# 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)

As I see it, yes.

```

### Notes on Included Oracles

This repository contains very simple oracles for Grover's algorithm, some of which already know the target value and encode it within the oracle circuit. While it may be trivial to choose a random answer from the array of strings upon which we obviously know the index, encoding the index as qubits, and then asking Grover's algorithm to find that index again for us, this example demonstrates how to use a controlled Z-Gate for this purpose.

Future applications should take this idea further to craft more unique and robust oracles for Grover's algorithm where the answer is not immediately known, but is rather determined through a set of clauses.

## License

MIT

## Author

Kory Becker

http://www.primaryobjects.com/kory-becker