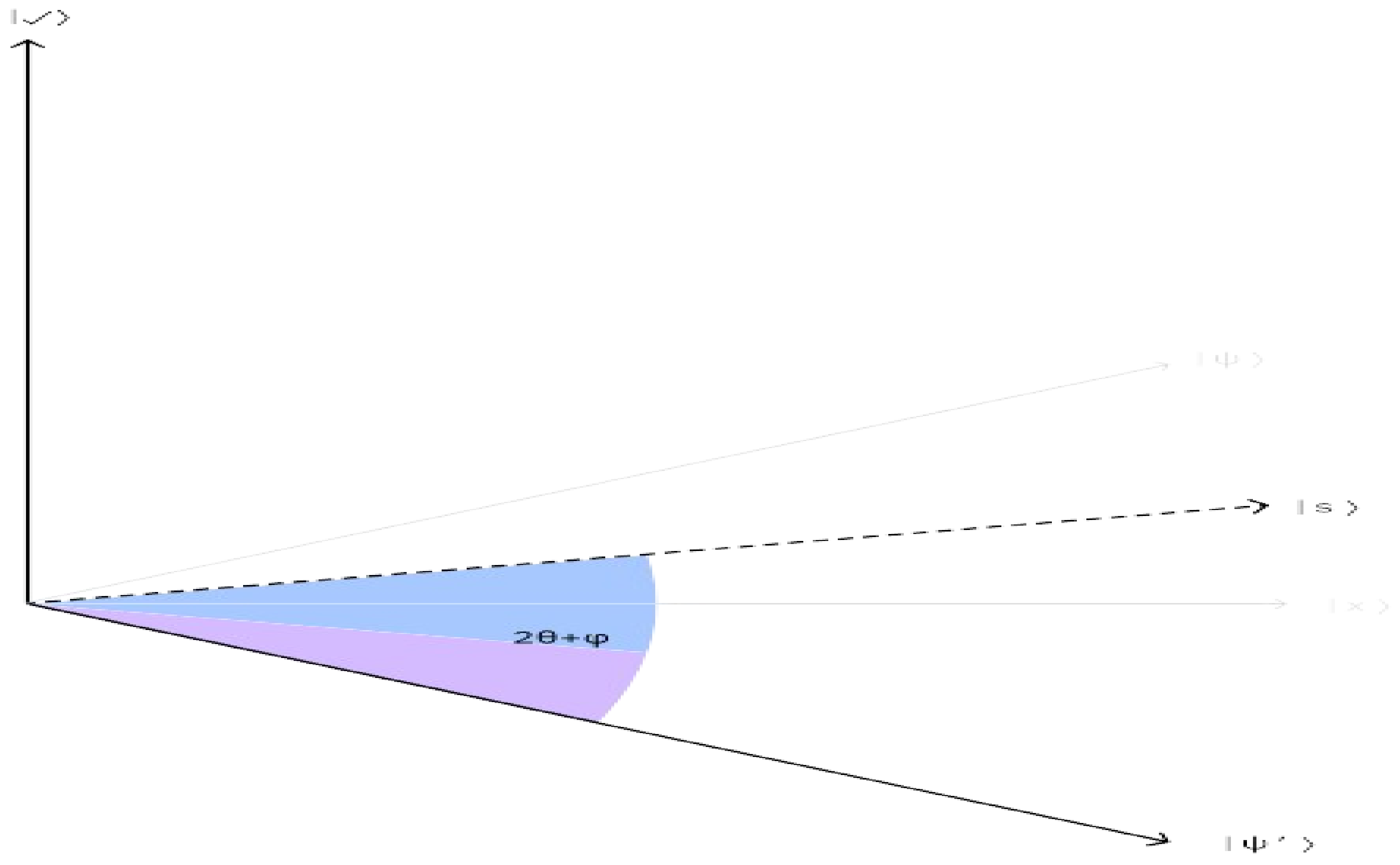
How many times do we need to query the oracle?



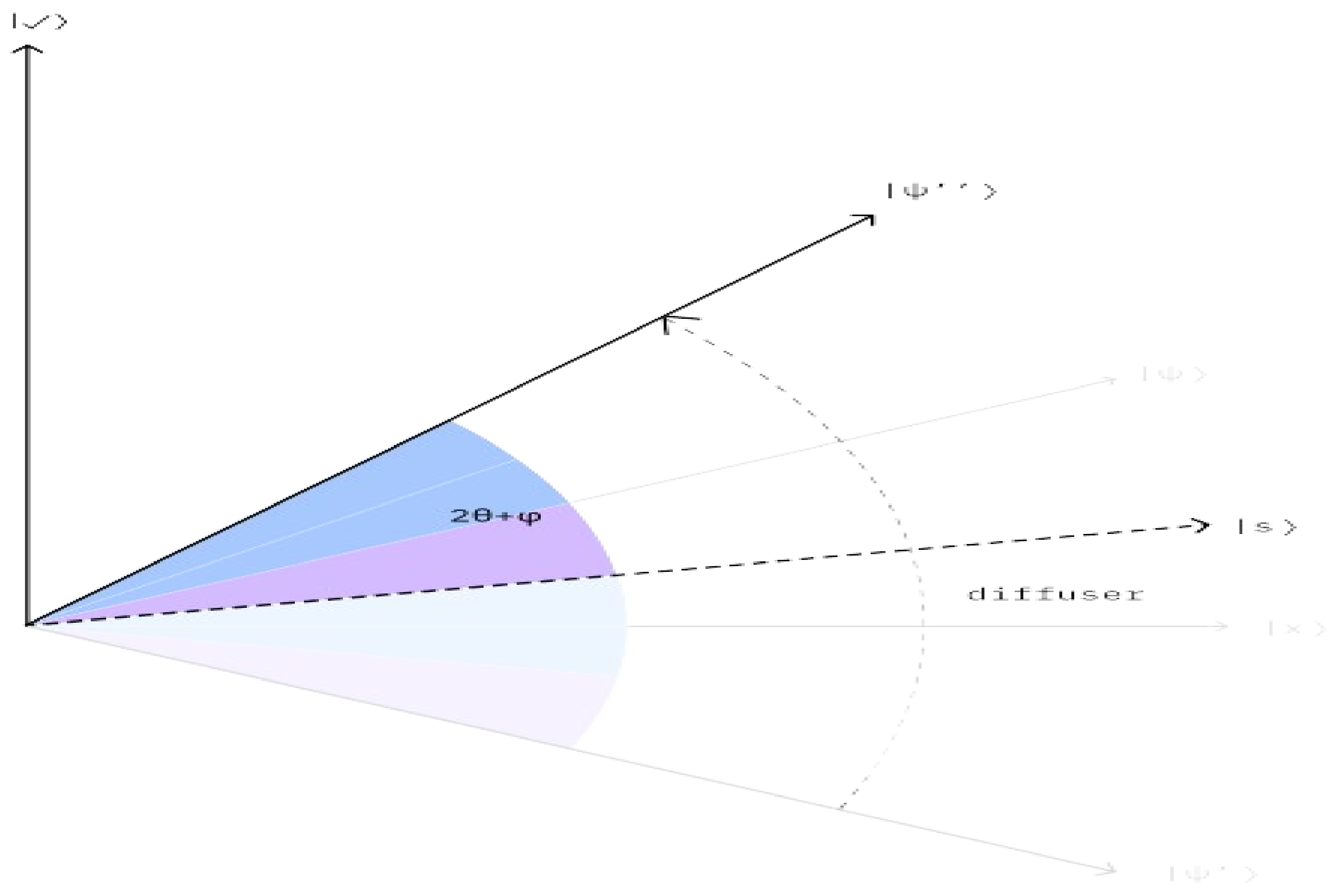
To work this out, we'll have to work how much each iteration rotates our state towards . Let's say we're somewhere in the middle of our algorithm, the state of our computer () is an angle from the starting state . The angle between and is .

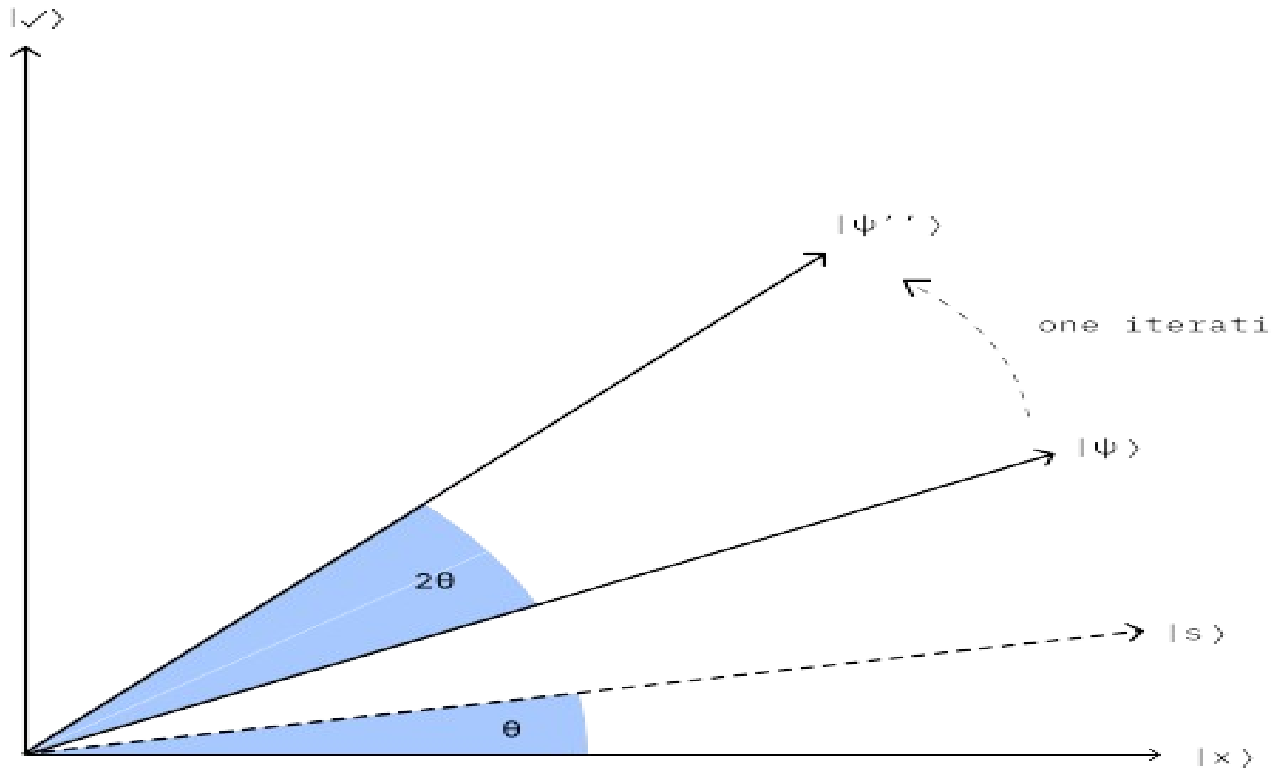


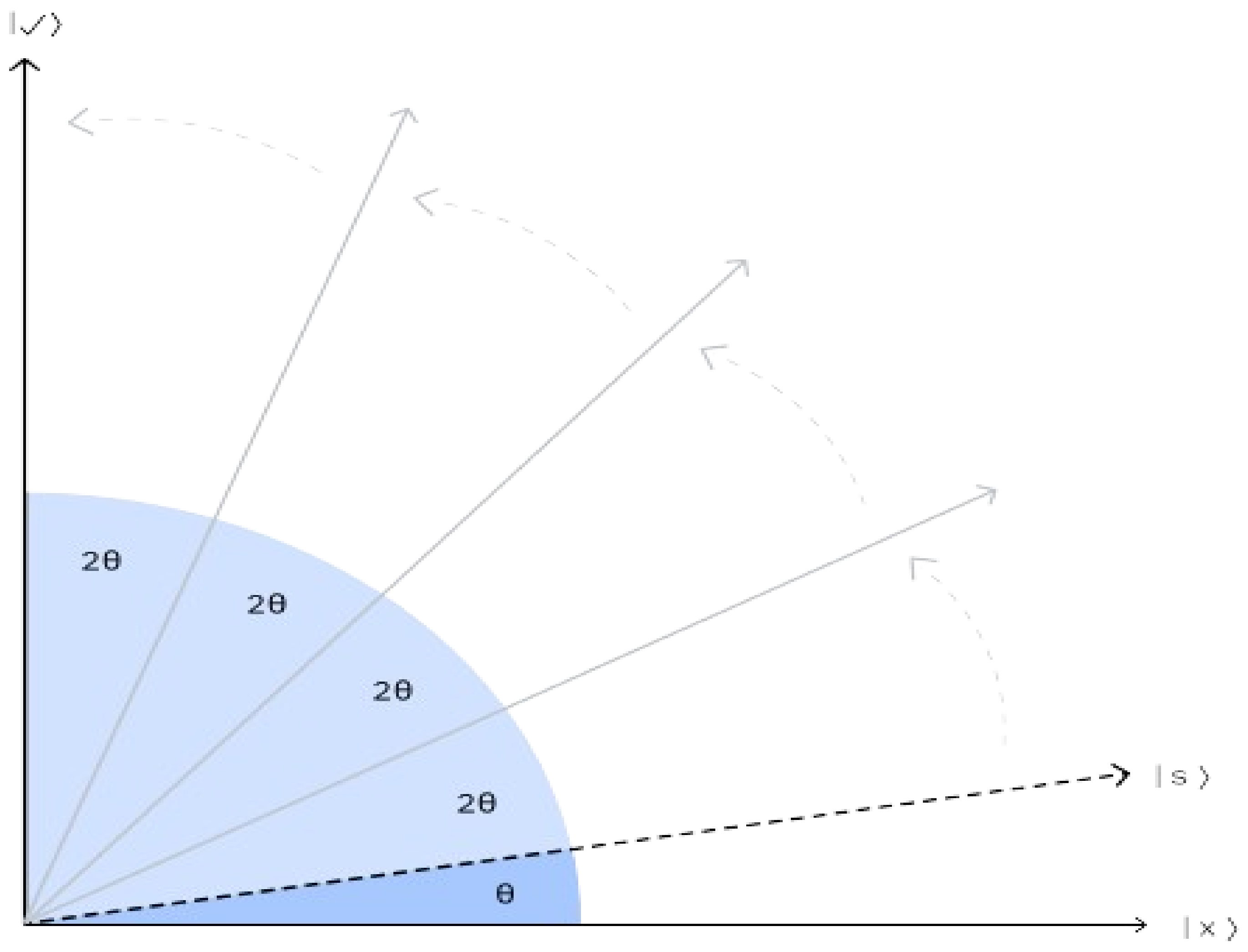
|✓⟩
|ψ⟩
φ
|s⟩
θ
|x⟩
θ+φ
oracle
|ψ'⟩



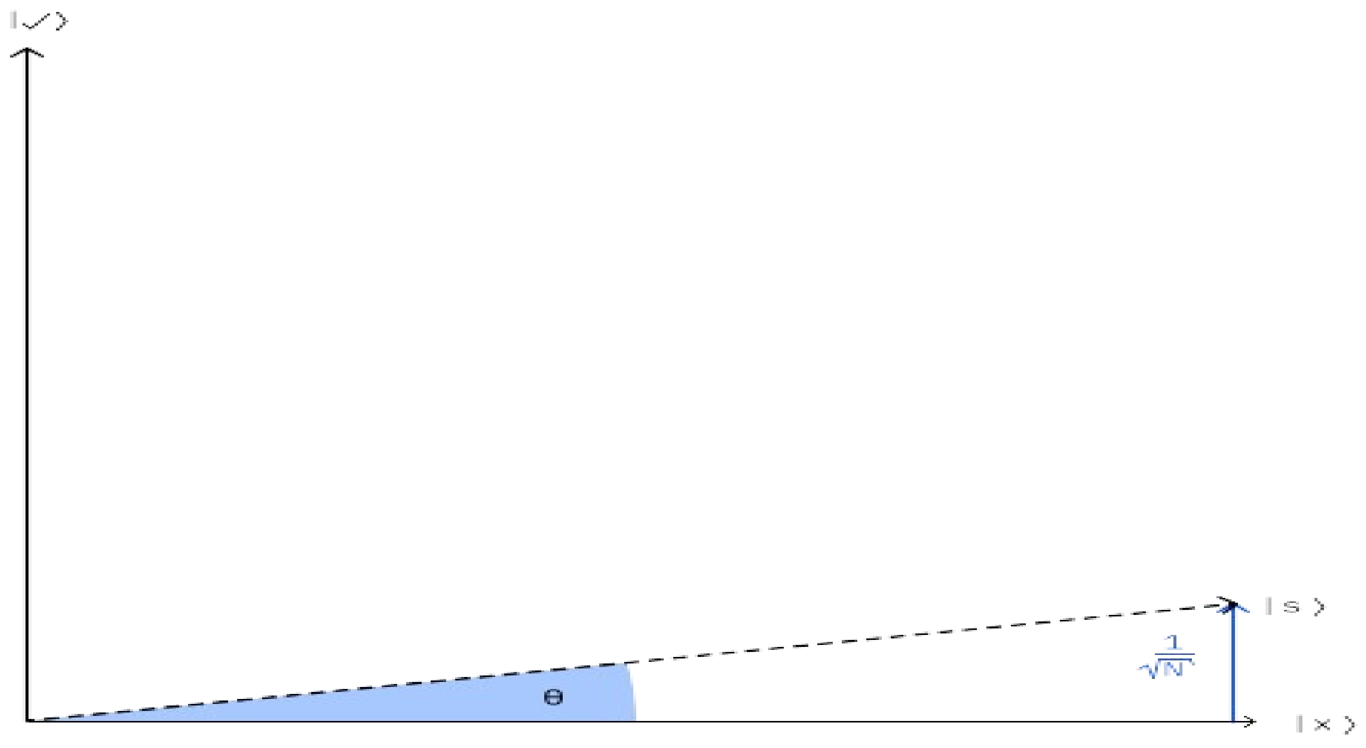
|✓⟩
|ψ⟩
|s⟩
|x⟩
2θ+φ
|ψ'⟩


|✓⟩
|ψ''⟩
|ψ⟩
2θ+φ
|s⟩
diffuser
|x⟩
|ψ'⟩

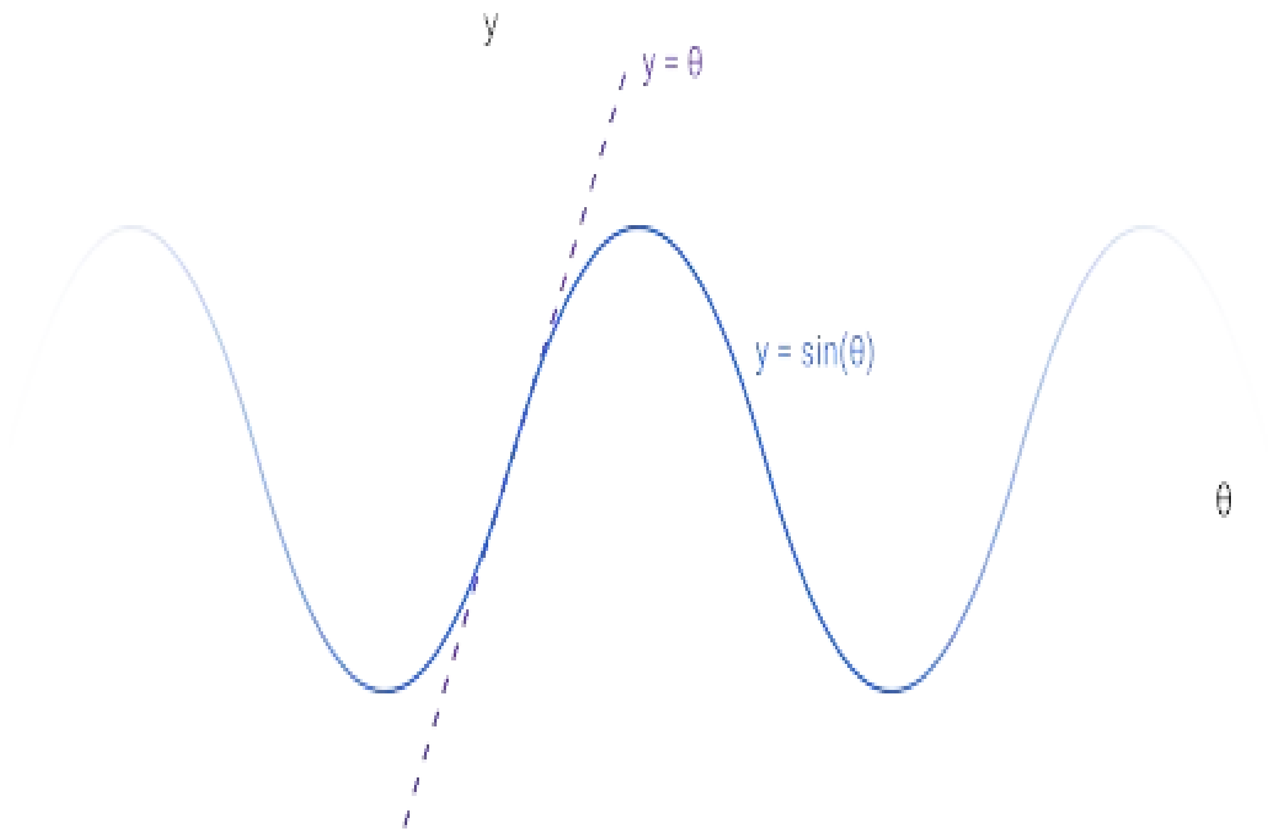

|✓⟩
|ψ''⟩
one iteration
|ψ⟩
2θ
|s⟩
θ
|x⟩

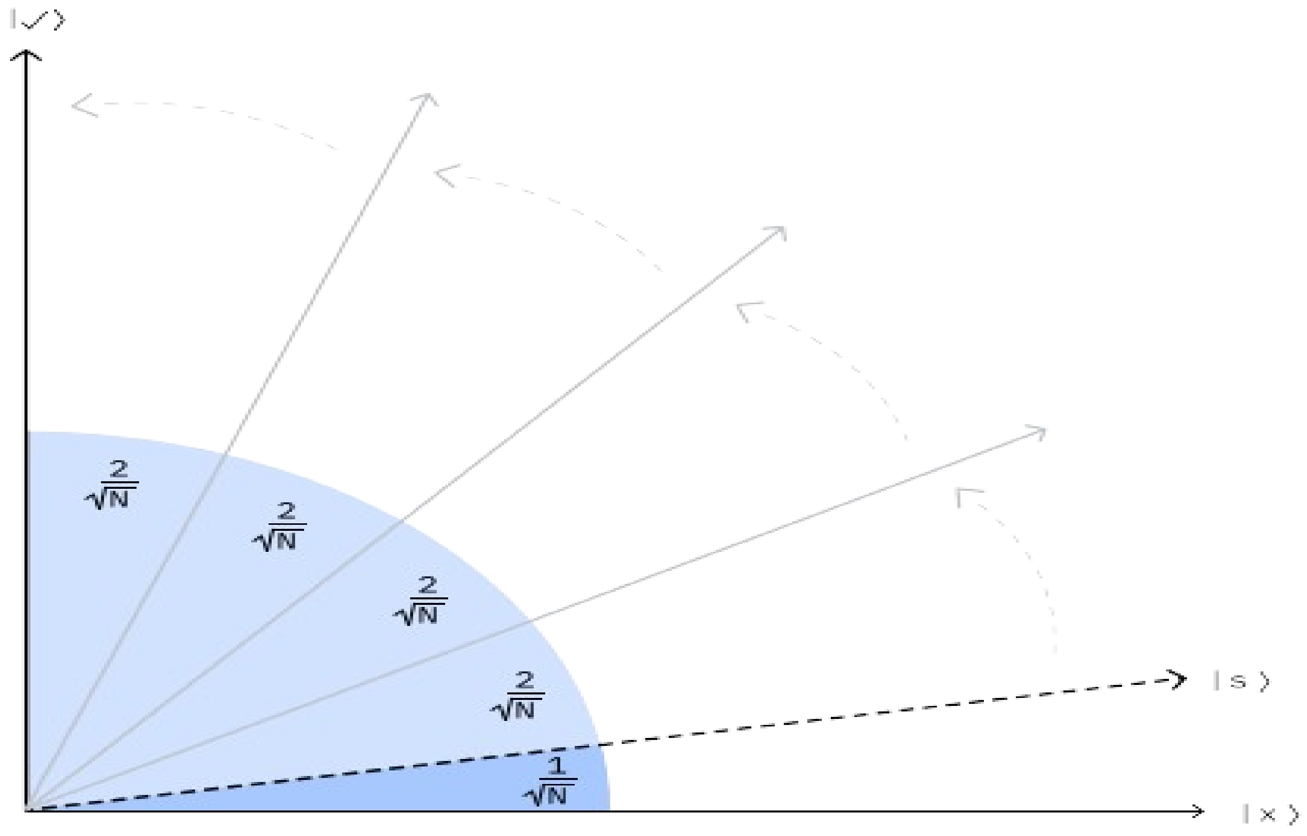


|✓⟩
2θ
2θ
2θ
2θ
θ
|s⟩
|x⟩



|✓⟩
|s⟩
$\frac{1}{\sqrt{N}}$
θ
|x⟩





-

Circuits for Grover's algorithm

To round off the chapter, we’ll create a simple circuit from scratch that implements Grover’s algorithm, and show it works. We’ll use two qubits, and we’ll start by creating an oracle circuit.

```
from qiskit import QuantumCircuit
```

The oracle

To keep things simple, we're not going to solve a real problem here. For this demonstration, we'll create a circuit that flips the phase of the state and leaves everything else unchanged. Fortunately, we already know of a two-qubit gate that does exactly that!

```
oracle = QuantumCircuit(2)
oracle.cz(0,1)  # invert phase of |11>
oracle.draw()
```

Here's a short function to show the matrix representation of this circuit:

```
def display_unitary(qc, prefix=""):
    """Simulates a simple circuit and display its matrix representation.
    Args:
        qc (QuantumCircuit): The circuit to compile to a unitary matrix
        prefix (str): Optional LaTeX to be displayed before the matrix
    Returns:
        None (displays matrix as side effect)
    """
    from qiskit import Aer
    from qiskit.visualization import array_to_latex
    sim = Aer.get_backend('aer_simulator')
    # Next, we'll create a copy of the circuit and work on
    # that so we don't change anything as a side effect
    qc = qc.copy()
    # Tell the simulator to save the unitary matrix of this circuit
    qc.save_unitary()
    unitary = sim.run(qc).result().get_unitary()
    display(array_to_latex(unitary, prefix=prefix))


display_unitary(oracle, "U_\\text{oracle}=")
```

$$ U_\text{oracle}= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ \end{bmatrix} $$

Creating the diffuser

Next we'll create a diffuser for two qubits. Remember that we want to do a reflection around the state , so let's see if we can use the tools we already have to build a circuit that does this reflection.

We've already seen that the `cz` gate does a reflection around (up to a global phase), so if we know the transformation that maps , we can:

1. Do the transformation
2. Reflect around (i.e the `cz` gate)
3. Do the transformation

We know that we can create the state from a the state by applying a H-gate to each qubit. Since the H-gate is it's own inverse, applying H-gates to each qubit also does .

```
diffuser = QuantumCircuit(2)
diffuser.h([0, 1])
diffuser.draw()
```

Now we need to work out how we transform .

So applying an X-gate to each qubit will do the transformation . Let's do that:

```
diffuser.x([0,1])
diffuser.draw()
```

Now we have the transformation , we can apply our `cz` gate and reverse the transformation.

```
diffuser.cz(0,1)
diffuser.x([0,1])
diffuser.h([0,1])
diffuser.draw()
```

Putting it together

We now have two circuits, `oracle` and `diffuser`, so we can put this together into a circuit that performs Grover's algorithm. Remember the three steps:

1. Initialise the qubits to the state
2. Perform the oracle
3. Perform the diffuser

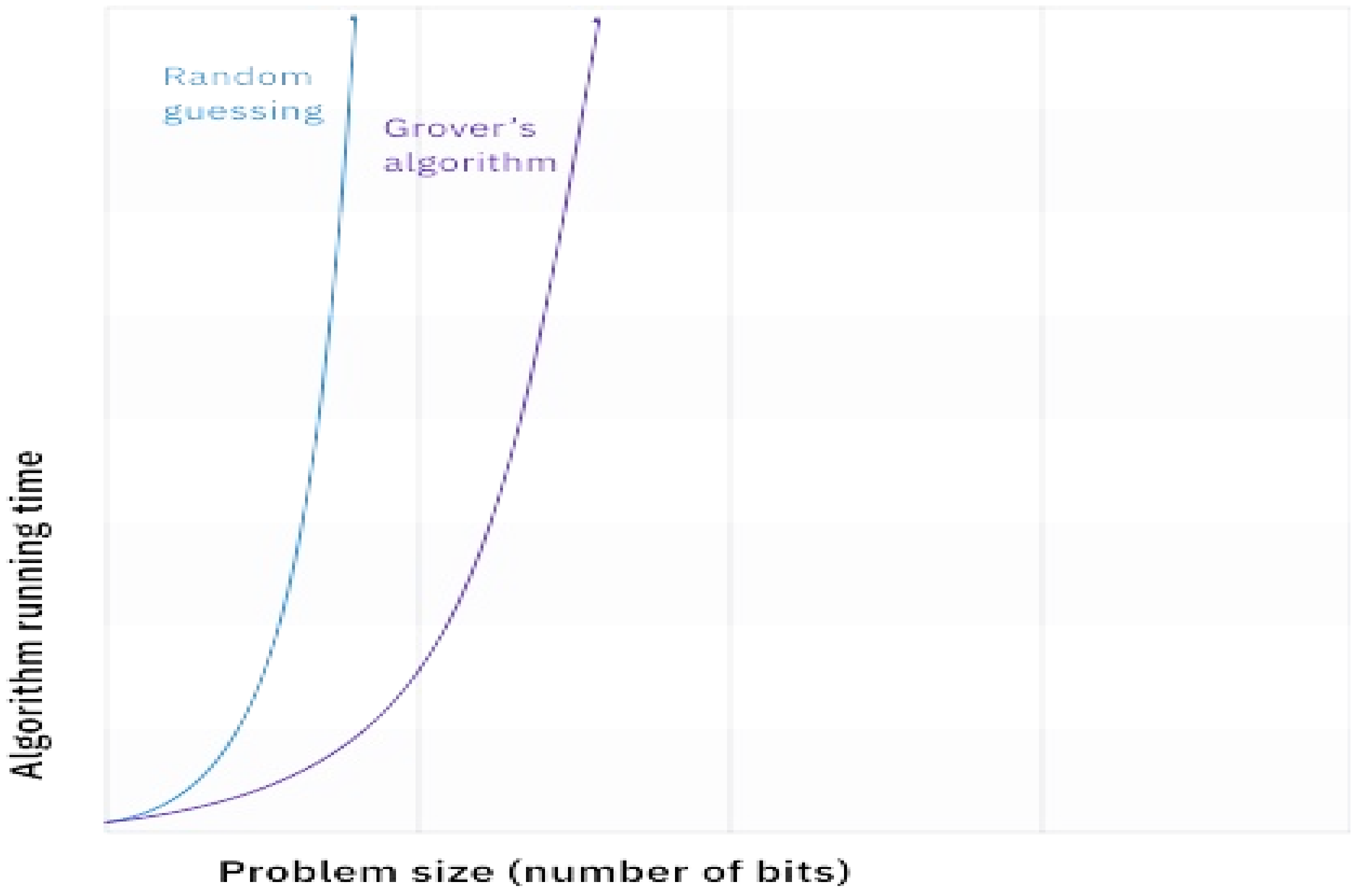
```
grover = QuantumCircuit(2)
grover.h([0,1])  # initialise |s>
grover = grover.compose(oracle)
grover = grover.compose(diffuser)
grover.measure_all()
grover.draw()
```

And when we simulate, we can see a 100% probability of measuring , which was the solution to our oracle!

```
from qiskit import Aer
sim = Aer.get_backend('aer_simulator')
sim.run(grover).result().get_counts()
```

```
 {'11': 1024}
```

SAT problems are hard

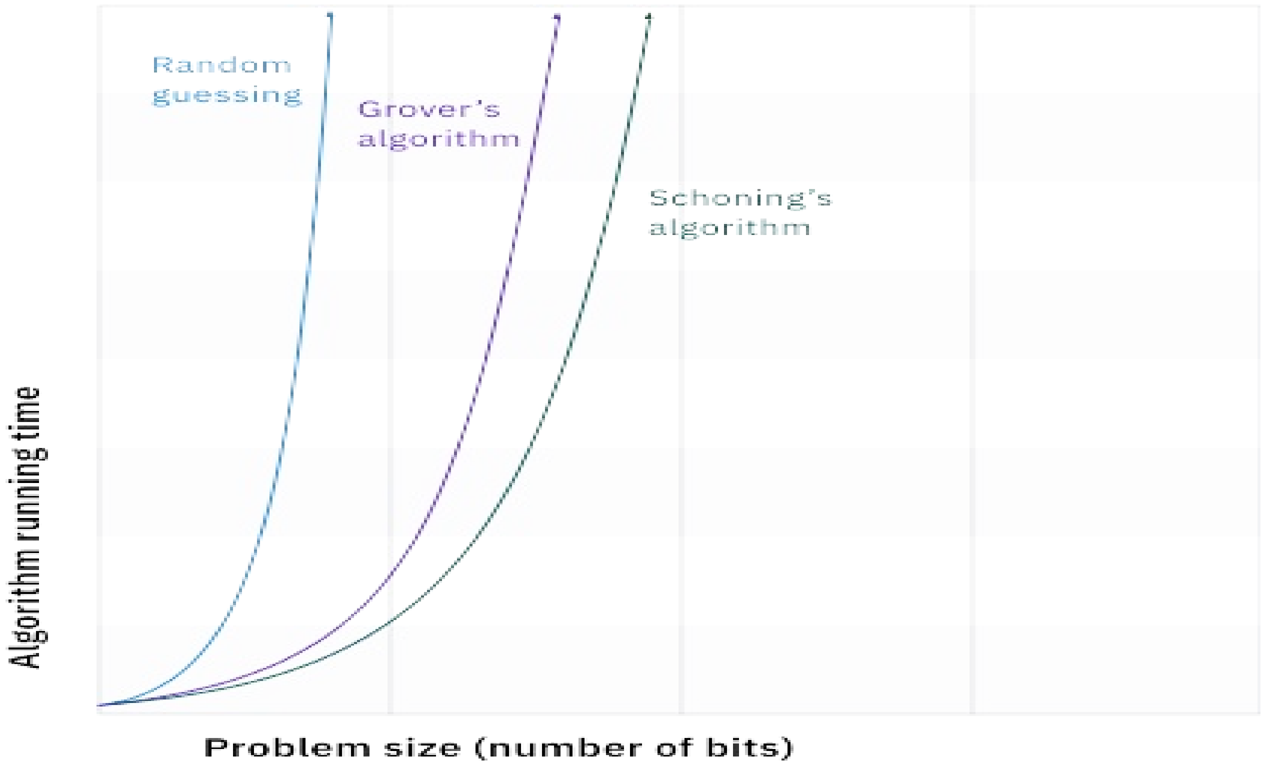


Random guessing grows linearly with the number of entries in the database, which isn't actually too bad (although we know we can do much better), but we usually measure how algorithms grow by their input length in *bits*. So how do these two connect? Each extra variable (bit) in our SAT problem *doubles* the number of possible solutions (i.e. entries to our database), so the search space grows exponentially with the number of bits.

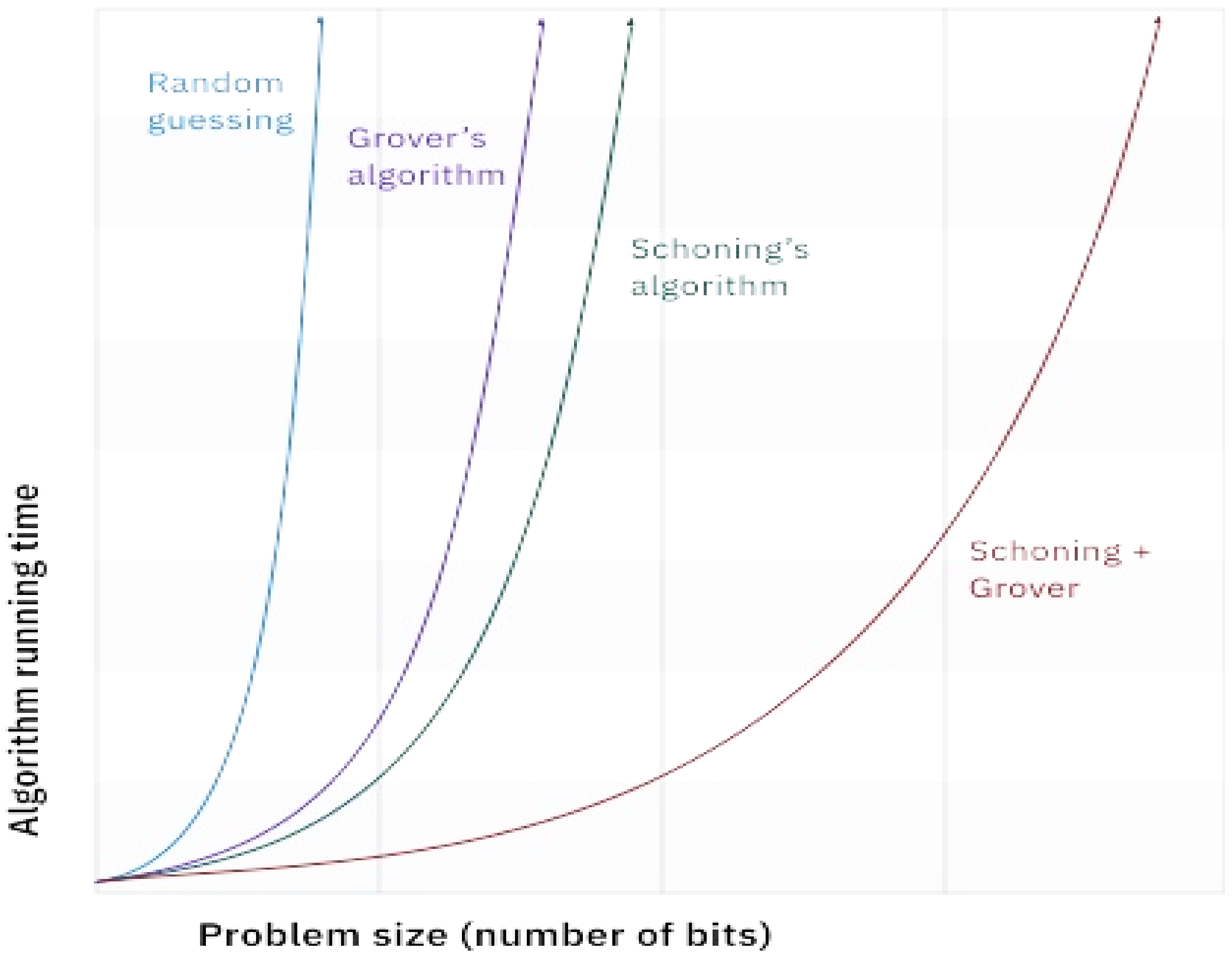
Since random guessing grows linearly with , the running time will grow by roughly .

Making use of structure

So far, we've treated SAT problems as if they're completely unstructured, but unlike the unsorted phonebook, we *do* have some clues that will help us in our search. A SAT problem isn't a black box, but a set of individual clauses, and we can use these clauses to home in on a correct answer. We won't get anything nearly as efficient as binary search, but it's still much better than random guessing. One (classical) algorithm that uses the structure of SAT problems is Schöning's algorithm.



Like random guessing, Schöning's algorithm chooses an input at random and checks if it works. But unlike random guessing, it doesn't just throw this string away. Instead, it picks an unsatisfied clause and toggles a bit in the string to satisfy that clause. Annoyingly, this new string might unsatisfy a different, previously-satisfied clause, but on average it's beneficial to keep toggling bits in this manner a few times. If the initial guess was close enough, there's a fair chance we'll stumble upon the correct solution. If not, then after some number of steps, the computer starts again with a new completely random guess. It turns out for 3-SAT (although not (>3)-SAT), this algorithm grows with roughly , which not only beats random guessing, but also beats Grover's algorithm!



It may not be obvious at first glance, but we can actually combine Grover and Schöning's algorithms to get something even better than either individually. If you create a circuit that carries out the bit-toggling part of Schöning's algorithm, you can use this as the oracle and use Grover's algorithm to find the best "initial guess". We won't go into it in this course, but it's a fun project to investigate it!

Grover's Algorithm

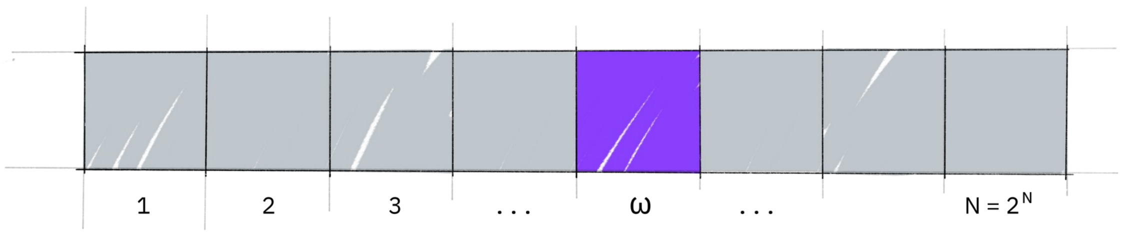
1. Introduction

You have likely heard that one of the many advantages a quantum computer has over a classical computer is its superior speed searching databases. Grover's algorithm demonstrates this capability. This 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.

Unstructured Search

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 $\sqrt{N}$ 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.

Creating an Oracle

For the examples in this textbook, our 'database' is comprised of all the possible computational basis states our qubits can be in. For example, if we have 3 qubits, our list is the states $|000\rangle, |001\rangle, \dots \vee 111\rangle$ (i.e the states $|0\rangle \rightarrow |7\rangle$).

Grover’s algorithm solves oracles that add a negative phase to the solution states. I.e. for any state $x\rangle$ in the computational basis:

$$U_\omega \vee x\rangle = \begin{cases} x\rangle \text{ if } x \neq \omega \\ - \vee x\rangle \text{ if } x = \omega \end{cases}$$

This oracle will be a diagonal matrix, where the entry that correspond to the marked item will have a negative phase. For example, if we have three qubits and $\omega = 101$, our oracle will have the matrix:

$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \leftarrow \omega = 101$$

¿

What makes Grover’s algorithm so powerful is how easy it is to convert a problem to an oracle of this form. There are many computational problems in which it’s difficult to *find* a solution, but relatively easy to *verify* a solution. For example, we can easily verify a solution to a sudoku by checking all the rules are satisfied. For these problems, we can create a function f that takes a proposed solution x, and returns $f(x) = 0$ if x is not a solution ($x \neq \omega$) and $f(x) = 1$ for a valid solution ($x = \omega$). Our oracle can then be described as:

$$U_\omega |x\rangle = (-1)^{f(x)} |x\rangle$$

and the oracle's matrix will be a diagonal matrix of the form:

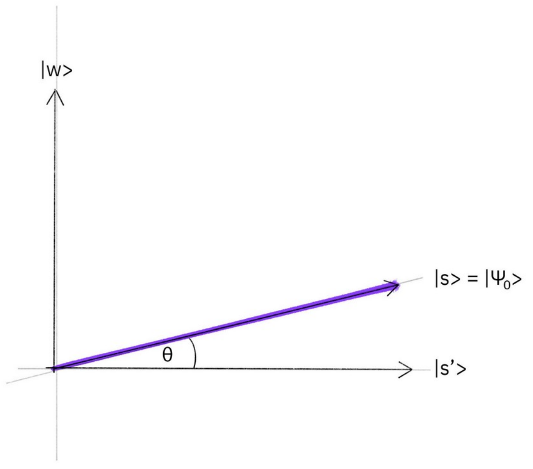
$$U_\omega = \begin{bmatrix} (-1)^{f(0)} & 0 & \cdots & 0 \\ 0 & (-1)^{f(1)} & \cdots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & (-1)^{f(2^n-1)} \end{bmatrix}$$

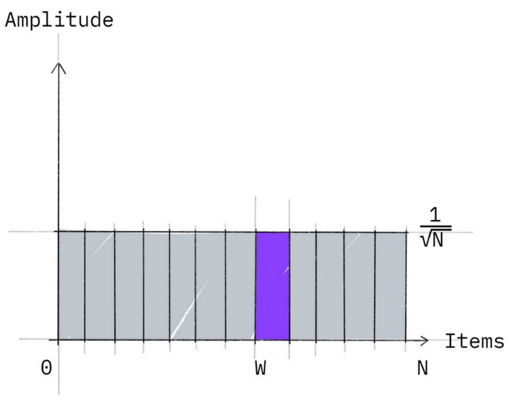
For the next part of this chapter, we aim to teach the core concepts of the algorithm. We will create example oracles where we know ω beforehand, and not worry ourselves with whether these oracles are useful or not. At the end of the chapter, we will cover a short example where we create an oracle to solve a problem (sudoku).

Amplitude Amplification

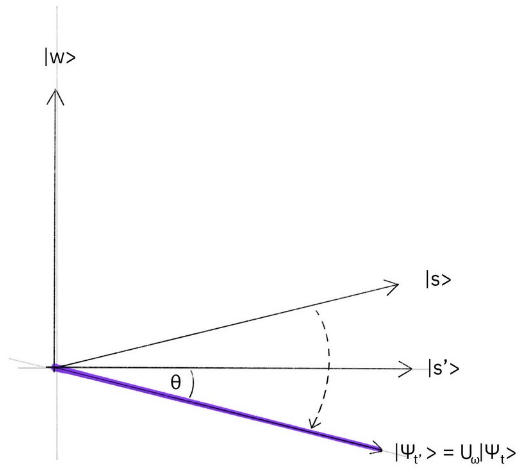
So how does the algorithm work? Before looking at the list of items, we have no idea where the marked item is. Therefore, any guess of its location is as good as any other, which can be expressed in terms of a uniform superposition: $\left| s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \square \right| x\rangle$.
If at this point we were to measure in the standard basis $\{\vee x\rangle\}$, this superposition would collapse, according to the fifth quantum law, to any one of the basis states with the same probability of $\frac{1}{N} = \frac{1}{2^n}$. Our chances of guessing the right value w is therefore 1 in 2^n, as could be expected. Hence, on average we would need to try about $N/2 = 2^{n-1}$ times to guess the correct item.
Enter the procedure called amplitude amplification, which is how a quantum computer significantly enhances this probability. This procedure stretches out (amplifies) the amplitude of the marked item, which shrinks the other items' amplitude, so that measuring the final state will return the right item with near-certainty.
This algorithm has a nice geometrical interpretation in terms of two reflections, which generate a rotation in a two-dimensional plane. The only two special states we need to consider are the winner $w\rangle$ and the uniform superposition $s\rangle$. These two vectors span a two-dimensional plane in the vector space C^N. They are not quite perpendicular because $w\rangle$ occurs in the superposition with amplitude $N^{-1/2}$ as well. We can, however, introduce an additional state $s'\rangle$ that is in the span of these two vectors, which is perpendicular to $w\rangle$ and is obtained from $s\rangle$ by removing $w\rangle$ and rescaling.
Step 1: The amplitude amplification procedure starts out in the uniform superposition $s\rangle$, which is easily constructed from $|s\rangle = H^{\otimes n} | 0\rangle^n$.

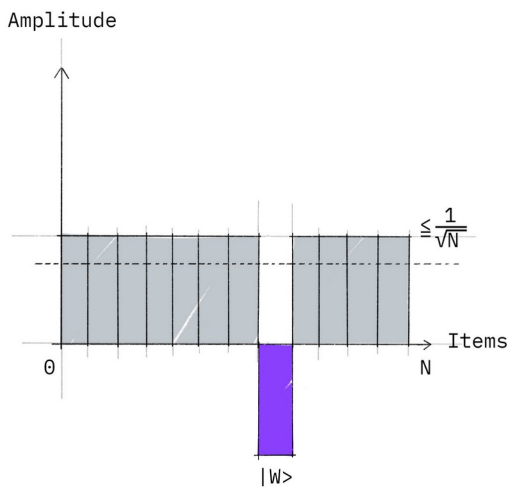
The left





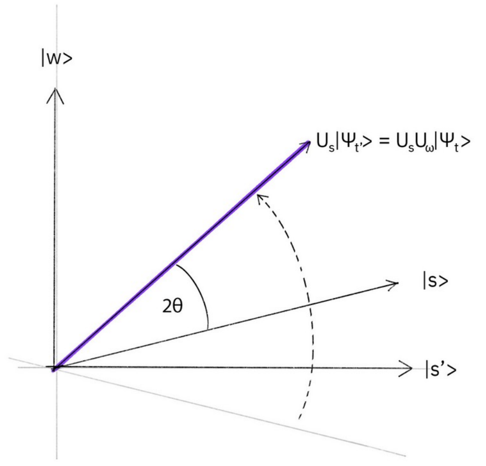
graphic corresponds to the two-dimensional plane spanned by perpendicular vectors $w\rangle$ and $s'\rangle$ which allows to express the initial state as $|s\rangle = \sin\theta | w\rangle + \cos\theta \vee s'\rangle$, where $\theta = \arcsin\langle s \vee w\rangle = \arcsin\frac{1}{\sqrt{N}}$. The right graphic is a bar graph of the amplitudes of the state $s\rangle$.
Step 2: We apply the oracle reflection U_f to the state $s\rangle$.

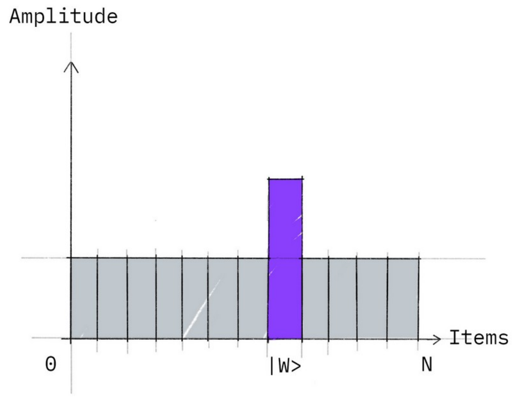




Geometrically this corresponds to a reflection of the state $s\rangle$ about $s'\rangle$. This transformation means that the amplitude in front of the $w\rangle$ state becomes negative, which in turn means that the average amplitude (indicated by a dashed line) has been lowered.
Step 3: We now apply an additional reflection (U_s) about the state $s\rangle$: $U_s = 2|s\rangle\langle s| - 1$. This transformation maps the state to $U_s U_f \vee s\rangle$ and completes the transformation.

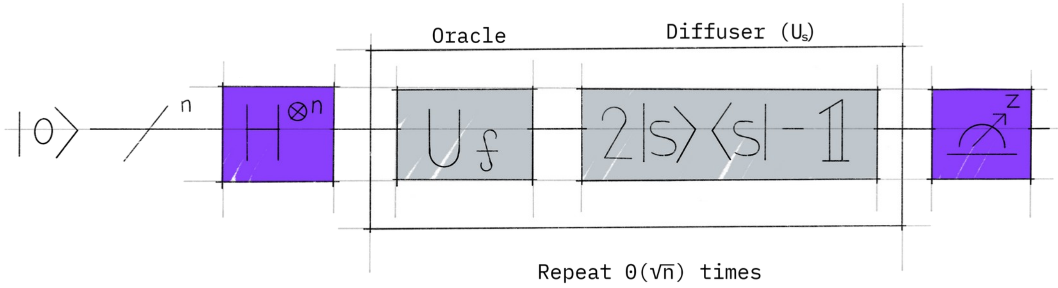
Two





reflections always correspond to a rotation. The transformation $U_s U_f$ rotates the initial state $s\rangle$ closer towards the winner $w\rangle$. 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 $w\rangle$ to roughly three times its original value, while it decreases the other amplitudes. We then go to **step 2** to repeat the application. This procedure will be repeated several times to zero in on the winner.
After t steps we will be in the state $\psi_t\rangle$ where: $\left|\psi_t\right\rangle = \left(U_s U_f\right)^t \left| s\right\rangle$.
How many times do we need to apply the rotation? It turns out that roughly $\sqrt{N}$ rotations suffice. This becomes clear when looking at the amplitudes of the state $\psi\rangle$. We can see that the amplitude of $w\rangle$ grows linearly with the number of applications $\sim t N^{-1/2}$. However, since we are dealing with amplitudes and not probabilities, the vector space's dimension enters as a square root. Therefore it is the amplitude, and not just the probability, that is being amplified in this procedure.
In the case that there are multiple solutions, M, it can be shown that roughly $\sqrt{(N/M)}$ rotations will suffice.

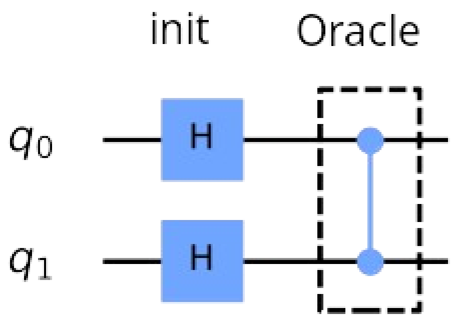
2.


Oracle
Diffuser (U_s)
$|0\rangle$
n
$H^{\otimes n}$
U_f
$2|s\rangle\langle s| - \mathbb{1}$
z
Repeat $O(\sqrt{n})$ times

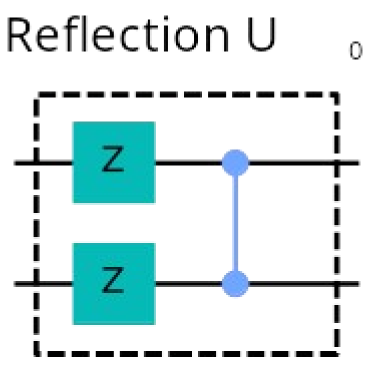
Example: 2 Qubits
Let's first have a look at the case of Grover's algorithm for $N=4$ which is realized with 2 qubits. In this particular case, only one rotation is required to rotate the initial state $s\rangle$ to the winner $w\rangle$[3]:
Following the above introduction, in the case $N=4$ we have
$\theta = \arcsin \frac{1}{2} = \frac{\pi}{6}$.
We will now follow through an example using a specific oracle.
Oracle for $\omega\rangle = 11\rangle$
Let's look at the case $w\rangle = 11\rangle$. The oracle U_ω in this case acts as follows:
$U_\omega \vee s\rangle = U_\omega \frac{1}{2}$
or:

$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

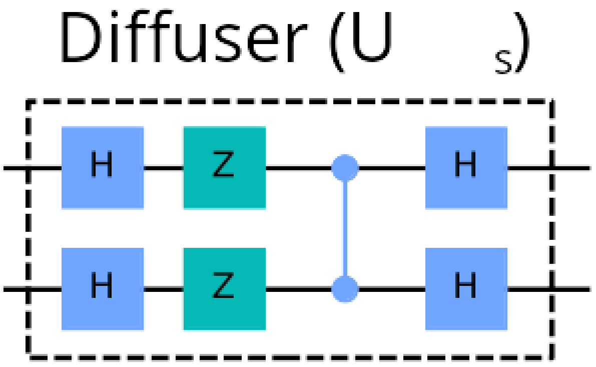
which you may recognise as the controlled-Z gate. Thus, for this example, our oracle is simply the controlled-Z gate:



Reflection U_s
In order to complete the circuit we need to implement the additional reflection $U_s = 2|s\rangle\langle s| - 1$. Since this is a reflection about $s\rangle$, we want to add a negative phase to every state orthogonal to $s\rangle$.
One way we can do this is to use the operation that transforms the state $|s\rangle \rightarrow |0\rangle$, which we already know is the Hadamard gate applied to each qubit:
$H^{\otimes n}$
Then we apply a circuit that adds a negative phase to the states orthogonal to $0\rangle$:
i.e. the signs of each state are flipped except for $00\rangle$. As can easily be verified, one way of implementing U_0 is the following circuit:

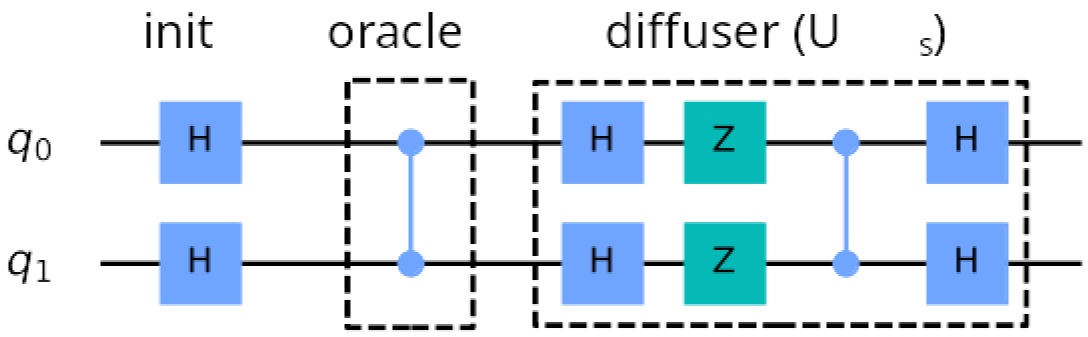


Circuit for reflection around |0>
Finally, we do the operation that transforms the state $|0\rangle \rightarrow |s\rangle$ (the H-gate again):
$H^{\otimes n} U_0 H^{\otimes n} = U_s$
The complete circuit for U_s looks like this:



Circuit for reflection around |s>
Full Circuit for $w\rangle = 11\rangle$
Since in the particular case of $N=4$ only one rotation is required we can combine the above components to build the full circuit for Grover's algorithm for the case $w\rangle = 11\rangle$:

2.1 Qiskit



Implementation
We now implement Grover's algorithm for the above case of 2 qubits for $w\rangle = 11\rangle$.

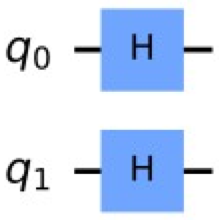
```
#initialization
import matplotlib.pyplot as plt
import numpy as np

# importing Qiskit
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit import QuantumCircuit, ClassicalRegister,
QuantumRegister
from qiskit.providers.ibmq import least_busy

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

We start by preparing a quantum circuit with two qubits:

```
n = 2
grover_circuit = QuantumCircuit(n)
```

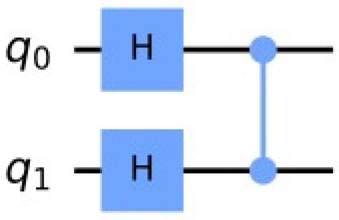
Then we simply need to write out the commands for the circuit depicted above. First, we need to initialize the state $s\rangle$. Let's create a general function (for any number of qubits) so we can use it again later:

```
def initialize_s(qc, qubits):
    """Apply a H-gate to 'qubits' in qc"""
    for q in qubits:
        qc.h(q)
    return qc
grover_circuit = initialize_s(grover_circuit, [0,1])
grover_circuit.draw()
```



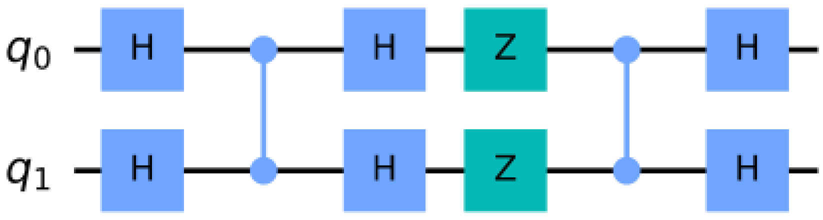
Apply the Oracle for . This oracle is specific to 2 qubits:

```
grover_circuit.cz(0,1) # Oracle
grover_circuit.draw()
```



We now want to apply the diffuser (U_s). As with the circuit that initializes $s\rangle$, we'll create a general diffuser (for any number of qubits) so we can use it later in other problems.

```
# Diffusion operator (U_s)
grover_circuit.h([0,1])
grover_circuit.z([0,1])
grover_circuit.cz(0,1)
grover_circuit.h([0,1])
grover_circuit.draw()
```



This is our finished circuit.

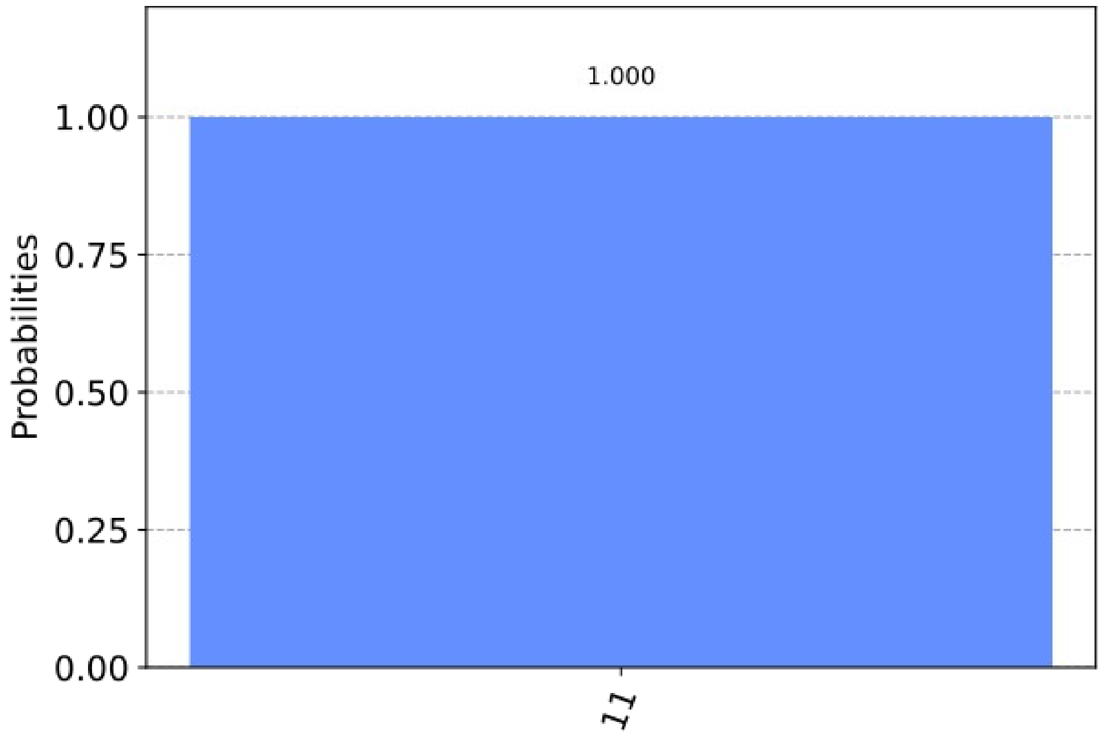
2.1.1 Experiment with Simulators

Let's run the circuit in simulation. First, we can verify that we have the correct statevector:

```
sim = Aer.get_backend('aer_simulator')
# we need to make a copy of the circuit with the 'save_statevector'
# instruction to run on the Aer simulator
grover_circuit_sim = grover_circuit.copy()
grover_circuit_sim.save_statevector()
qobj = assemble(grover_circuit_sim)
result = sim.run(qobj).result()
statevec = result.get_statevector()
from qiskit_textbook.tools import vector2latex
vector2latex(statevec, pretext="|\\psi\\rangle =")
<IPython.core.display.Math object>
```

As expected, the amplitude of every state that is not 11⟩ is 0, this means we have a 100% chance of measuring 11⟩:

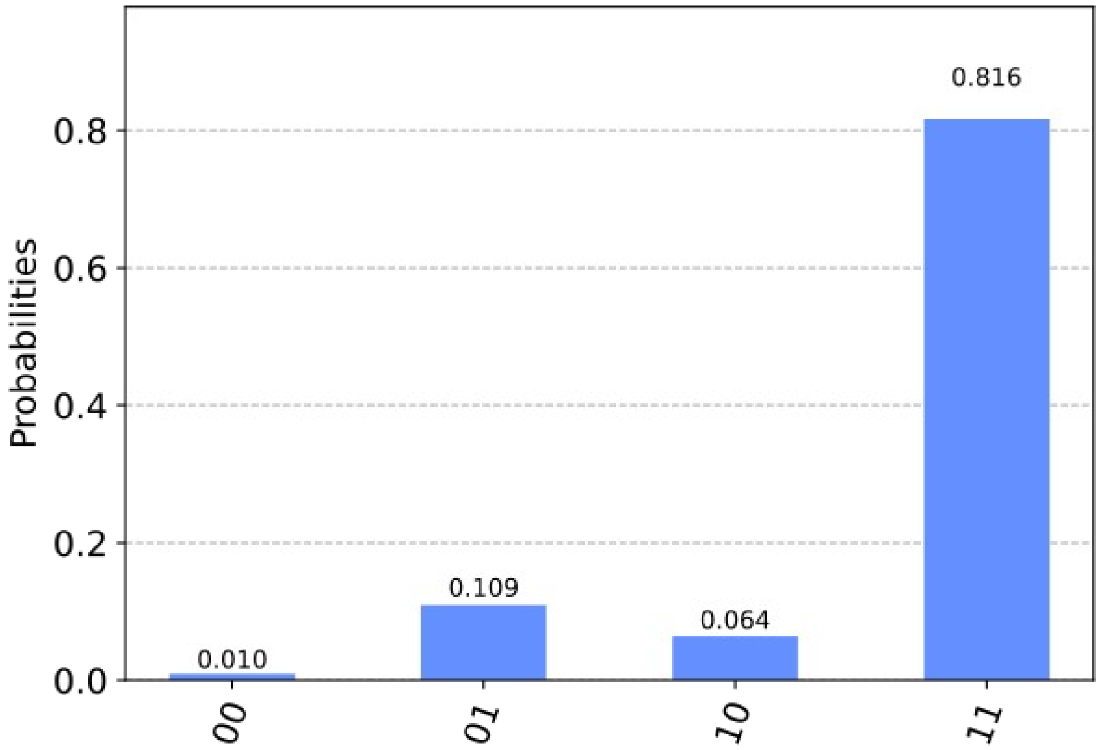
```
grover_circuit.measure_all()

aer_sim = Aer.get_backend('aer_simulator')
qobj = assemble(grover_circuit)
result = aer_sim.run(qobj).result()
counts = result.get_counts()
plot_histogram(counts)
```



2.1.2 Experiment with Real Devices

We can run the circuit a real device as below.

```
# Load IBM Q account and get the least busy backend device
provider = IBMQ.load_account()
provider = IBMQ.get_provider("ibm-q")
device = least_busy(provider.backends(filters=lambda x:
x.configuration().n_qubits >= 3 and
                                       not x.configuration().simulator
and x.status().operational==True))
print("Running on current least busy device: ", device)
Running on current least busy device:  ibmq_mumbai
# Run our circuit on the least busy backend. Monitor the execution
of the job in the queue
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_circuit, device,
optimization_level=3)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)
Job Status: job has successfully run
# Get the results from the computation
results = job.result()
answer = results.get_counts(grover_circuit)
plot_histogram(answer)
```

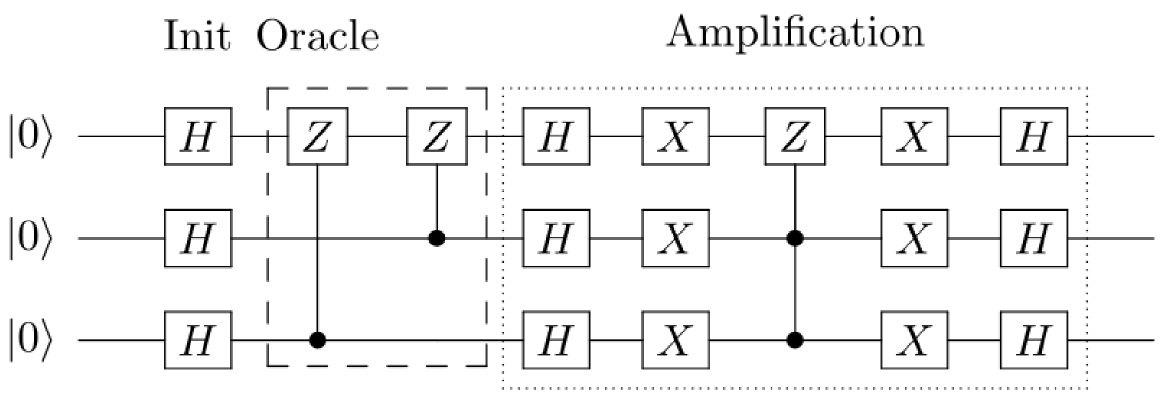


We confirm that in the majority of the cases the state 11⟩ is measured. The other results are due to errors in the quantum computation.

3. Example: 3 Qubits

We now go through the example of Grover's algorithm for 3 qubits with two marked states 101⟩ and 110⟩, following the implementation found in Reference [2]. The quantum circuit to solve the problem using a phase oracle is:

Note that since there are 2



solutions and 8 possibilities, we will only need to run one iteration (steps 2 & 3).

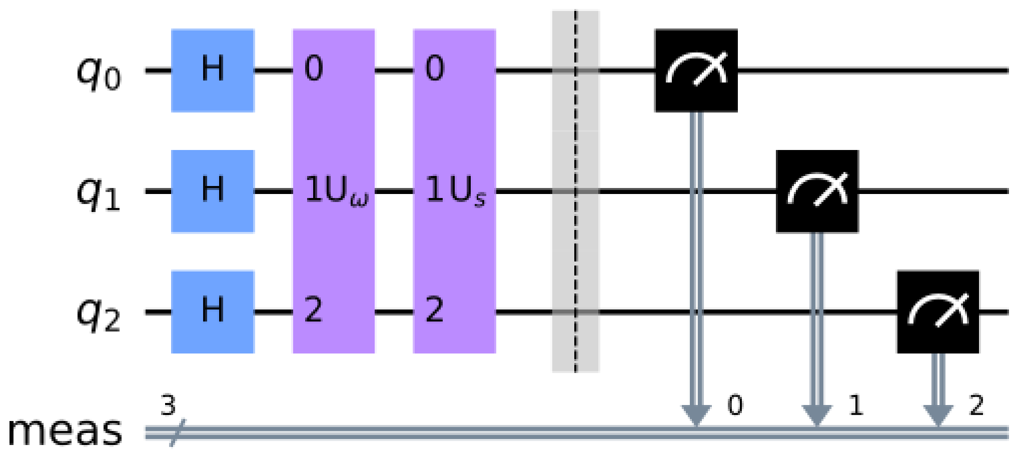
3.1 Qiskit Implementation

We now implement Grover's algorithm for the above example for 3-qubits and searching for two marked states $101\rangle$ and $110\rangle$. **Note:** Remember that Qiskit orders it's qubits the opposite way round to this resource, so the circuit drawn will appear flipped about the horizontal.

We create a phase oracle that will mark states $101\rangle$ and $110\rangle$ as the results (step 1).

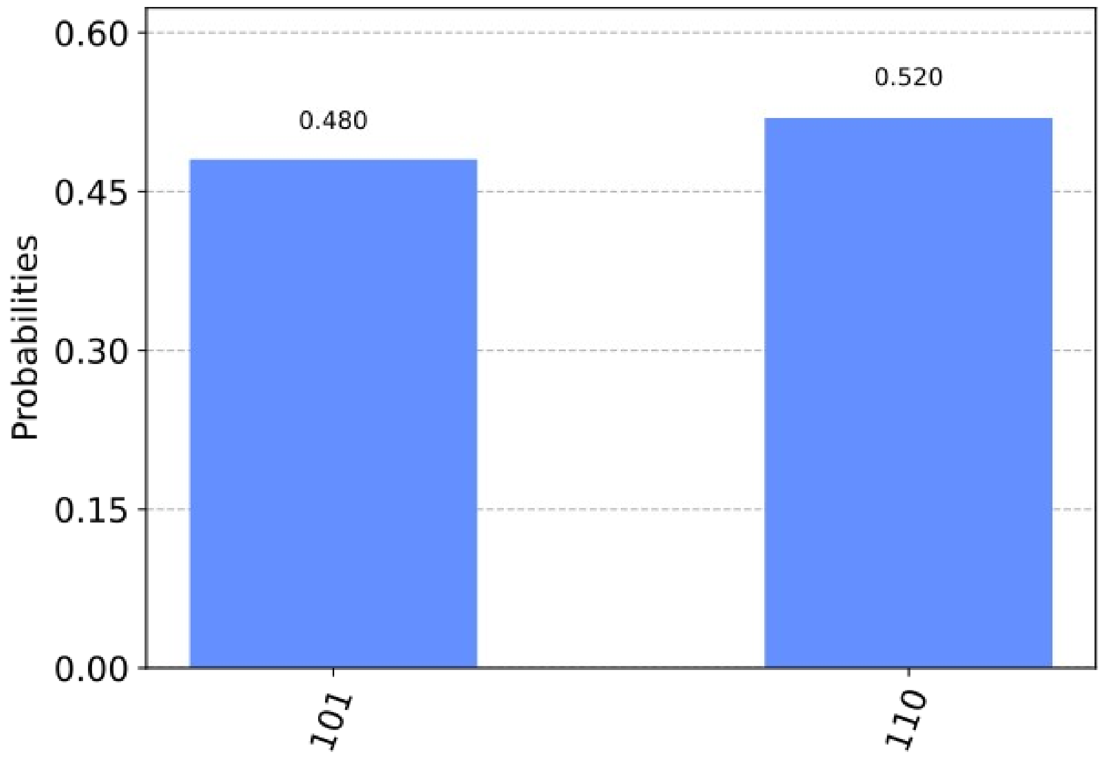
```
qc = QuantumCircuit(3)
qc.cz(0, 2)
qc.cz(1, 2)
oracle_ex3 = qc.to_gate()
oracle_ex3.name = "U$_\omega$"
```

In the last section, we used a diffuser specific to 2 qubits, in the cell below we will create a general diffuser for any number of qubits.

```
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
```
We'll now put the pieces together, with the creation of a uniform superposition at the start of the circuit and a measurement at the end. Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration.
```
n = 3
grover_circuit = QuantumCircuit(n)
grover_circuit = initialize_s(grover_circuit, [0,1,2])
grover_circuit.append(oracle_ex3, [0,1,2])
grover_circuit.append(diffuser(n), [0,1,2])
grover_circuit.measure_all()
grover_circuit.draw()
```


3.1.1 Experiment with Simulators

We can run the above circuit on the simulator.
```
aer_sim = Aer.get_backend('aer_simulator')
transpiled_grover_circuit = transpile(grover_circuit, aer_sim)
qobj = assemble(transpiled_grover_circuit)
results = aer_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)
```

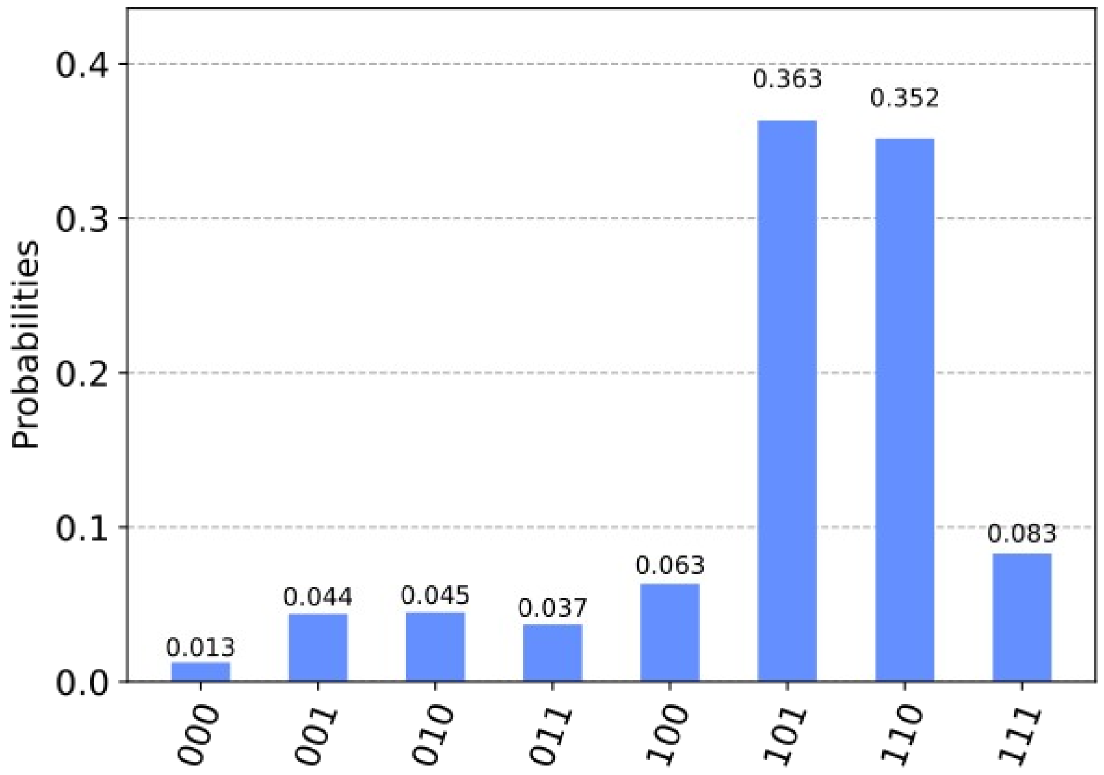


As we can see, the algorithm discovers our marked states 101⟩ and 110⟩.

3.1.2 Experiment with Real Devices

We can run the circuit on the real device as below.

```
backend = least_busy(provider.backends(filters=lambda x:
x.configuration().n_qubits >= 3 and
                                   not x.configuration().simulator
and x.status().operational==True))
print("least busy backend: ", backend)
least busy backend:  ibmq_mumbai
# Run our circuit on the least busy backend. Monitor the execution
of the job in the queue
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_circuit, device,
optimization_level=3)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)
Job Status: job has successfully run
# Get the results from the computation
results = job.result()
answer = results.get_counts(grover_circuit)
plot_histogram(answer)
```

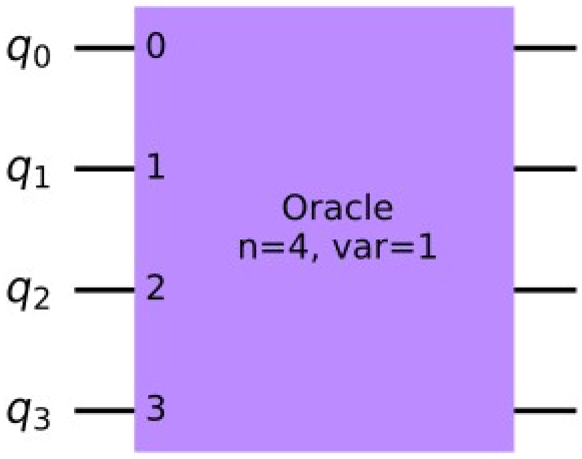


As we can (hopefully) see, there is a higher chance of measuring 101⟩ and 110⟩. The other results are due to errors in the quantum computation.

4. Problems

The function `grover_problem_oracle` below takes a number of qubits (`n`), and a `variant` and returns an n-qubit oracle. The function will always return the same oracle for the same `n` and `variant`. You can see the solutions to each oracle by setting `print_solutions = True` when calling `grover_problem_oracle`.

```
from qiskit_textbook.problems import grover_problem_oracle
## Example Usage
n = 4
oracle = grover_problem_oracle(n, variant=1)  # 0th variant of
oracle, with n qubits
qc = QuantumCircuit(n)
qc.append(oracle, [0,1,2,3])
qc.draw()
```



1. `grover_problem_oracle(4, variant=2)` uses 4 qubits and has 1 solution.
   a. How many iterations do we need to have a > 90% chance of measuring this solution?
   b. Use Grover's algorithm to find this solution state.
   c. What happens if we apply more iterations the number we calculated in problem 1a above? Why?
1. With 2 solutions and 4 qubits, how many iterations do we need for a >90% chance of measuring a solution? Test your answer using the oracle `grover_problem_oracle(4, variant=1)` (which has two solutions).
1. Create a function, `grover_solver(oracle, iterations)` that takes as input:
   - A Grover oracle as a gate (`oracle`)
   - An integer number of iterations (`iterations`)

   and returns a `QuantumCircuit` that performs Grover's algorithm on the '`oracle`' gate, with '`iterations`' iterations.

5. Solving Sudoku using Grover's Algorithm

The oracles used throughout this chapter so far have been created with prior knowledge of their solutions. We will now solve a simple problem using Grover's algorithm, for which we do not necessarily know the solution beforehand. Our problem is a 2×2 binary sudoku, which in our case has two simple rules:

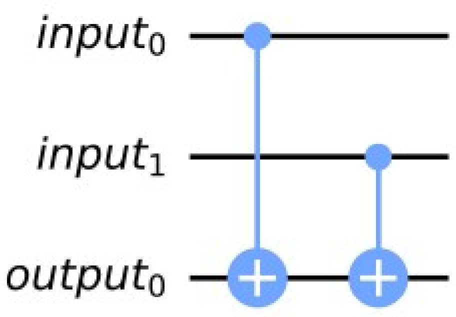
- No column may contain the same value twice
- No row may contain the same value twice

If we assign each square in our sudoku to a variable like so:
2×2 binary sudoku, with each square allocated to a different variable
we want our circuit to output a solution to this sudoku.
Note that, while this approach of using Grover's algorithm to solve this problem is not practical (you can probably find the solution in your head!), the purpose of this example is to demonstrate the conversion of classical decision problems into oracles for Grover's algorithm.

5.1 Turning the Problem into a Circuit

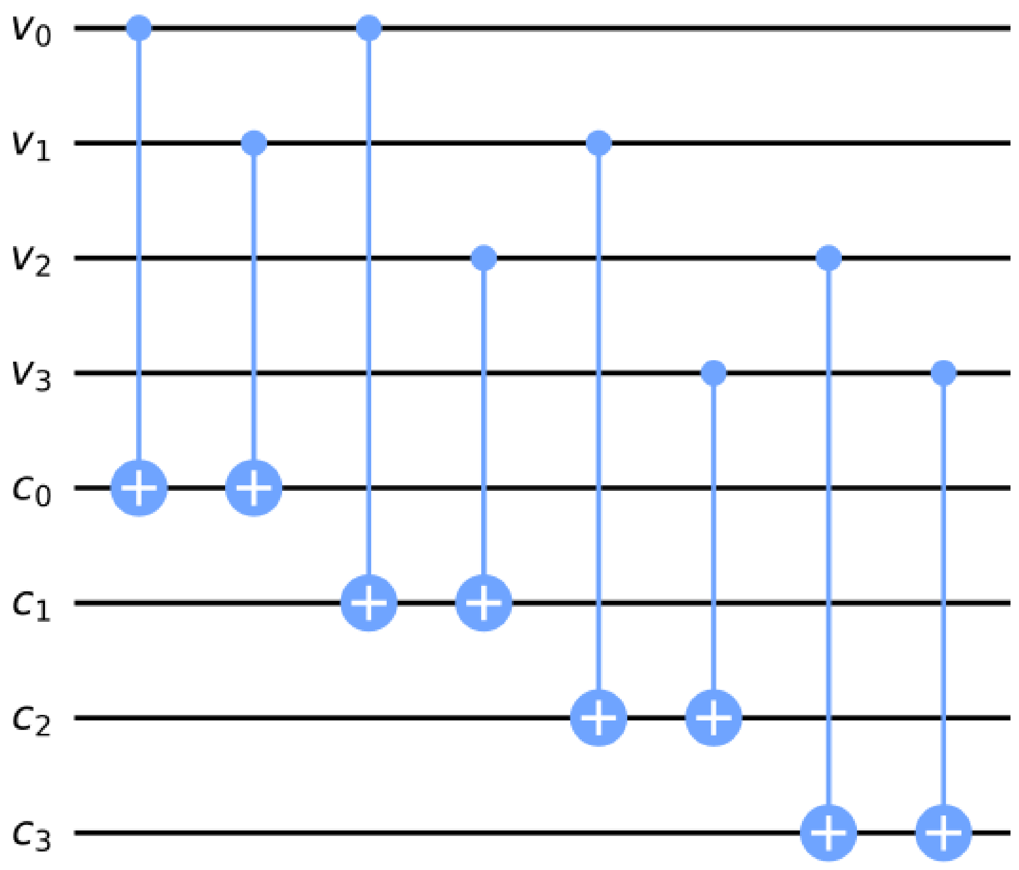
We want to create an oracle that will help us solve this problem, and we will start by creating a circuit that identifies a correct solution. Similar to how we created a classical adder using quantum circuits in *The Atoms of Computation*, we simply need to create a *classical* function on a quantum circuit that checks whether the state of our variable bits is a valid solution.
Since we need to check down both columns and across both rows, there are 4 conditions we need to check:

```
v0 ≠ v1   # check along top row
v2 ≠ v3   # check along bottom row
```

```
v0 ≠ v2   # check down left column
v1 ≠ v3   # check down right column
```
Remember we are comparing classical (computational basis) states. For convenience, we can compile this set of comparisons into a list of clauses:
```
clause_list = [[0,1],
               [0,2],
               [1,3],
               [2,3]]
```
We will assign the value of each variable to a bit in our circuit. To check these clauses computationally, we will use the `XOR` gate (we came across this in the atoms of computation).
```
def XOR(qc, a, b, output):
    qc.cx(a, output)
    qc.cx(b, output)
```
Convince yourself that the `output0` bit in the circuit below will only be flipped if `input0 ≠ input1`:
```
# We will use separate registers to name the bits
in_qubits = QuantumRegister(2, name='input')
out_qubit = QuantumRegister(1, name='output')
qc = QuantumCircuit(in_qubits, out_qubit)
XOR(qc, in_qubits[0], in_qubits[1], out_qubit)
qc.draw()
```


This circuit checks whether `input0 == input1` and stores the output to `output0`. To check each clause, we repeat this circuit for each pairing in `clause_list` and store the output to a new bit:
```
# Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v')  # variable bits
clause_qubits = QuantumRegister(4, name='c')  # bits to store
clause-checks

# Create quantum circuit
qc = QuantumCircuit(var_qubits, clause_qubits)

# Use XOR gate to check each clause
i = 0
```

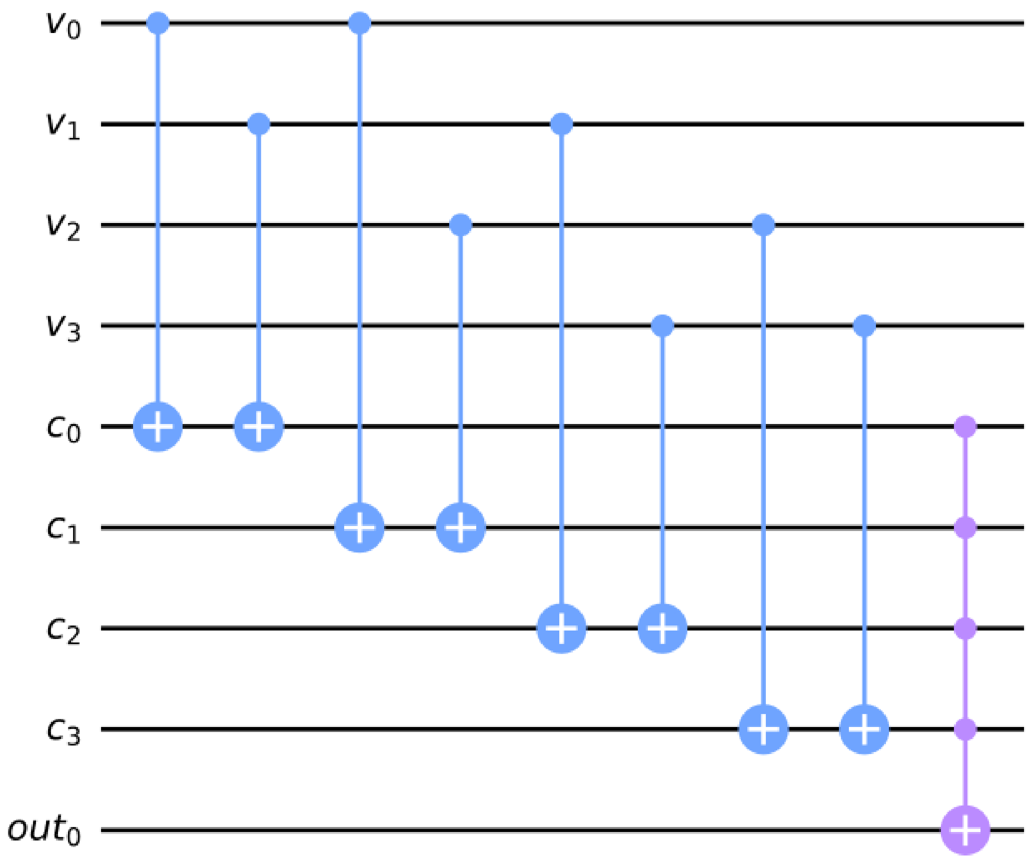
```
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

qc.draw()
```



The final state of the bits `c0, c1, c2, c3` will only all be `1` in the case that the assignments of `v0, v1, v2, v3` are a solution to the sudoku. To complete our checking circuit, we want a single bit to be `1` if (and only if) all the clauses are satisfied, this way we can look at just one bit to see if our assignment is a solution. We can do this using a multi-controlled-Toffoli-gate:

```
# Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit)

# Compute clauses
i = 0
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

# Flip 'output' bit if all clauses are satisfied
```

```
qc.mct(clause_qubits, output_qubit)

qc.draw()
```



The circuit above takes as input an initial assignment of the bits `v0`, `v1`, `v2` and `v3`, and all other bits should be initialized to `0`. After running the circuit, the state of the `out0` bit tells us if this assignment is a solution or not; `out0 = 0` means the assignment *is not* a solution, and `out0 = 1` means the assignment *is* a solution.

Important: Before you continue, it is important you fully understand this circuit and are convinced it works as stated in the paragraph above.

5.2 Uncomputing, and Completing the Oracle

We can now turn this checking circuit into a Grover oracle using phase kickback. To recap, we have 3 registers:

- One register which stores our sudoku variables (we'll say $x = v_3, v_2, v_1, v_0$)
- One register that stores our clauses (this starts in the state $0000\rangle$ which we'll abbreviate to $0\rangle$)
- And one qubit ($\text{out}_0\rangle$) that we've been using to store the output of our checking circuit.

To create an oracle, we need our circuit (U_ω) to perform the transformation:

$U_\omega |x\rangle |0\rangle$

If we set the `out0` qubit to the superposition state $-\rangle$ we have:

$U_\omega |x\rangle |0\rangle \vee -\rangle \quad U_\omega |x\rangle |0\rangle \otimes 1/\sqrt{2}(|0\rangle - |1\rangle)$

¿ ¿

If $f(x) = 0$, then we have the state:

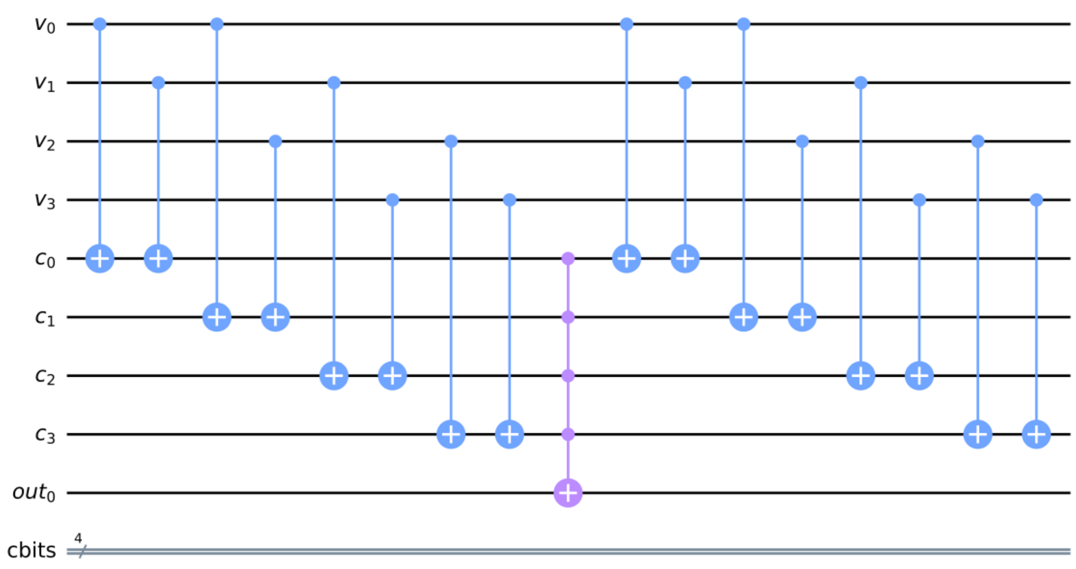
$|x\rangle |0\rangle \otimes 1/\sqrt{2}(|0\rangle - |1\rangle)$

$|x\rangle |0\rangle \vee -\rangle$

(i.e. no change). But if $f(x) = 1$ (i.e. $x = \omega$), we introduce a negative phase to the $-\rangle$ qubit:

$|x\rangle|0\rangle \otimes 1/\sqrt{2}(|1\rangle - |0\rangle)$
$|x\rangle|0\rangle \otimes -1/\sqrt{2}(|0\rangle - |1\rangle)$
$-|x\rangle|0\rangle \vee -\rangle$
This is a functioning oracle that uses two auxiliary registers in the state $|0\rangle|-\rangle$:
$$U_\omega|x\rangle|0\rangle \vee -\rangle = \begin{cases} |x\rangle|0\rangle \vee -\rangle \text{ for } x \neq \omega \\ -|x\rangle|0\rangle \vee -\rangle \text{ for } x = \omega \end{cases}$$
To adapt our checking circuit into a Grover oracle, we need to guarantee the bits in the second register (`c`) are always returned to the state $0000\rangle$ after the computation. To do this, we simply repeat the part of the circuit that computes the clauses which guarantees `c0 = c1 = c2 = c3 = 0` after our circuit has run. We call this step *'uncomputation'*.

```python
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
cbits = ClassicalRegister(4, name='cbits')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

def sudoku_oracle(qc, clause_list, clause_qubits):
    # Compute clauses
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

    # Flip 'output' bit if all clauses are satisfied
    qc.mct(clause_qubits, output_qubit)

    # Uncompute clauses to reset clause-checking bits to 0
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

sudoku_oracle(qc, clause_list, clause_qubits)
qc.draw()
```



In summary, the circuit above performs:

$$U_\omega |x\rangle|0\rangle \vee \text{out}_0\rangle = \{ \begin{array}{l} |x\rangle|0\rangle \vee \text{out}_0\rangle \text{ for } x \neq \omega \\ |x\rangle|0\rangle \otimes X \vee \text{out}_0\rangle \text{ for } x = \omega \end{array}$$

and if the initial state of ,:

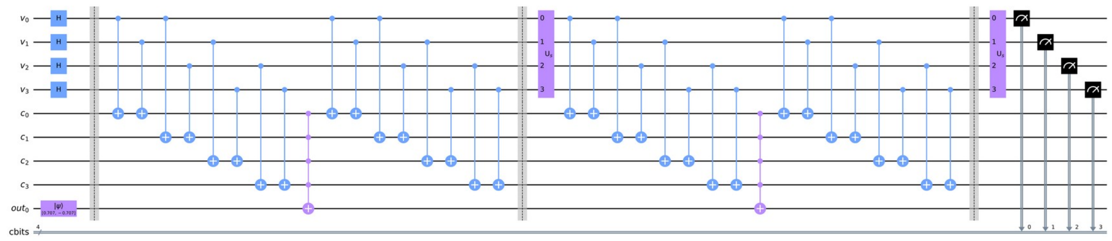
$$U_\omega |x\rangle|0\rangle \vee -\rangle = \{ \begin{array}{l} |x\rangle|0\rangle \vee -\rangle \text{ for } x \neq \omega \\ -|x\rangle|0\rangle \vee -\rangle \text{ for } x = \omega \end{array}$$

5.3 The Full Algorithm

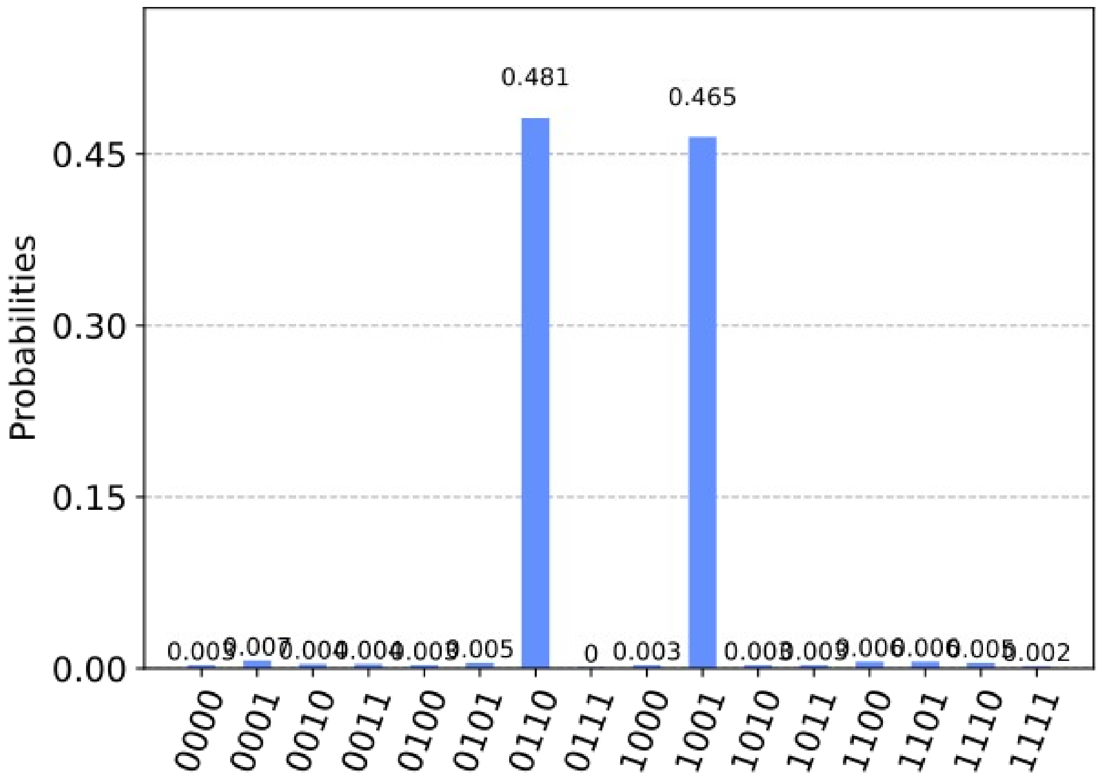
All that's left to do now is to put this oracle into Grover's algorithm!

```
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
cbits = ClassicalRegister(4, name='cbits')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

# Initialize 'out0' in state |->
qc.initialize([1, -1]/np.sqrt(2), output_qubit)

# Initialize qubits in state |s>
qc.h(var_qubits)
qc.barrier()  # for visual separation

## First Iteration
# Apply our oracle
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier()  # for visual separation
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

## Second Iteration
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier()  # for visual separation
```

```
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

# Measure the variable qubits
qc.measure(var_qubits, cbits)

qc.draw(fold=-1)
```



```
# Simulate and plot results
aer_simulator = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, aer_simulator)
qobj = assemble(transpiled_qc)
result = aer_sim.run(qobj).result()
plot_histogram(result.get_counts())
```



There are two bit strings with a much higher probability of measurement than any of the others, `0110` and `1001`. These correspond to the assignments:

```
v0 = 0
v1 = 1
v2 = 1
v3 = 0
```

and

```
v0 = 1
v1 = 0
v2 = 0
v3 = 1
```
which are the two solutions to our sudoku! The aim of this section is to show how we can create Grover oracles from real problems. While this specific problem is trivial, the process can be applied (allowing large enough circuits) to any decision problem. To recap, the steps are:

1. Create a reversible classical circuit that identifies a correct solution
1. Use phase kickback and uncomputation to turn this circuit into an oracle
1. Use Grover's algorithm to solve this oracle

General Imports

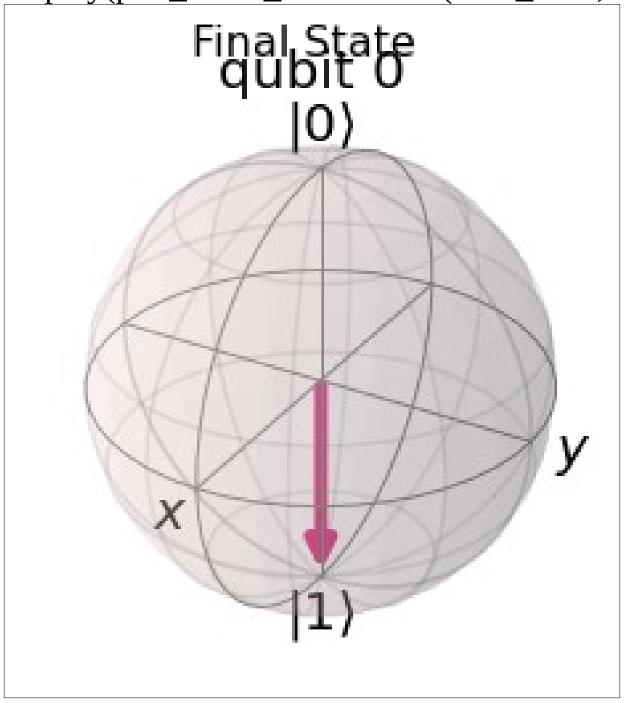
[1]:
```
from qiskit import QuantumCircuit,QuantumRegister,ClassicalRegister, Aer,
assemble ,execute ,transpile, IBMQ

from math import pi, sqrt
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.quantum_info import Statevector
from qiskit_textbook.tools import array_to_latex

style = {'backgroundcolor': 'lightgreen'}

# oi 3 simulators
svsim = Aer.get_backend('statevector_simulator')
usim = Aer.get_backend('unitary_simulator')
qasm_sim = Aer.get_backend('qasm_simulator')

shots=1000 #gia to qasm_simulator , poses fores na trexei
```
Example 1

As shown in Problem Set 1, we know that for 1=1 the eigenvector is $||u1\rangle$=12√(11) and for 2=−1 the eigenvector is $||u2\rangle$=12√(1−1)

We now want to *prove* that $X||uj\rangle=\ j||uj\rangle,j=1,2$

[2]:
```
u_1 = [1/sqrt(2), 1/sqrt(2)]  # Define state |u_1>
u_2 = [1/sqrt(2), -1/sqrt(2)]  # Define state |u_2>

qc = QuantumCircuit(1)
qc.x(0)
# qc.measure_all()

# trexo pali to statevector_simulator gia na paro tin teliki katastasi
test = transpile(qc,svsim)
```

```
qobj = assemble(test)
final_state = svsim.run(qobj).result().get_statevector()
display(plot_bloch_multivector(final_state, title="Final State"))
```

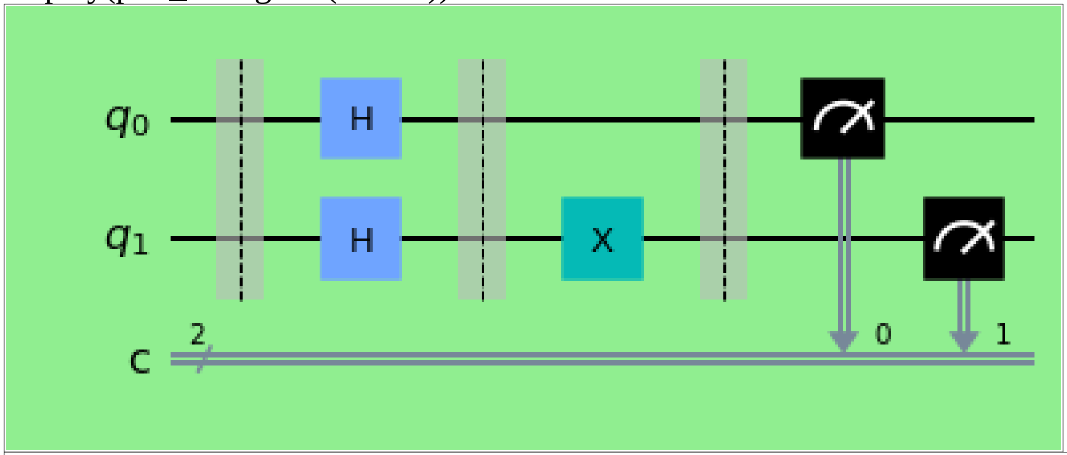


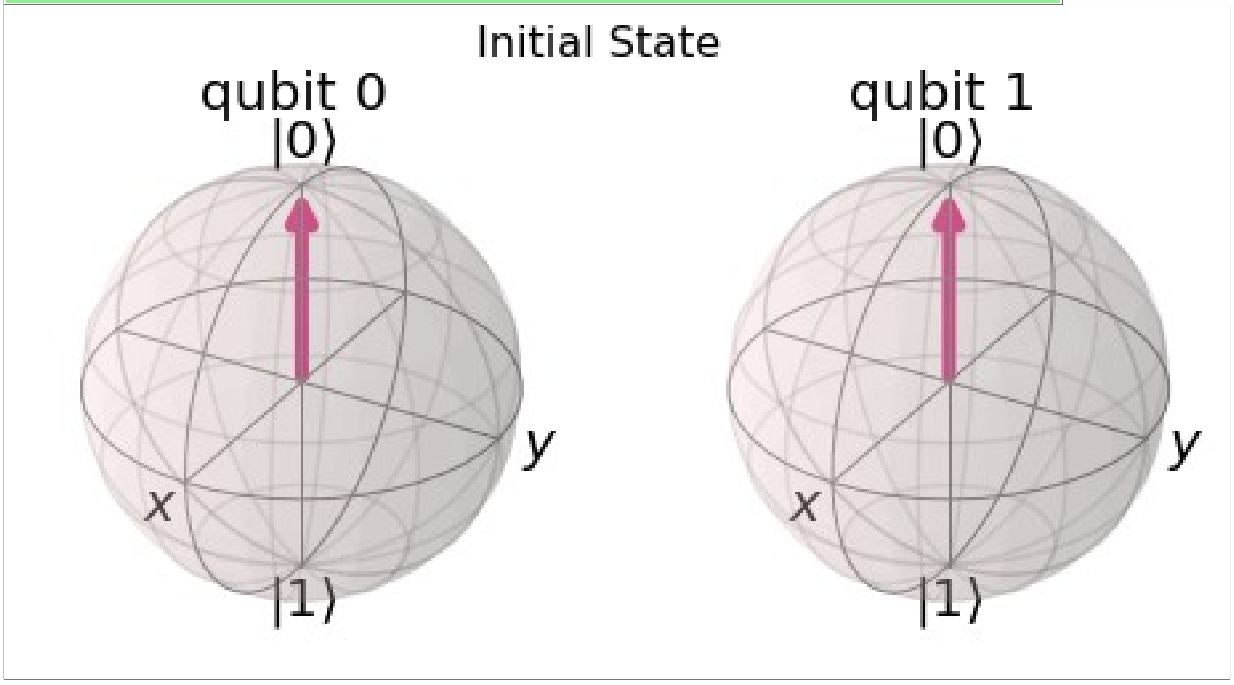
Example 2

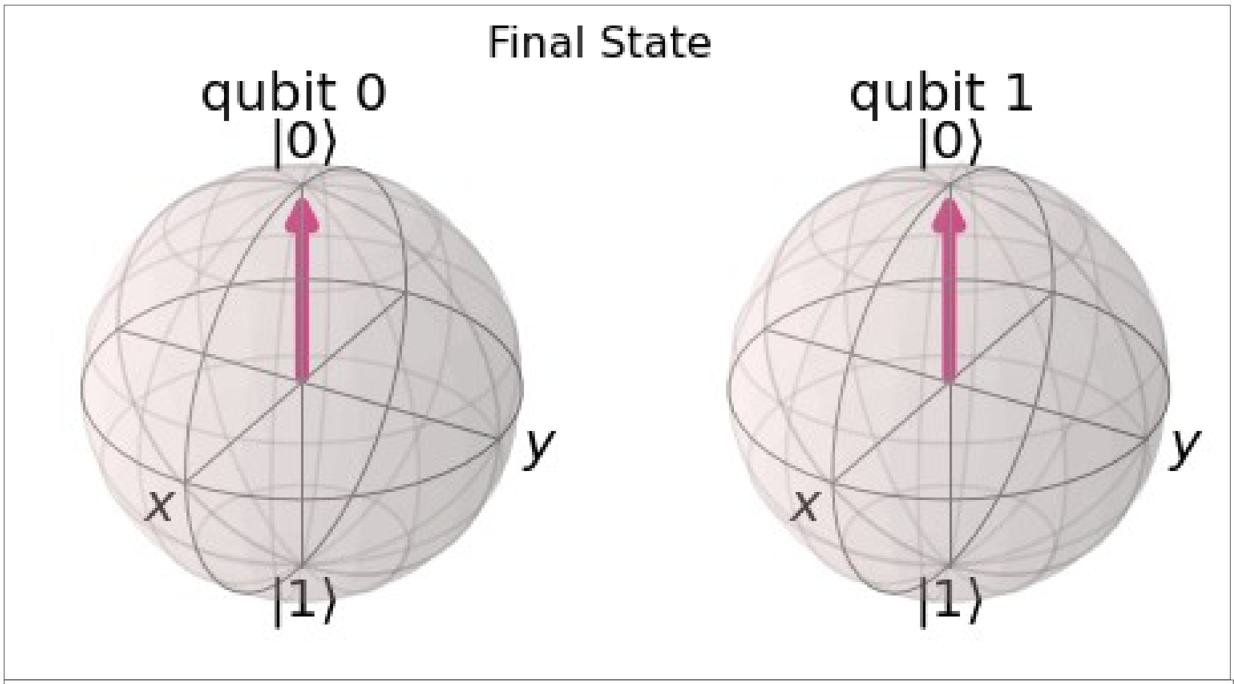
Circuit 1

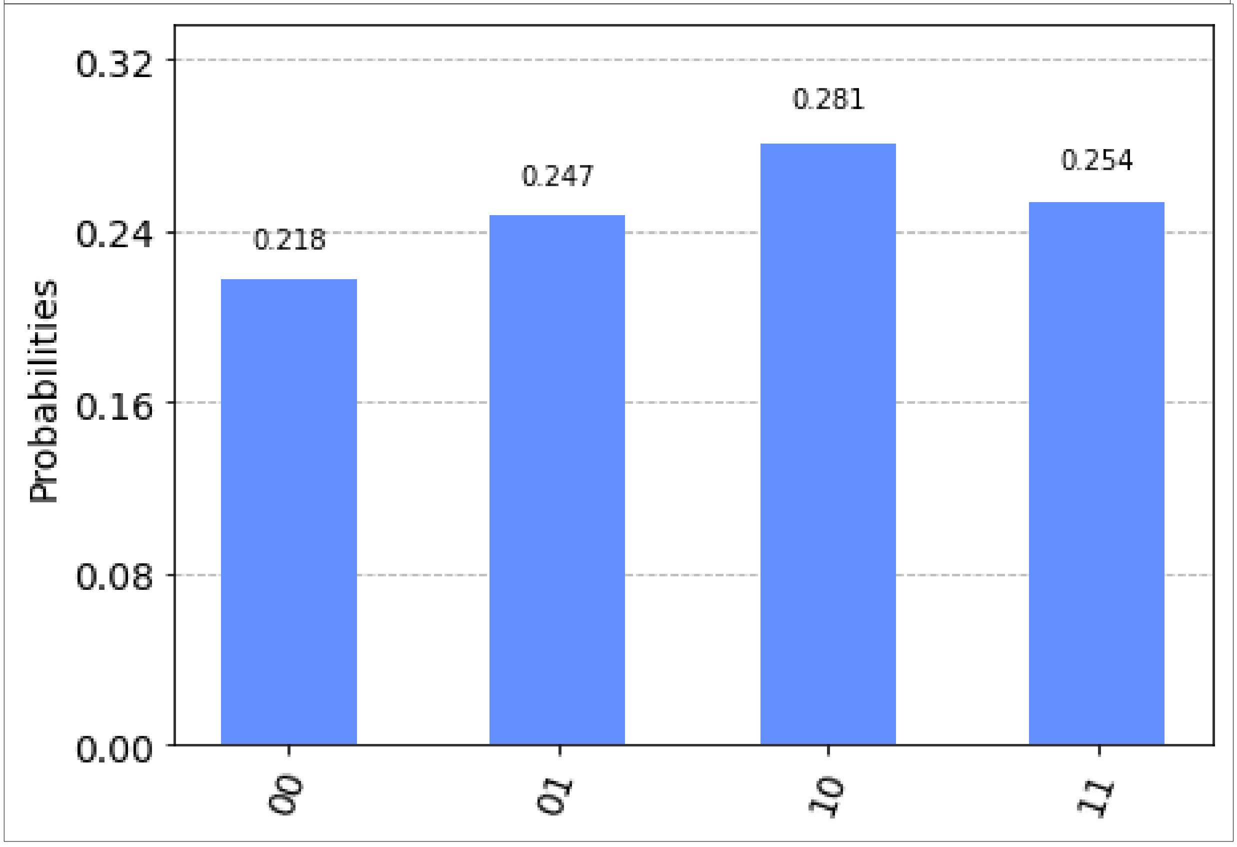
```
[3]:
# quantum circuit with 2 qubit and 2 classical bits
qc = QuantumCircuit(2,2)

#We run statevector_simulator to find the initial state 00
test = transpile(qc,svsim)
qobj = assemble(test)
initial_state = svsim.run(qobj).result().get_statevector()

#the circuit
qc.barrier()
qc.h(0)
qc.h(1)
qc.barrier()
qc.x(1)
qc.barrier()

qc.measure(0,0)   #save measurment to the first classical bit
qc.measure(1,1)   #save measurment to the second classical bit

display(qc.draw(output='mpl',style=style))

# trexo pali to statevector_simulator gia na paro tin teliki katastasi
test = transpile(qc,svsim)
qobj = assemble(test)
final_state = svsim.run(qobj).result().get_statevector()
# display(test.draw(output='mpl',style=style)) #we can see that the transpiler did not change our circuit
```

```
display(plot_bloch_multivector(initial_state, title="Initial State"))
display(plot_bloch_multivector(final_state, title="Final State"))

#trexoume ton qasm_simulator
job = execute(qc, qasm_sim, shots=shots)
counts = job.result().get_counts()
display(plot_histogram(counts))
```









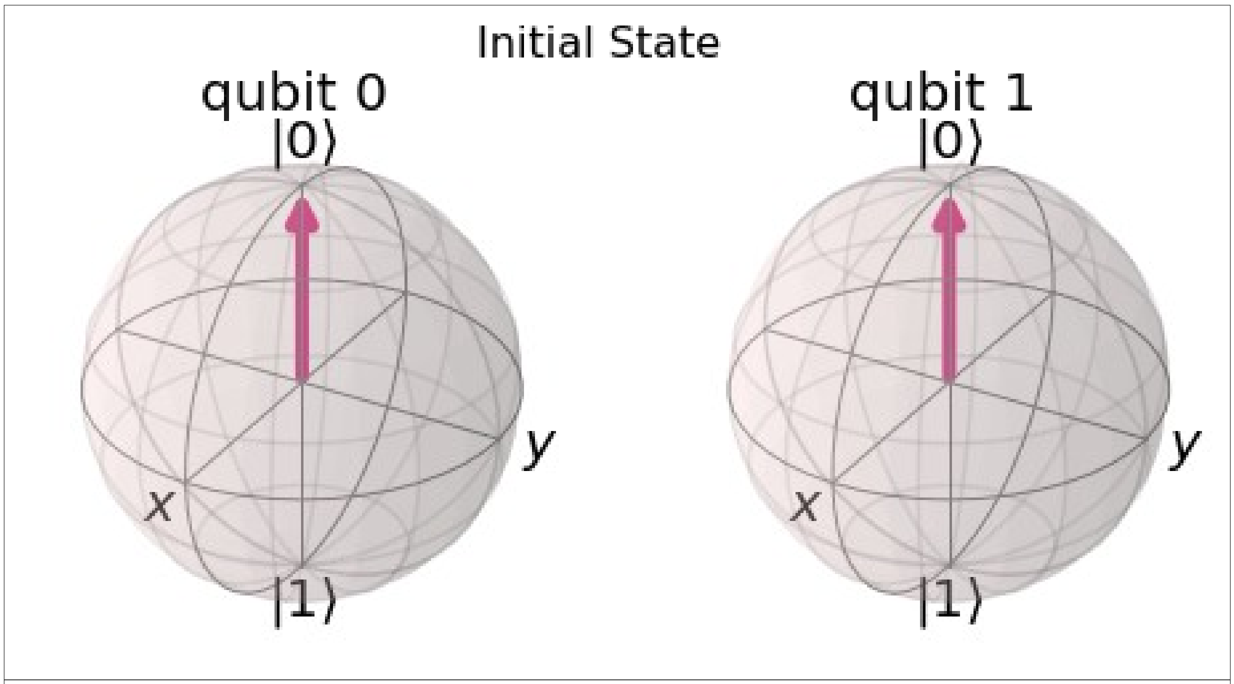
Circuit 2

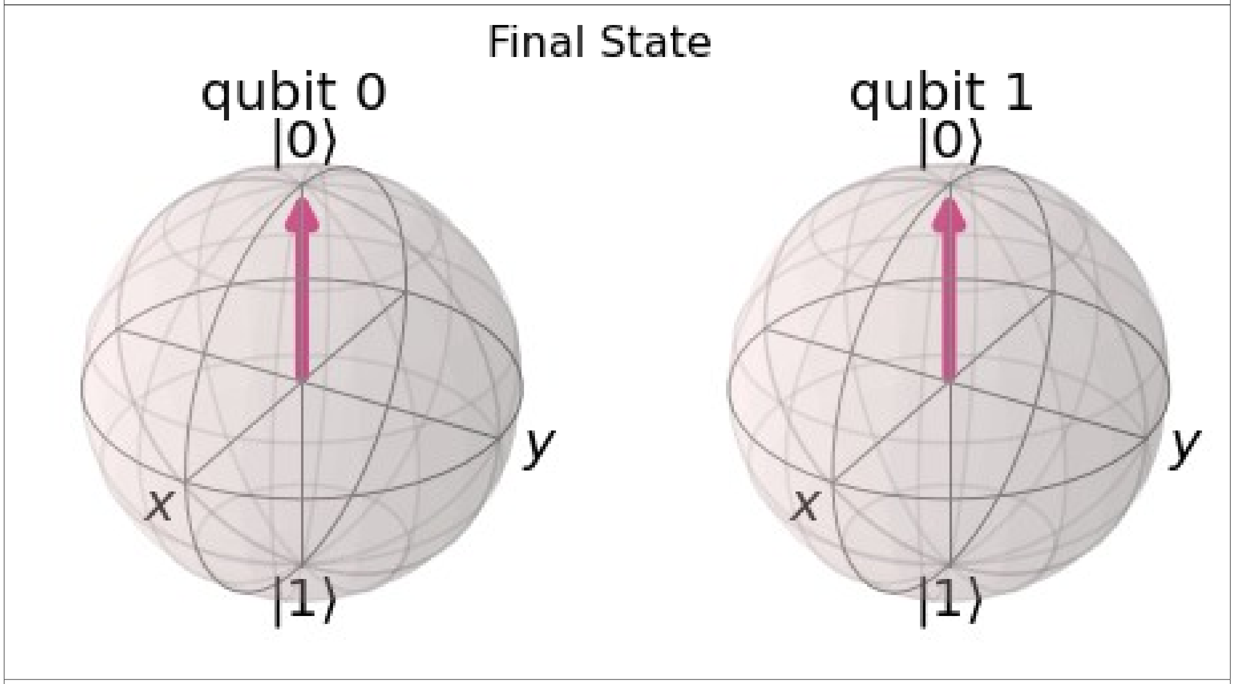
```
# quantum circuit with 2 qubit and 2 classical bits
qc = QuantumCircuit(2,2)

#We run statevector_simulator to find the initial state 00
test = transpile(qc,svsim)
qobj = assemble(test)
initial_state = svsim.run(qobj).result().get_statevector()

#the circuit
qc.barrier()
qc.x(1)
qc.barrier()
qc.h(0)
qc.h(1)
qc.barrier()
qc.x(1)
qc.barrier()

qc.measure(0,0)   #save measurment to the first classical bit
qc.measure(1,1)   #save measurment to the second classical bit

display(qc.draw(output='mpl',style=style))

# trexo pali to statevector_simulator gia na paro tin teliki katastasi
test = transpile(qc,svsim)
qobj = assemble(test)
final_state = svsim.run(qobj).result().get_statevector()
# display(test.draw(output='mpl',style=style)) #we can see that the transpiler did not change our
circuit

display(plot_bloch_multivector(initial_state, title="Initial State"))
display(plot_bloch_multivector(final_state, title="Final State"))

#trexoume ton qasm_simulator
job = execute(qc, qasm_sim, shots=shots)
counts = job.result().get_counts()
display(plot_histogram(counts))
```

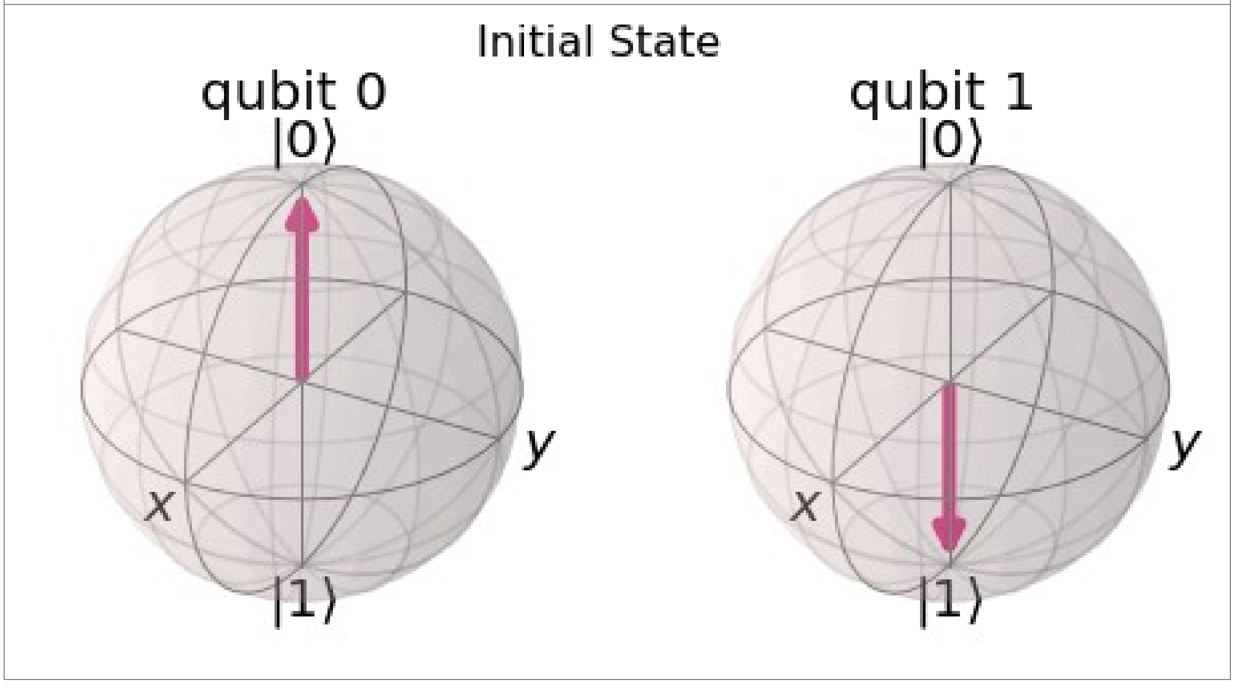
Example 3

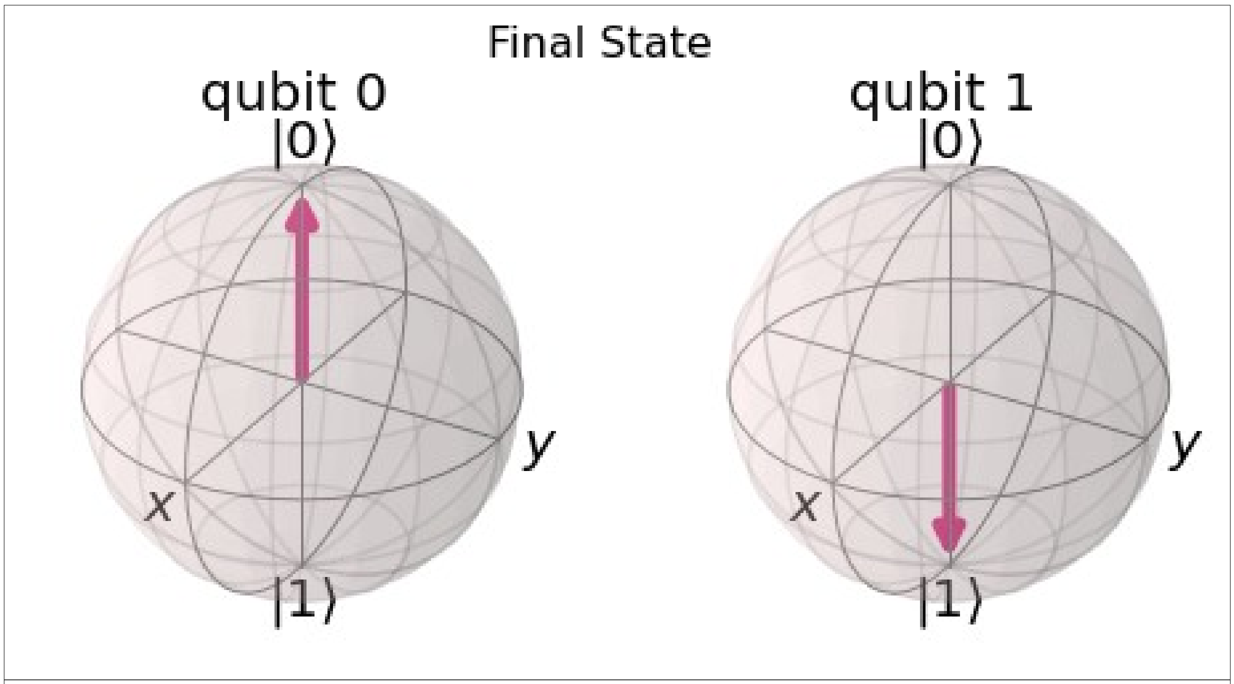
[5]:

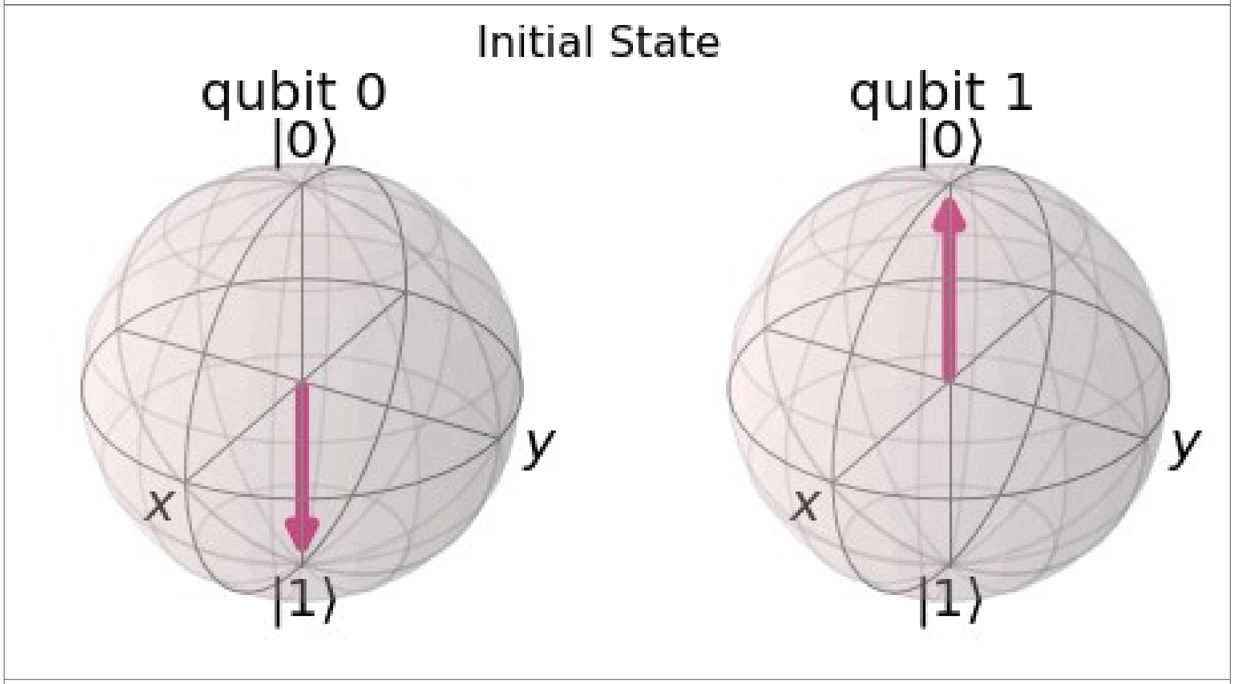
```
def getState(i):
    if(i==0):
        return [1, 0]
    return [0, 1]

def getStatePair(i, j):
    if(i==0 and j==0):
        return [1, 0, 0, 0]
    if(i==0 and j==1):
        return [0, 0, 1, 0]
    if(i==1 and j==0):
        return [0, 1, 0, 0]
    return [0, 0, 0, 1]

def getCircWithInitial(i, j):
#let us create a function that returns the circuit with different
#initializations and bruteforce all posibilities of initial states

    # quantum circuit with 2 qubits
    qc = QuantumCircuit(2)
    #force the initial state
    qc.initialize(getState(i), 0)
    qc.initialize(getState(j), 1)
    #the circuit
    qc.barrier()
    qc.h(0)
    qc.x(1)
    qc.barrier()
    qc.z(0)
    qc.y(1)
    qc.barrier()
    qc.h(0)
    qc.z(1)
    qc.barrier()
    qc.s(0)
    qc.barrier()
    qc.z(0)
    qc.barrier()
    qc.t(0)
    qc.barrier()
    qc.t(0)
    qc.barrier()
    qc.x(0)
    qc.barrier()
    qc.measure_all()
    return qc


for x in [0, 1]:
    for y in [0, 1]:
        display(plot_bloch_multivector(getStatePair(x, y), title="Initial State"))
        qc = getCircWithInitial(x, y)
        test = transpile(qc,svsim)
        qobj = assemble(test)
        final_state = svsim.run(qobj).result().get_statevector()
        display(plot_bloch_multivector(final_state, title="Final State"))



display(qc.draw(output='mpl',style=style))
```

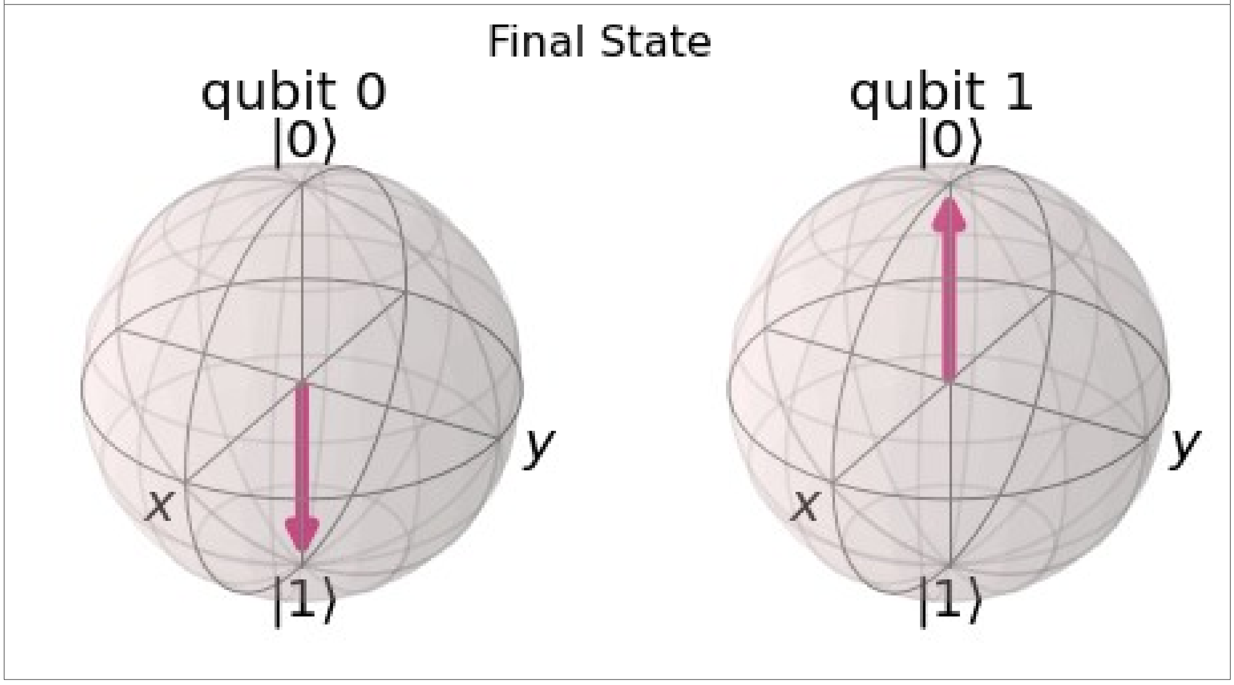

Initial State
qubit 0
|0⟩
y
x
|1⟩
qubit 1
|0⟩
y
x
|1⟩

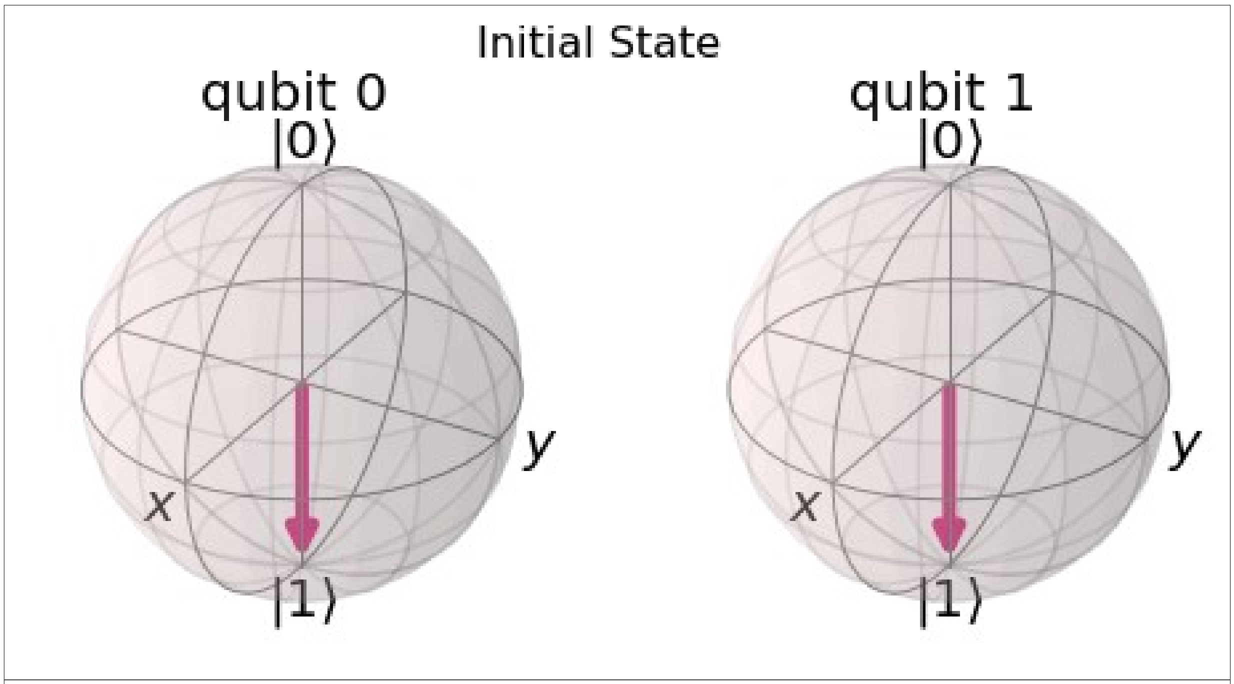

Final State
qubit 0
|0⟩
y
x
|1⟩
qubit 1
|0⟩
y
x
|1⟩

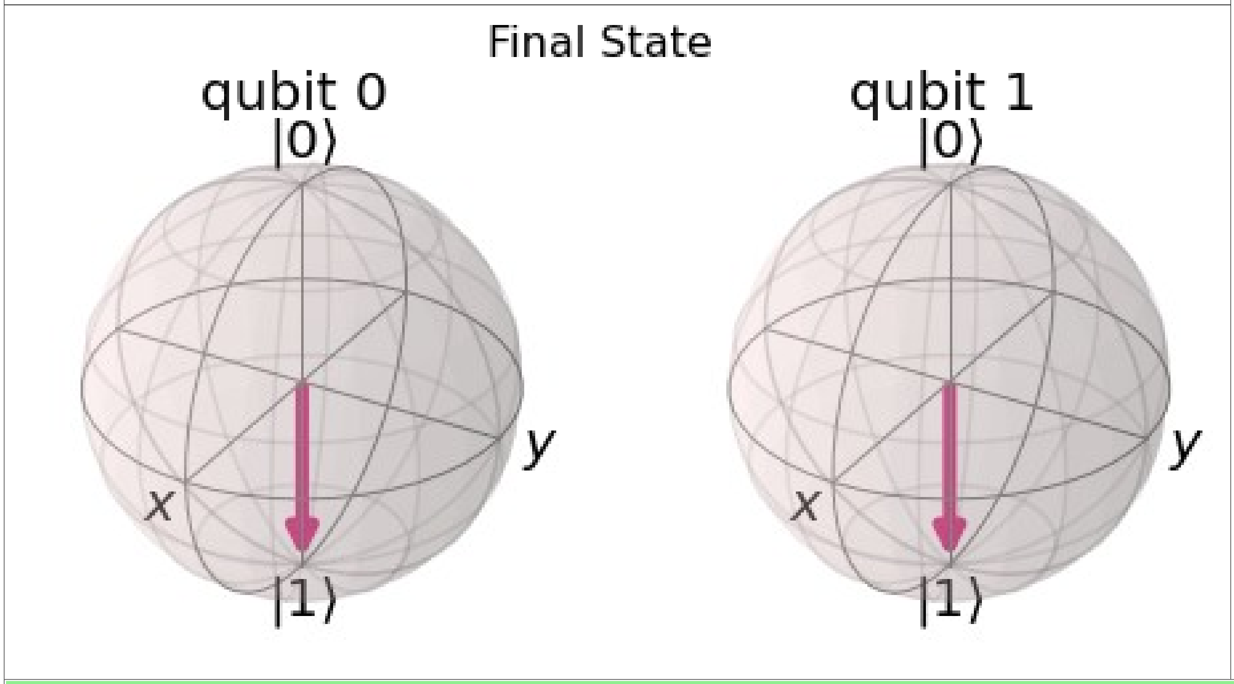

Initial State
qubit 0
|0⟩
y
x
|1⟩
qubit 1
|0⟩
y
x
|1⟩

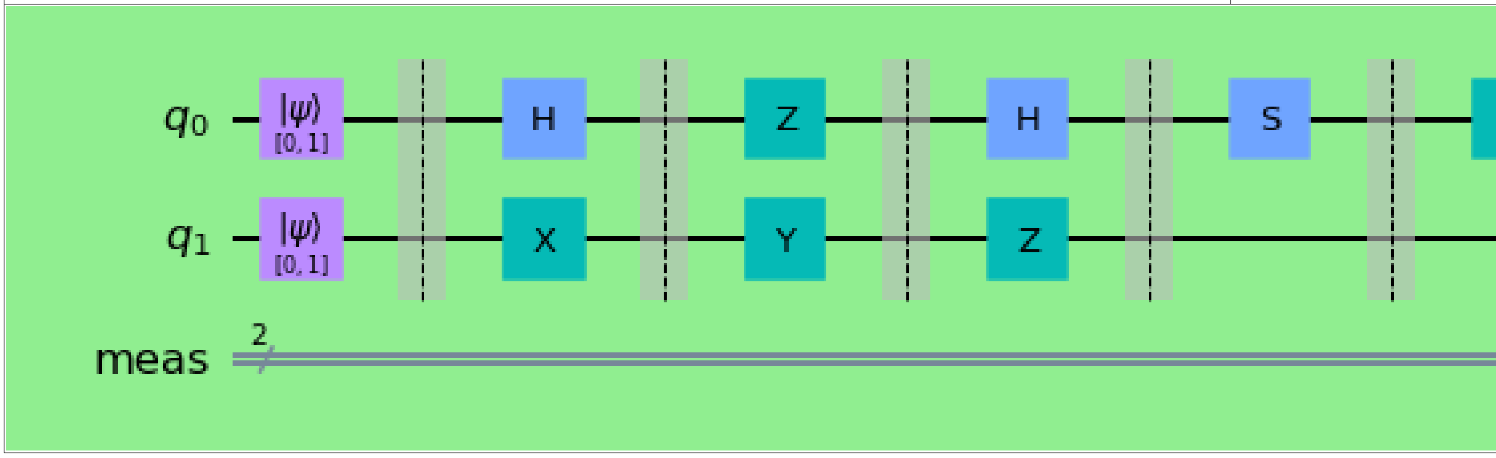

Final State
qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y


Initial State
qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y


Final State
qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y



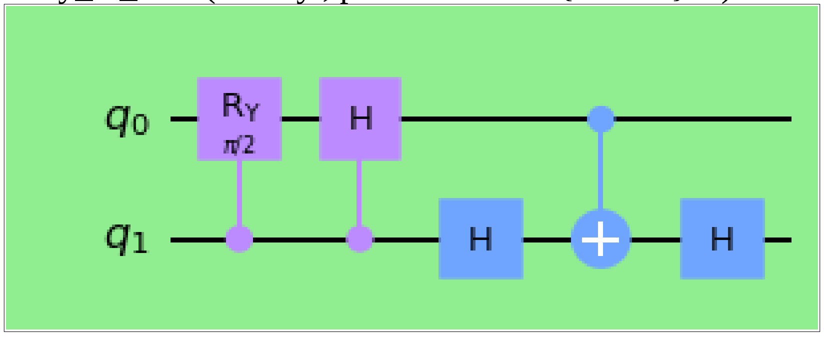




From the output above,
we can conclude that for each initial state, we get the same final state, and therefore the unitary of the circuit is the *Identity* matrix

[6]:
# quantum circuit with 2 qubit and 2 classical bits

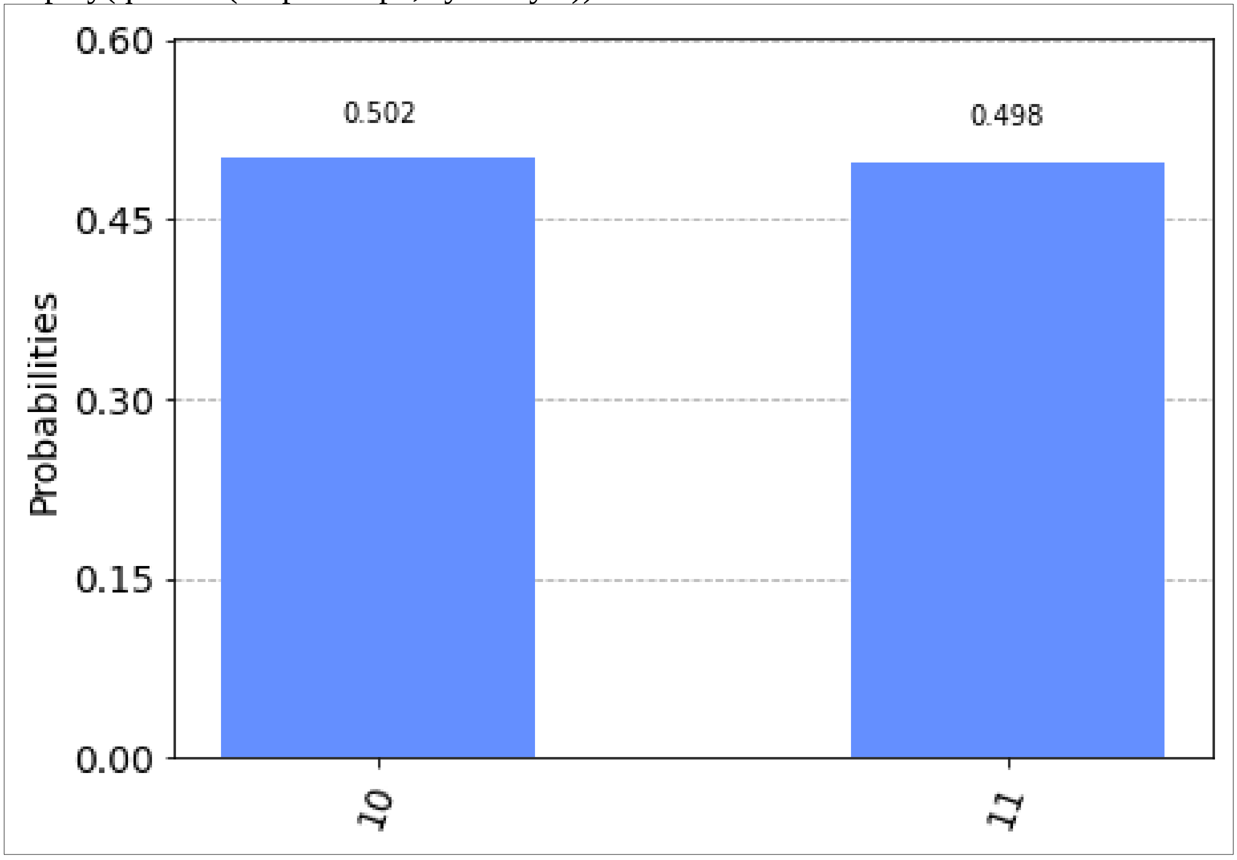
```
qc = QuantumCircuit(2)

#We run statevector_simulator to find the initial state 00
test = transpile(qc,svsim)
qobj = assemble(test)
initial_state = svsim.run(qobj).result().get_statevector()

#the circuit
qc.cry(pi/2,1,0)
qc.ch(1,0)
qc.h(1)
qc.cx(0,1)
qc.h(1)

display(qc.draw(output='mpl',style=style))

#trexo unitary_simulator gia na paro to matrix representation tou kuklomatos
test = transpile(qc,usim)
qobj = assemble(test)
unitary = usim.run(qobj).result().get_unitary()
array_to_latex(unitary , pretext = "\\text{Circuit}=")
```

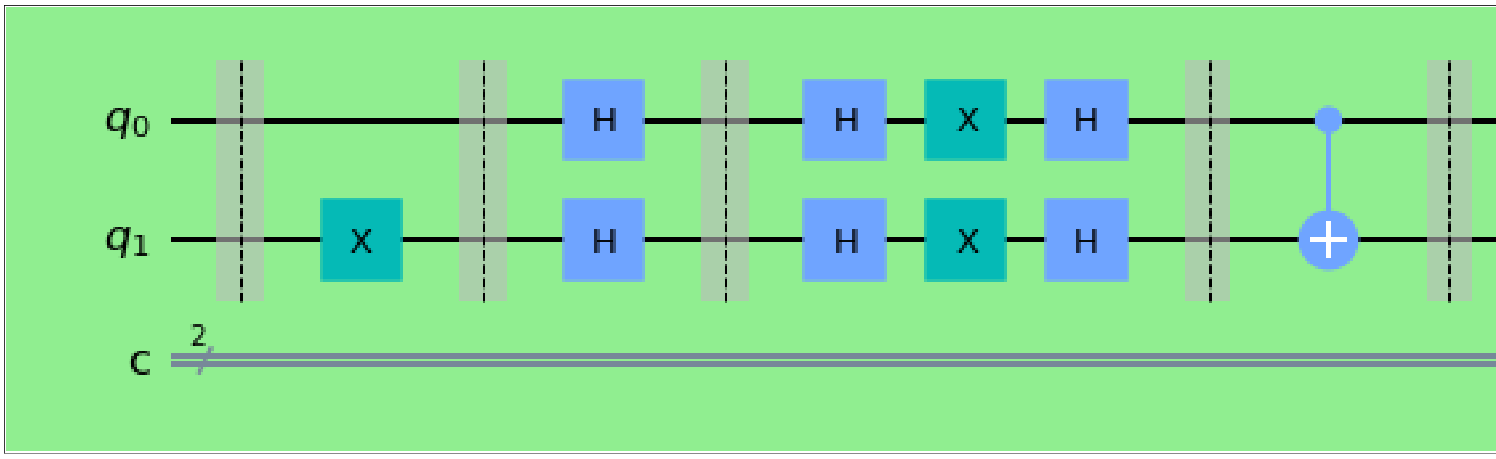


```
Circuit=●→✓✓✓✓1000010000100001❒❍▬
```

Example 4

```
[7]:
def oracleType(qc, function):
    if (function == "constant"):
        qc.h(0)
        qc.x(0)
        qc.h(0)

    if (function == "balanced"):
        qc.h(0)
        qc.h(1)
        qc.x(0)
        qc.x(1)
        qc.h(0)
        qc.h(1)

[8]:
qc = QuantumCircuit(2, 2)
qc.barrier()
qc.x(1)
qc.barrier()
qc.h(0)
qc.h(1)
qc.barrier()
oracleType(qc, "balanced")
qc.barrier()
qc.cx(0, 1)
qc.barrier()
qc.h(0)
qc.barrier()

# Measures the qubits on a circuit
qc.measure(0, 1)
qc.measure(1, 0)
# Running the circuit on a simulator which will give you theoretical results
#trexoume ton qasm_simulator
job = execute(qc, qasm_sim, shots=shots)
counts = job.result().get_counts()
display(plot_histogram(counts))

display(qc.draw(output='mpl',style=style))
```





Example 5

Grover's Algorithm with 3 qubits

[9]:

```
def initialize_s(qc, qubits):
    """Apply a H-gate to 'qubits' in qc"""
    for q in qubits:
        qc.h(q)
    return qc

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
```

[10]:

```
qc = QuantumCircuit(3)
qc.cz(0, 2)
```

qc.cz(1, 2)
oracle_ex3 = qc.to_gate()
oracle_ex3.name = "U$_\omega$"
grover_circuit = QuantumCircuit(3)
grover_circuit = initialize_s(grover_circuit, [0,1,2])
grover_circuit.append(oracle_ex3, [0,1,2])
grover_circuit.append(diffuser(3), [0,1,2])
grover_circuit.measure_all()
grover_circuit.draw()
[10]:

```
        ┌───┐┌─────────────┐┌──────────┐ ░ ┌─┐      
   q_0: ┤ H ├┤0            ├┤0         ├─░─┤M├──────
        ├───┤│             ││          │ ░ └╥┘┌─┐   
   q_1: ┤ H ├┤1 U$_\omega$ ├┤1 U$_s$   ├─░──╫─┤M├───
        ├───┤│             ││          │ ░  ║ └╥┘┌─┐
   q_2: ┤ H ├┤2            ├┤2         ├─░──╫──╫─┤M├
        └───┘└─────────────┘└──────────┘ ░  ║  ║ └╥┘
meas: 3/════════════════════════════════════╩══╩══╩═
                                            0  1  2 
```

# Oracle

A quantum computing demo using various oracles in Grover's
[algorithm](https://qiskit.org/textbook/ch-algorithms/grover.html).

Read the full article [Quantum Computing Hello
World](http://www.primaryobjects.com/2022/01/22/quantum-computing-hello-world/).

Here's an example for printing "Hello World"!

```text
Random letters:
['l', 'h', 'Q', 'o', 'l', 'Q', 'C', 'e']

Final result from the quantum circuit:

h (at index 1 [001])
e (at index 7 [111])
l (at index 0 [000])
l (at index 0 [000])
o (at index 3 [011])
```

## What is it?

Oracle is a tutorial example of writing a quantum computing program in Qiskit that searches through a random array of letters in order to find each letter in sequence for a target sentence, such as "Hello World!".

The idea is similar to the traditional [ransom](https://leetcode.com/problems/ransom-note/) [note](https://dev.to/teekay/algorithms-problem-solving-ransom-note-2f5f) [problem](https://medium.com/@harycane/ransom-note-af09b54904d0). You're given a magazine of randomly cut out letters. The letters are strewn across a table in a random fashion. Your task is to find enough letters from the table in order to paste them together to produce the phrase "hello world".

Given two stings ransomNote and magazine, return true if ransomNote can be constructed from magazine and false otherwise. Each letter in magazine can only be used once in ransomNote.

#### Other Examples

- [Hello World](hello.py)
- [Even Numbers](even.py)
- [Odd Numbers](odd.py)
- [Magic Eight Ball](magicball.py)

## Why would you use a quantum computer for this?

To demonstrate the power of a quantum computer compared to a classical one, of course!

On a classical computer, the time complexity for searching through an unordered list of elements would take O(n). That is, in the worst case we would need to iterate through the entire array in order to locate a target element - if the last element in the array is the target.

By contrast, a quantum computer using Grover's [algorithm](https://en.wikipedia.org/wiki/Grover%27s_algorithm) can locate the target element in O(sqrt(n)). This provides a [quadratic](https://www.quora.com/When-people-say-a-quantum-computer-gains-a-quadratic-speedup-for-search-algorithms-does-that-mean-the-complexity-is-square-rooted-i-e-sqrt-n) speed-up in time complexity to find the target element. A quantum computer does this by representing each bit in the length of the array with a qubit. For example, for a random array of length 8, we can represent this on a quantum computer using 3 qubits, which gives us 2^3=8 possible combinations. Since a quantum computer can calculate all possible values for qubits within a quantum circuit in a single cycle, the program can evaluate all potential indices in the array in order to locate the target element.

While the example in this tutorial of selecting letters from an array of random elements is simplistic, it nevertheless demonstrates the speed-up in time complexity for searching and locating the desired elements.

## Hello World

Below is an example of a random array being searched to produce the word "hello".

Consider the following array of random letters that we want to search through to produce the word "hello".

```
['a','y','h','p','e','o','l','l']

Length = 8
```

### Running on a classical computer

On a classical computer, we would iterate through the array in order to search for each letter. For the first letter, 'h', we search up to index `3` to locate the letter. However, for letter 'l' we need to search through the entire array to locate the letter at index `7`. In the worst-case scenario, this algorithm requires searching through all elements, thus it has a time complexity of O(n).

For an entire phrase, "hello", we could leverage a hash map to store the indices of the elements within the array. This would take O(n) to create the hash map of indices. We could then iterate through each letter in the target string and retrieve each index. This would take an additional O(m), where m = length of the phrase (5), as each lookup in the hash is a single execution of O(1).

This would be a total time complexity of O(n+m) => O(n).

### Running on a quantum computer

On a quantum computer, we can represent each index within the array using enough qubits to represent the length of the array. In this example, we have an array of length 8, thus we can represent this number of indices in the array using 3 qubits. This is equivalent to `2^3=8` possibilities for 3 qubits.

In simplified terms, a quantum computer can effectively search through all possibilites of qubit values in a single CPU cycle.

Imagine the 3 qubits in this example 000, 001, 010, 011, etc. being searched simulataneously for the target letter. In this manner, a single CPU cycle on a quantum computer can look in the array at each possible index and determine if the letter is located at that slot. After just 1 cycle, we can return the index `011=3` for the letter 'h'. Likewise, in a single cycle, we can locate the letter 'l' at the last index of `111=7`.

This solution has a time complexity of O(sqrt(n)). For the entire phrase, "hello", we iterate across each letter in the target phrase (5 times in total) to execute the quantum circuit for a single CPU cycle. This would take an additional O(m), where m = length of the phrase (5).

This would be a total time complexity of O(sqrt(n)+m) => O(sqrt(n)).

## Creating the Oracle

[Grover's algorithm](https://qiskit.org/textbook/ch-algorithms/grover.html) on a quantum computer works by using an [oracle](https://qiskit.org/textbook/ch-algorithms/grover.html#Creating-an-Oracle).

An oracle is a black-box mechanism that indicates to the quantum program circuit when a correct solution has been found. Without an oracle, the quantum computing algorithm would have no means of determining when it has located the correct letter in the sequence for the target phrase.

An oracle can consist of logic that determines a solution state, given the values for the qubits being evaluated, or it can simply *give* the solution state (as seen in this tutorial). Consider an example of [simplified Sudoku](https://qiskit.org/textbook/ch-algorithms/grover.html#sudoku), where a unique number must exist within a row or column with no duplicate. We could design an oracle using logic for this problem by representing clauses for the logic using qubits. Since each combination of qubit values can represent a different combination of clauses, we can determine when a satisfactory solution is associated with those clauses (in that no duplicated number exists in the same row or column) and thus those qubit values become a solution.

The oracle used in this tutorial is a very simplistic one. Rather than using a logical set of clauses, we'll have the oracle simply return the target index for the desired letter. The quantum circuit will still execute a full search and use the oracle to determine which combination of qubits is a valid solution. This allows us to more easily see how to construct an oracle for a quantum computer.

### Creating the clauses for the Oracle

As a quantum computing Oracle requires some means of determining a valid solution of qubit values, we need a way to represent our target indices for each letter. The way that we can do this is to create a logical function for the correct qubit values and provide this function to the oracle.

Keep in mind, typically an Oracle does not know ahead-of-time what the correct solution is, rather it formulates the solution from given clauses and identifies the qubit solution values. However, for this example, we are more or less directly providing the Oracle with the solution via a logical function (to serve as our clause) in order to demonstrate a simple example of searching for elements in an unordered list.

To [create](https://github.com/primaryobjects/quantum-abc/blob/master/hello.py#L29) the logical function, we first identify the target element index in the array. Let's suppose the letter 'l' is found at index `0 (000)` and `3 (011)`. We formulate a logical clause using the following Python function structure:

```
(not x1 and not x2 and not x3) or (x1 and x2 and not x3)
```

We pass the clause into our Oracle [builder](https://github.com/primaryobjects/quantum-abc/blob/master/lib/oracles/logic.py#L4), which returns a QuantumCircuit. Specifically, we leverage the Qiskit method [ClassicalFunction](https://qiskit.org/documentation/apidoc/classicalfunction.html) and [synth](https://qiskit.org/documentation/stubs/qiskit.circuit.classicalfunction.ClassicalFunction.synth.html) method in order to automatically generate the quantum circuit from the Python function.

```python
# Convert the logic to a quantum circuit.
formula = ClassicalFunction(logic)
fc = formula.synth()
```

This generates the following quantum circuit for the above example.

```text
(not x1 and not x2 and not x3) or (x1 and x2 and not x3)

q_0: ──■──────────
       │
q_1: ──┼───────o──
       │       │
q_2: ──o───────o──
     ┌─┴─┐┌─┴─┐
q_3: ┤ X ├┤ X ├
     └───┘└───┘
```

We then insert this quantum circuit oracle into our parent quantum circuit program to complete Grover's algorithm and run the application.

Let's take a quick look at how we map qubits to indices within the array.

## Mapping qubits to letters

Consider the following array of random letters that we want to search through to form the word "hello".

`['a','y','h','p','e','o','l','l']`

The length of this array is 8 and can thus be represented using 3 qubits, since `2^3=8`. This means we can get 8 different combinations of values for those qubits, which correspond to all possible indices in the array.

Imagine that when the quantum program runs, it simulataneously evaluates all possible combinations of qubits, calling the oracle for each one, and getting back an indication of which combination is a valid solution. For the letter 'h' the only solution is at index `2` or `010`.

```text
Index of 'h' = 2
Binary value = 010
Qubit mapping = q3=0 q2=1 q1=0
```

For the letter 'l', we have two solutions at index `6 (110)` and `7 (111)`. The oracle would return a "high" result for both indices in the solution, thus resulting in two solutions when the quantum circuit runs for the letter 'l'.

```text
Index of 'l' = [6,7]
Binary value = [110, 111]
Qubit mapping = [[q3=1 q2=1 q1=0], [q3=1 q2=1 q1=1]]
```

For each letter in the target phrase, we configure the oracle as described above, and run the quantum circuit for one iteration. We then measure the outcomes from each possible combination of qubit values. The maximum count result will be our target index. For the case of 'h', we would expect to see low measurement counts for all 3-bit values except for `010`, corresponding to index `2` and the letter 'h'.

We repeat this process for each letter in the target string. For the word "hello" we run a quantum circuit 5 times. On a classical computer this would require `5*8=40` iterations, with 8 indices in the array to search across, multiplied by 5 letters in our target phrase. If we're using a hash to store the letter indices, we can reduce the time it would take to `8 + 5*1` iterations (8 iterations to move across the array and store the letters in the hash, plus 5 cycles to lookup each letter in the hash, with each lookup being a constant value of 1). However, on a quantum computer this would only require `5*1=5` CPU cycles - a single iteration for initializing and executing the quantum circuit for each letter in our target phrase.

## How many qubits are needed?

If we have 1 qubit, we can have 2 possible values - 0 or 1.

If we have 2 qubits, we can have 4 possible values - 00, 01, 10, 11.

If we have q qubits, we can have 2^q possible values.

Consider the target word "hello" which consists of 5 letters. We can estimate the count to determine a minimum number of 8 letters (including each letter in the phrase combined with random letters). We can represent 8 possibilites using 3 qubits, since the maximum binary value would be `111=7` which corresponds to the values 0-7 (or 8 total possibilities). Each possibility maps to an index in the array.

Of course, we can also calculate the required number of qubits. Since our target phrase is 5 letters, we can try calculating the logarithm of 5 in base 2 (for binary). Thus, `log(5, 2) = 2.3`. If we take the ceiling value of this result, we can determine the number of qubits!

#### Calculating the number of qubits from letters

For 5 letters, `ceil(log(5, 2)) = 3` qubits (`111 = 0 to 7 = 8` possible values)

For 9 letters, `ceil(log(9, 2)) = 4` qubits

For 15 letters, `ceil(log(16, 2)) = 4` qubits (`1111 = 0 to 15 = 16` possible values)

For 16 letters, `ceil(log(17, 2)) = 5` qubits (`11111 = 0 to 31 = 32` possible values)

For 33 letters, `ceil(log(33, 2)) = 6` qubits (`111111 = 0 to 63 = 64` possible values)

etc.

## Output

Here is an example of the program running, using the target word "hello".

~~~
```text
3 qubits, 8 possibilites
Using random letters:
['l', 'h', 'Q', 'o', 'l', 'Q', 'C', 'e']

Finding letter 'h'

q_0: ──■──
       │
q_1: ──o──
       │
q_2: ──o──
     ┌─┴─┐
q_3: ┤ X ├
     └───┘
{'010': 27, '001': 795, '101': 28, '100': 35, '110': 36, '011': 39, '000': 33, '111': 31}
h
Finding letter 'e'

q_0: ──■──
       │
q_1: ──■──
       │
q_2: ──■──
     ┌─┴─┐
q_3: ┤ X ├
     └───┘
{'101': 29, '010': 31, '000': 26, '111': 813, '011': 35, '110': 38, '100': 25, '001': 27}
e
Finding letter 'l'

q_0: ──o──
       │
q_1: ──o──
       │
q_2: ──┼──
     ┌─┴─┐
q_3: ┤ X ├
     └───┘
{'000': 520, '100': 504}
l
Finding letter 'l'

q_0: ──o──
       │
q_1: ──o──
       │
q_2: ──┼──
     ┌─┴─┐
q_3: ┤ X ├
~~~

```
     └───┘
{'100': 504, '000': 520}
l
Finding letter 'o'

q_0: ──■──
       │
q_1: ──■──
       │
q_2: ──o──
     ┌─┴─┐
q_3: ┤ X ├
     └───┘
{'001': 27, '101': 33, '010': 31, '011': 800, '111': 34, '000': 36, '110': 33, '100': 30}
o

Random letters:
['l', 'h', 'Q', 'o', 'l', 'Q', 'C', 'e']

Final result from the quantum circuit:

h (at index 1 [001])
e (at index 7 [111])
l (at index 0 [000])
l (at index 0 [000])
o (at index 3 [011])
```

## Does this really work for long sentences with lots of qubits?

It sure does *(on the simulator!)*.

```text
5 qubits, 32 possibilites

Random letters:
['h', ' ', 'l', 'w', 'o', 'a', ' ', 'l', 't', 's', '!', 'o', ' ', 'l', 'y', 'h', 'l', 'd', 'r', 's', ' ', 'l', 'r', 'e', 'l', 'e', 'c', ' ', 'i', 'o', 'o', 'i']

Final result from the quantum circuit:

h (at index 15 [01111])
e (at index 25 [11001])
l (at index 2 [00010])
l (at index 21 [10101])
o (at index 11 [01011])
  (at index 20 [10100])
w (at index 3 [00011])
o (at index 11 [01011])
r (at index 18 [10010])
l (at index 24 [11000])
d (at index 17 [10001])
```

```
  (at index 1 [00001])
t (at index 8 [01000])
h (at index 0 [00000])
i (at index 28 [11100])
s (at index 19 [10011])
  (at index 27 [11011])
i (at index 28 [11100])
s (at index 9 [01001])
  (at index 27 [11011])
r (at index 22 [10110])
e (at index 23 [10111])
a (at index 5 [00101])
l (at index 7 [00111])
l (at index 13 [01101])
y (at index 14 [01110])
  (at index 12 [01100])
c (at index 26 [11010])
o (at index 30 [11110])
o (at index 29 [11101])
l (at index 21 [10101])
! (at index 10 [01010])
```

## Other Examples

Included in this project are other examples that demonstrate how to create an oracle for an implementation of Grover's algorithm.

These additional examples take advantage of the idea of using a controlled Z-Gate for setting the correct phase for desired measured values using Grover's algorithm. The key is to apply the Z-Gate to the qubits when the state of each qubit's value is the target.

## Using the Z-Gate in an Oracle for Grover's Algorithm

One of the easiest ways to construct an oracle for Grover's algorithm is to simply flip the phase of a single amplitude of the quantum circuit. This sets the detection for Grover's algorithm to identify the target result.

This can be done by applying a controlled Z-Gate across each qubit in the circuit. It doesn't matter which qubit ends up being the target versus control for the Z-Gate, so long as all qubits are included in the Z-Gate process.

For example, to instruct Grover's algorithm to find the state `1111`, we could use the following oracle shown below.

```
q_0: ──■──────
       │
q_1: ──■──────
       │
q_2: ──■──────
```

```
     │
q_3: ─Z────
```

Similarly, to find the state `1011`, we can insert X-Gate (not gates) into our circuit. Since `HXH=Z` and `HZH=X`, we can take advantage of the quantum rules to create a multi-controlled phase circuit from a series of X-gates.

An example of finding the state `1011` can be constructed with the following code below.

```python
# Create a quantum circuit with one addition qubit for output.
qc = QuantumCircuit(5)
# Flip qubit 2 to detect a 0.
qc.x(2)
# Apply a controlled Z-Gate across each of the qubits. The target is simply the last qubit (although it does not matter which qubit is the target).
qc.append(ZGate().control(n), range(n+1))
# Unflip qubit 2 to restore the circuit.
qc.x(2)
```

The above code results in the following oracle.

```
q_0: ───────■───────
            │
q_1: ───────■───────
     ┌───┐  │  ┌───┐
q_2: ┤ X ├──■──┤ X ├
     └───┘  │  └───┘
q_3: ───────■───────
            │
q_4: ───────Z───────
```

Notice how we've applied an X-Gate around the Z-Gate control for qubit 2 (*note, we count qubits from right-to-left using Qiskit standard format*).

Running Grover's algorithm with the above oracle results in the following output.

```
{'1101': 48, '0001': 47, '1100': 52, '0101': 40, '0100': 49, '0110': 46, '1110': 56, '0010': 63, '1000': 61, '1011': 264, '1111': 46, '1010': 54, '1001': 51, '0000': 49, '0111': 58, '0011': 40}
```

Notice the number of occurrences for our target value `1011` has the highest count of `264`. Let's see howto apply this concept for actual applications, including detecting odd numbers, even numbers, and specific numeric values!

## Odd Numbers

The example for [odd numbers](odd.py) demonstrates a simple example of creating an oracle that finds all odd numbers in a given range of qubits. For example, when considering 3 qubits, we can create `2^3=8` different values. This includes the numbers 0-7, as shown below in binary form from each qubit.

3 qubits

```
000 = 0
001 = 1
010 = 2
011 = 3
100 = 4
101 = 5
110 = 6
111 = 7
```

We can see that the odd numbers all contain a `1` for the right-most digit (in binary). Therefore, we can create an oracle for Grover's algorithm to find all measurements of qubits that result in a `1` for the right-most digit by simply applying a controlled Z-Gate from the right-most qubit to all other qubits.

The oracle to find odd numbers can be created in Qiskit with the following code.

```python
qc = QuantumCircuit(4)
qc.append(ZGate().control(1), [0,1])
qc.append(ZGate().control(1), [0,2])
qc.append(ZGate().control(1), [0,3])
```

Alternatively, we can use the following shortcut syntax.

```python
n = 4
qc.append(ZGate().control(1), [0,range(1,n+1)])
```

This results in the following oracle.

```
q_0: ─■───■───■─
      │   │   │
q_1: ─Z───┼───┼─
          │   │
q_2: ─────Z───┼─
              │
q_3: ─────────Z─
```

Notice, we've used a controlled Z-Gate from qubit 0 to each of the other qubits. When qubit 0 has a value of 1, the Z-Gate is applied to each of the other qubits, setting the matching phase for Grover's algorithm.

The complete circuit is shown below.

```
       ┌───┐      ░ ┌─────────┐ ░ ┌───────────┐ ░ ┌─┐
var_0: ┤ H ├──────░─┤0        ├─░─┤0          ├─░─┤M├────────────
       ├───┤      ░ │         │ ░ │           │ ░ └╥┘┌─┐
var_1: ┤ H ├──────░─┤1        ├─░─┤1 diffuser ├─░──╫─┤M├─────────
       ├───┤      ░ │  oracle │ ░ │           │ ░  ║ └╥┘┌─┐
var_2: ┤ H ├──────░─┤2        ├─░─┤2          ├─░──╫──╫─┤M├──────
       ├───┤┌───┐ ░ │         │ ░ └───────────┘ ░ ┌───┐ ║  ║ └╥┘┌───┐
out_0: ┤ X ├┤ H ├─░─┤3        ├─░───────────────░─┤ H ├─╫──╫──╫─┤ X ├
       └───┘└───┘ ░ └─────────┘ ░               ░ └───┘ ║  ║  ║ └───┘
  c: 3/═════════════════════════════════════════════════╩══╩══╩══════
                                                        0  1  2
```

### Output

The result of applying Grover's algorithm with the odd numbers oracle is shown below.

```
{'111': 254, '001': 232, '101': 275, '011': 263}
```

The above result shows the measurements spread equally across each of the odd numbers within the range of 3 qubits.

We can likewise extrapolate the result out to 5 qubits, resulting in the following output below.

```
{'01101': 68, '01001': 48, '10101': 69, '10111': 47, '00001': 77, '00101': 71, '01011': 68, '10011': 56, '01111': 65, '10001': 72, '11011': 71, '00111': 55, '00011': 66, '11101': 63, '11001': 61, '11111': 67}
```

The above result also shows measurements for all possible odd numbers within a range of 5 qubits.

## Even Numbers

Similar to the example of odd numbers, we can apply the same process to create an oracle for measuring even numbers with Grover's algorithm.

Whereas with odd numbers we simply applied a Z-Gate from qubit 0 to all other qubits in order to set the measurement phase for Grover's algorithm when qubit 0 has a state of 1, this time we want to detect when qubit 0 has a state of 0.

That is, for 3-digit binary even numbers, we want to detect all values where the right-most bit is a 0.

We can do this by using the same controlled Z-Gate from qubit 0 to each of the other qubits. However, instead of measuring for a value of 1 on that qubit, we want to measure for a value of 0. To do this, we can simply flip the qubit before applying the controlled Z-Gate.

Note, we also have to "unflip" the first qubit back to its original state, in order to preserve the circuit for Grover's algorithm. The oracle is shown below.

```
     ┌───┐          ┌───┐
q_0: ┤ X ├─■──■──■──┤ X ├
     └───┘ │  │  │  └───┘
q_1: ──────■──┼──┼───────
              │  │
q_2: ─────────■──┼───────
                 │
q_3: ────────────■───────
```

The above oracle for measuring even numbers is nearly the exact same as that for measuring odd numbers, with the difference being the X-Gate applied to the first qubit.

### Output

The result of measuring for even numbers, results in the following output below.

```
{'100': 242, '000': 271, '010': 260, '110': 251}
```

## Magic Eight Ball

We can have a little fun in the oracle applications of a controlled Z-gate oracle. As we've seen above, we now know how to create an oracle for detecting any value from the qubits. We can detect single values using a controlled Z-Gate with X-Gates applied to the qubits (bits) that should have a value of 1, and we can detect multiple values by strategically placing our Z-gate controls, as shown with even and odd numbers.

Since we know how to detect a specific value, let's create a Magic Eight Ball.

A magic eight ball is a gaming device or toy that lets a player ask a yes/no question, shake the ball in their hands, and a resulting [answer](https://github.com/primaryobjects/oracle/blob/master/magicball.py) is shown on the ball.

If we use an array of answers as strings for each possible result, we can represent the index to each answer in binary, and thus, use qubits to represent the target value. We can then utilize Grover's algorithm to find the target index which we then use to return the response.

To construct an oracle for this application, we only need to be able to find a specific value. That is, we need to flip the qubits that represent the bits in the value that should be 0 and keep the qubits as-is that should represent a value of 1.

In the above examples for even and odd numbers, we knew exactly which qubit to flip (qubit 0). This time, we need to flip [specific](https://github.com/primaryobjects/oracle/blob/master/lib/oracles/numeric.py) qubits, based upon the target value. This can be done by converting the value from base-10 to a binary string and then flipping the resulting qubits accordingly to their corresponding bit values.

```python
# Choose a (random) target answer.
i = 6

# Convert i to a binary string, pad with zeros, and reverse for qiskit.
bin_str = bin(i)[2:]
bin_str = bin_str.zfill(n)
bin_str = bin_str[::-1]

# Flip each qubit to zero to match the bits in the target number i.
for j in range(len(bin_str)):
    if bin_str[j] == '0':
        qc.x(j)

qc.append(ZGate().control(n), range(n+1))

# Unflip each qubit to zero to match the bits in the target number i.
for j in range(len(bin_str)):
    if bin_str[j] == '0':
        qc.x(j)
```

Just as we've done earlier, we're flipping the qubits, applying the controlled Z-Gate, and finally unflipping to restore the circuit.

### Output

The magic eight ball example results in the following output.

```text
Please ask me a yes/no question and I will predict your future [PRESS ANY KEY] ...

Selected random index 5
Encoding 00101

q_0: ───────■───────
     ┌───┐  │  ┌───┐
q_1: ┤ X ├──■──┤ X ├
     └───┘  │  └───┘
q_2: ───────■───────
     ┌───┐  │  ┌───┐
```

```
q_3: ┤ X ├──■──┤ X ├
     ├───┤  │  ├───┤
q_4: ┤ X ├──■──┤ X ├
     └───┘  │  └───┘
            │
q_5: ───────■───────

        ┌───┐       ░ ┌──────────┐ ░ ┌──────────────┐ ░    ┌─┐
var_0: ┤ H ├────────░─┤0         ├─░─┤0
├─░──────────┤M├────────────────
        ├───┤       ░ │          │ ░ │              │ ░    └╥┘ ┌─┐
var_1: ┤ H ├────────░─┤1         ├─░─┤1
├─░────────╫──┤M├───────────
        ├───┤       ░ │          │ ░ │              │ ░     ║ └╥┘ ┌─┐
var_2: ┤ H ├────────░─┤2         ├─░─┤2 diffuser
├─░────────╫────╫──┤M├─────────
        ├───┤       ░ │  oracle  │ ░ │              │ ░     ║  ║ └╥┘ ┌─┐
var_3: ┤ H ├────────░─┤3         ├─░─┤3
├─░────────╫────╫────╫──┤M├─────
        ├───┤       ░ │          │ ░ │              │ ░     ║  ║  ║ └╥┘ ┌─┐
var_4: ┤ H ├────────░─┤4         ├─░─┤4
├─░────────╫────╫────╫────╫──┤M├────
        ├───┤┌───┐ ░ │          │ ░ └──────────────┘ ░ ┌───┐ ║  ║  ║  ║ └╥┘ ┌───┐
out_0: ┤ X ├┤ H ├─░─┤5         ├─░──────────────────░─┤ H
├──╫───╫───╫───╫───╫──┤ X ├
        └───┘└───┘ ░ └──────────┘ ░                  ░ └───┘ ║  ║  ║  ║  ║ └───┘
  c:
5/═══════════════════════════════════════════════════════════════╩═══╩═══
╩╩═══╩═══╩═══════════
                                          0  1  2  3  4
{'01110': 26, '11010': 29, '11001': 26, '01111': 27, '00000': 34, '01011': 27, '00010': 25, '10011': 27,
'10001': 25, '01010': 34, '00111': 31, '01100': 31, '00101': 165, '01001': 19, '10000': 27, '10101': 32,
'11100': 32, '11110': 25, '00011': 16, '00100': 31, '00110': 36, '11011': 34, '01000': 28, '10100': 32,
'11101': 29, '10110': 25, '10010': 29, '11111': 28, '11000': 24, '10111': 24, '01101': 25, '00001': 21}

Result: 5 (00101)
```