In [1]:
```
%run utils.py
```

COMS 4281 - Intro to Quantum Computing

Problem Set 3, Quantum Algorithms

Due: November 2, 11:59pm.

Collaboration is allowed and encouraged (teams of at most 3). Please read the syllabus carefully for the guidlines regarding collaboration. In particular, everyone must write their own solutions in their own words.

Name: Chandan Suri, UNI: CS4090

Write your collaborators here:

AS6413

Problem 1: Basic Fourier Math

Problem 1.1

For two strings $x, y \in \{0,1\}^n$, let $x \cdot y$ denote

$$x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \bmod 2.$$

Prove that for all $x \in \{0,1\}^n$,

$$\sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} = \begin{cases} 0 & \text{if } x \neq 0^n \\ 2^n & \text{if } x = 0^n \end{cases}$$

Solution

Considering both the cases:

If $x = 0^n$,

As $x_1 = x_2 = x_3 = ... = x_n = 0$, $x.y = 0 \bmod 2 = 0$.

This makes the summation as:

$$\sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} = \sum_{y \in \{0,1\}^n} (-1)^0 = \sum_{y \in \{0,1\}^n} 1$$

As total possible combinations of y can be 2^n, so, the 1 will be summed up that many times which can be computed to be:

$$\sum_{y \in \{0,1\}^n} 1 = 2^n$$

Coming to the 2nd case if $x \neq 0^n$,

We will prove the same by induction:

Base case: when n = 1, this means that $x_1 \neq 0$, therefore $x_1 = 1$.

Then $x.\, y = x_1 y_1 \bmod 2 = 1.y_1 \bmod 2 = y_1 \bmod 2$

Thus, the summation becomes:

$$\sum_{y \in \{0,1\}^1} (-1)^{y \bmod 2}$$

Since, y can take on 2 values, this becomes:

$$\sum_{y \in \{0,1\}^1} (-1)^{y \bmod 2} = (-1)^{0 \bmod 2} + (-1)^{1 \bmod 2} = 0$$

Now, for the inductive case, let's take the assumption for some k, when $x \neq 0^k$,

let:

$$\sum_{y \in \{0,1\}^k} (-1)^{x \cdot y} = 0$$

So, for $n = k + 1$,

$$\sum_{y \in \{0,1\}^{k+1}} (-1)^{x \cdot y} = \sum_{y \in \{0,1\}^k} (-1)^{x \cdot y} + \sum_{y_{k+1} \in \{0,1\}} (-1)^{x_{k+1} \cdot y_{k+1}}$$

Here, y_{k+1} can take values similar to the one in the base case which makes the second term zero and the first term being zero was our assumption which makes this as:

$$\sum_{y \in \{0,1\}^{k+1}} (-1)^{x \cdot y} = 0$$

Therefore, by induction we have proven that:

$$\sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} = 0$$

when $x \neq 0^n$.

Thus, we have proven both the cases as shown above.

Problem 1.2

Let $\omega_n = \exp\left(\frac{2\pi i}{n}\right)$ denote the n-th root of unity. Prove that for all integers j,

$$\sum_{k=0}^{n-1} \omega_n^{jk} = \begin{cases} 0 & \text{if } j \text{ is not a multiple of } n \\ n & \text{if } j \text{ is a multiple of } n \end{cases}$$

Solution

Considering both the cases:

If j is a multiple of n:

we can write $j = an$ where a is any integer.

Furthermore,

$$\sum_{k=0}^{n-1} \omega_n^{jk} = \sum_{k=0}^{n-1} \omega_n^{ank} = \sum_{k=0}^{n-1} (\omega_n^n)^{ak}$$

Since, $\omega_n^n = 1$, we have the same as:

$$= \sum_{k=0}^{n-1} 1^{ak} = \sum_{k=0}^{n-1} 1 = n$$

Thus, we have proven the first case when j is a multiple of n.

Going over to the second case when j is not a multiple of n:

What we get here is basically the geometric progression which can be computed as follows:

$$\sum_{k=0}^{n-1} \omega_n^{jk} = \omega_n^0 * \frac{\omega_n^{jn} - 1}{\omega_n^j - 1}$$

Again, since $\omega_n^n = 1$, the numerator in the formulation above becomes zero thus giving us the following:

$$\omega_n^0 * \frac{\omega_n^{jn} - 1}{\omega_n^j - 1} = \omega_n^0 * \frac{1^j - 1}{\omega_n^j - 1} = 0$$

Thus, we have proven the second case as well where j is not a multiple of n.

Problem 1.3

Let N denote a positive integer between 2^{n-1} and 2^n, and let $1 \leq x \leq N-1$ denote an integer such that $\gcd(x, N) = 1$ (meaning that x is coprime to N). Let r be the order of x modulo N; meaning that $x^r \equiv 1 \bmod N$ (i.e., dividing x^r by N yields a remainder of 1). Recall the unitary map acting on n qubits such that for all $0 \leq y < 2^n$,

$$U_x \left|y\right\rangle = \begin{cases} \left|xy \bmod N\right\rangle & \text{if } y < N \\ \left|y\right\rangle & \text{otherwise} \end{cases}$$

and recall the eigenvectors

$$\left|u_s\right\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \omega_r^{ks} \left|x^k \bmod N\right\rangle \ .$$

Show that

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \left|u_s\right\rangle = \left|1\right\rangle \ .$$

Solution

Let's take the formulation above to prove:

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \left|u_s\right\rangle$$

Substituting the value of $\left|u_s\right\rangle$ gives us:

$$\frac{1}{r} \sum_{s=0}^{r-1} \sum_{k=0}^{r-1} \omega_r^{ks} \left|x^k \bmod N\right\rangle$$

Taking out the term for $k = 0$ as that's the simple part of the equation gives us:

$$\frac{1}{r}(1 + 1 + \ldots + 1 \text{ r times}) \left|1\right\rangle + \frac{1}{r} \sum_{s=0}^{r-1} \sum_{k=1}^{r-1} \omega_r^{ks} \left|x^k \bmod N\right\rangle$$

This gives us the following:

$$\frac{1}{r} . r \left|1\right\rangle + \frac{1}{r} \sum_{s=0}^{r-1} \sum_{k=1}^{r-1} \omega_r^{ks} \left|x^k \bmod N\right\rangle = \left|1\right\rangle + \frac{1}{r} \sum_{s=0}^{r-1} \sum_{k=1}^{r-1} \omega_r^{ks} \left|x^k \bmod N\right\rangle$$

Rearranging the summations in the second term gives us the following:

$$\left|1\right\rangle + \frac{1}{r} \sum_{k=1}^{r-1} \left|x^k \bmod N\right\rangle \sum_{s=0}^{r-1} \omega_r^{ks}$$

The second summation basically gives us a geometric progression, using the formulation for the same gives us:

$$|1\rangle + \frac{1}{r}\sum_{k=1}^{r-1}\left|x^k \bmod N\right\rangle \left(\omega_r^0 * \frac{\omega_r^{rk}-1}{\omega_r^k-1}\right)$$

Now, since $\omega_r^r = 1$, the geometric progression sums to a value of zero as shown below:

$$|1\rangle + \frac{1}{r}\sum_{k=1}^{r-1}\left|x^k \bmod N\right\rangle * 0 = |1\rangle$$

Thus, we prove that:

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|u_s\rangle = |1\rangle$$

Problem 1.4

Recall the Fourier Transform unitary F_N that acts as follows: for all $0 \leq j < N$,

$$F_N\left|f_j\right\rangle = |j\rangle$$

where

$$\left|f_j\right\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}\omega_N^{-jk}\ |k\rangle\ .$$

Compute the vector $F_N\left|j\right\rangle$.

Note: The definition of $\left|f_j\right\rangle$ differs from the one presented in class, because the exponent of ω_N is $-jk$ rather than jk.

Solution

As taught in class, we know that the Fourier Transform unitary F_N is basically the change of basis matrix here.

This would give us the obvious fact that the (m,n)th entry in F_N ($F_{m,n}$) is given by the coefficients of $|m\rangle$ when the $\left|f_n\right\rangle$ is written in terms of the standard basis. Even with the negative sign in the exponent, using the expression for $\left|f_j\right\rangle$ it basically gives us that F_N is symmetric as $F_{m,n} = F_{n,m} = \omega_N^{mj}$.

Translating the indices for F_N as i and j and based on the above finding, we can write the expressions as follows:

$$F_N^T F_N\left|f_j\right\rangle = F_N^T\left|j\right\rangle$$

Since the omega values are roots of unity and multiplying the transpose of a matrix with itself gives us an indentity matrix (also as F_N is symmetric, the transpose will essentially be same as the matrix itself) as shown below:

$$I. \left|f_j\right\rangle = F_N \left|j\right\rangle$$

This gives us:

$$\left|f_j\right\rangle = F_N \left|j\right\rangle$$

Therefore, we find that:

$F_N \left|j\right\rangle$ can be computed as the vector $\left|f_j\right\rangle$.

Problem 2: Modified Simon's Algorithm

In this problem you will analyze a variant of Simon's Problem. Suppose you are given black-box query access to a function $f : \{0,1\}^n \to \{0,1\}^n$ that encodes **two** secrets $s, t \in \{0,1\}^n$. This means that

$$f(x) = f(y) \qquad \text{if and only if} \qquad x + y \in \{0^n, s, t, s+t\}$$

We will assume that s, t are both nonzero and are not equal to each other.

Note: the sum $x + y$ of two bit strings is another string where we've XOR'd the coordinates of x , y.

Problem 2.1

Given such a function f, what is the size of its range? Meaning, how many possible strings can be output by f?

Solution

For calculating the size of it's range, I will be taking some assumptions:

If s, t are both non-zero and unique as given in the question (assumption), then for any $x \in \{0,1\}^n$:

$x, x+s, x+t, x+s+t$ are distinct elements in $\{0,1\}^n$

Also, for the function f, if we are taking these secrets into consideration, then $x, x+s, x+t, x+s+t$ are mapped to the same value of f since $f(x) = f(y)$ for these.

Additionally, no other bit string with n elements can be mapped to this same value.

Therefore, the range of f contains 2^{n-2} n-bit strings.

Problem 2.2

Write a classical + quantum hybrid algorithm that makes $\text{poly}(n)$ queries to f and outputs, with high probability, the secrets s, t. You should use Simon's Algorithm for inspiration.

Solution

A classical + quantum hybrid algorithm can be derived from Simon's algorithm itself.

Going forward if we keep the Simons subroutine to be the same while only changing the classical part of post-processing. As now, we have two secrets encoded here, each query or call made to the circuit will basically yield 2 linear equations instead of just one. These linear equations will be of the form:

$$y \cdot s = 0$$

and,

$$y \cdot t = 0$$

As we can see above, the number of linear equations will be doubled along with the number of variables being doubled. As we have a system of linear of equations, now solving which would give us the result accordingly. So, only the classical post-processing part changes here essentially.

Furthermore, this also means that this algorithm with modification will also yield the results with high probability as the original algorithm.

Problem 2.3

Prove that your algorithm works.

Solution

Following the formulation for Simon's algorithm from class, the state before measurement will be:

$$\frac{1}{2^n} \sum_x \sum_y (-1)^{x \cdot y} |y\rangle |f(x)\rangle$$

If we consider A to be the range of function f, we can re-write the above state before measurement as:

$$\frac{1}{2^n} \sum_{a \in A} \sum_y [(-1)^{x_a \cdot y} + (-1)^{(x_a + s) \cdot y} + (-1)^{(x_a + t) \cdot y} + (-1)^{(x_a + s + t) \cdot y}] |y\rangle |a\rangle$$

Now, if we do the measurement, then measuring the qubits corresponding to a will yield the following:

$$\frac{1}{2^n}\sum_y[(-1)^{x_a.y}+(-1)^{(x_a+s).y}+(-1)^{(x_a+t).y}+(-1)^{(x_a+s+t).y}]\,|y\rangle$$

As we can see, the $(-1)^{x_a.y}$ is common to all the terms inside the summation which can be factorized out giving us the following:

$$\frac{1}{2^n}\sum_y[(-1)^{x_a.y}(1+(-1)^{s.y}+(-1)^{t.y}+(-1)^{(s+t).y})]\,|y\rangle$$

Factoring it further will give us the following:

$$\frac{1}{2^n}\sum_y(-1)^{x_a.y}\left[1+(-1)^{t.y}\right]\left[1+(-1)^{s.y}\right]|y\rangle$$

Following the formulation above:

If $s.y\neq 0$, then $s.y=1$ which would make the complete expression zero yielding us nothing.

Next, if $t.y\neq 0$, then $t.y=1$ which would make the complete expression again zero yielding us nothing.

Therefore, after we measure the n qubits, we will only see y when all the system of equations are satisfied meaning $y.s=0$ and $y.t=0$.

Therefore, we have proven that our algorithm will work.

Problem 2.4

What if we now considered the situation where the function f is hiding not two but k secrets $s^{(1)},\ldots,s^{(k)}$ (which are all nonzero and distinct strings). How would your answers to 7.2 and 7.3 change?

Solution

When we have k secrets, our algorithm again won't change a lot.

Again, the Simon's subroutine will stay the same and only the classical part of post-processing will change. Now, each measurement will give us k linear equations rather than just two.

The proof of the algorithm will also mostly remain the same, the only change will be in the final post measurement state in the form of the range of the function as follows:

$$\frac{1}{2^n}\sum_y(-1)^{x_a.y}\prod_{i=1}^{k}[1+(-1)^{s^{(i)}.y}]\,|y\rangle$$

Problem 3: Quantum Minimum Finding

Problem 3.1

Let $M \geq 2$ be an integer. Devise a quantum algorithm that, in expectation, makes $O(\sqrt{N})$ queries to a function $f : \{0,1\}^n \to \{0,1,\ldots,M-1\}$, and computes $\min_x f(x)$. You can assume that f is injective.

Hint: Use that Grover's algorithm, when run on a function $g : \{0,1\}^n \to \{0,1\}$ with T solutions, outputs a uniformly random x such that $g(x) = 1$ using $c\sqrt{N/T}$ queries in expectation for some constant c.

Hint: Your algorithm can have unbounded running time; you just need to show that the *expected* number of queries before the algorithm finds the minimum is $O(\sqrt{N})$.

Solution

The algorithm derived is as follows:

1. Choose an initial threshold y in the range $0 \leq y \leq 2^n$.
2. As Grover's algorithm works with boolean function, we will chhange our function to a boolean one such that it minimizes $f(x)$ as shown below:

$$g(f(x)) = \begin{cases} 1 & \text{if f(x) <= f(y)} \\ 0 & \text{otherwise} \end{cases}$$

3. Although it's given that the function is injective, we will need to reduce our solution space one by one and thus, reduce the span of candidate solutions. Using the hint for Grover's algorithm as stated above for T solutions, we will need to make $c\sqrt{N/T}$ queries to the oracle. Initially, we take the value of T to be any random value between 0 and $\sqrt{N}$.
4. We will use grover's iterate here involving the oracle calls and the diffusion gate in pairs. We will append the oracle and diffusion gate (in pairs) to the circuit $\sqrt{N/T}$ number of times. This basically means that we run the grover's algorithm some constant number of times with the algorithm's complexity as $O(\sqrt{N})$. The oracle used is shown below:

$$O_f = (-1)^{g(f(x))} |x\rangle$$

5. We will look at the output of the Grover's algorithm (y') and check if $f(y') < f(y)$ and we update y if that's the case. This wouldn't necessarily give us the global minima and thus, we might have to repeat the step for smaller set of candidate solutions.
6. We will repeat step 4 and 5 above some constant number of times which should give us the correct result with high probability. In the proposed solution above, we make at most $\sqrt{N}$ calls to the oracle and we just need to run the same some constant number of times in order to get the global minima with high probability. Thus, the complexity of the algorithm will be $O(\sqrt{N})$.

In our implementation below in Problem 3.3, I halve the T value after each iteration so as to get some factor of repeats. This version of implementation doesn't produce the minima with 100

probability but gives us the same with some high probability. For getting the minima with even higher probability, we increase the constant number of times we have to make calls to the algorithm as a whole while maintaining the $O(\sqrt{N})$ calls to the oracle. Thus, the expected number of calls or queries made before we get the minima will still remain of the order $O(\sqrt{N})$. A very similar algorihtm published by Christoph Durr and Peter Hoyer was shown to find the minima with $22.5 * O(\sqrt{N})$ complexity with the constant shown there. The paper for the same can be found here: https://arxiv.org/pdf/quant-ph/9607014.pdf

Problem 3.2

Show that, assuming that Grover's algorithm is optimal for unstructured search (meaning that any quantum algorithm finding a preimage of $f : \{0,1\}^n \to \{0,1\}$ requires $\Omega(\sqrt{N})$ queries to f), any quantum algorithm that finds the minimum of a function with high probability must also make $\Omega(\sqrt{N})$ queries.

Solution

First of proving the same would be through contradiction:

Assuming that the grover's algorithm is optimal for unstructured search problems and that there exits a quantum algorithm that finds the minima a function with high probability but makes less than $\Omega(\sqrt{N})$ queries. Following the same, finding the minima of a function with high probability will not fall under the umbrella of unstructured search. Let's move forward then:

To prove the same, let's analyze the problem under consideration:

If we have a function f such that it is a strinctly increasing function, then we know for sure that f is injective. We can find the minima of this function for all x and y as if $x < y$, then, $f(x) < f(y)$. Our algorithm would still make an order of $\sqrt{N}$ queries to the oracle while a classical algorithm will be able to give the solution in constant time. However, this kind of problem falls under the structured search problems as the function has a structure to it.

Furthermore, since we know that there exists an algorithm that finds the minima of a function with high probability but makes a single query to the oracle function f. However, if we solely focus on unstructured search problem as there is no structure to the data here (meaning strictly increasing or strictly decreasing), then assuming that grover's algorithm is the optimal one for unstructured search, then we can show the same that there exists a quantum algorithm that will have to make $\Omega(\sqrt{N})$ queries and we cannot go below this. Thus, for any quantum algorithm that finds the minima of a function with high probability must also make $\Omega(\sqrt{N})$ queries.

Another way of proving the same is described below:

As we have done above, we know that an unstructured search problem can be converted to a minima finding algorithm. We can do so by actually mapping the value of 0 (output of the function) to the marked input that is the minima (preimage value) and mapping all other values

greater than the marked input (in this case that's $\{1, 2, \ldots, N-1\}$) as to the other bit strings of the function (the pre-images). Now, we know that if this kind of unstructured search problem can be solved with less than $\Omega(\sqrt{N})$ queries to the oracle by any other quantum algorithm, then that would imply that Grover's algorithm is not an optimal algorithm for unstructured search problem which is our assumption for the proof.

So, in the first part above, we contradicted one part of the assumption and proved that it's wrong and in the second part, we contradicted another part of the assumption and disproved as well. This gives us the complete proof that any quantum algorithm that finds the minimum of a function with high probability must also make $\Omega(\sqrt{N})$ queries.

Problem 3.3

Now let's implement our minimum finding algorithm for arbitrary 5 qubit functions. Below are helpers for converting functions into quantum oracles, and an implementation of the Grover's Diffusion gate, use those to help you implement minimum search, and test is on the examples given below.

For fun, track the number of oracle calls you use (num_oracle_calls), but you won't lose points for using too many oracle calls if your solution is correct asymptotically.

In [2]:

```
num_oracle_calls = 0

def inc_num_oracle_calls():
    global num_oracle_calls
    num_oracle_calls += 1

def reset_num_oracle_calls():
    global num_oracle_calls
    num_oracle_calls = 0

def append_oracle(input_circuit: QuantumCircuit,
                  f: Callable[[int], bool],
                  quantum_registers: List[int]) -> QuantumCircuit:
    '''
    append_oracle takes a boolean function (with inputs from 0 to 31) and append
    '''
    inc_num_oracle_calls()
    oracle_gate = UnitaryGate(np.diag([(-1)**f(i) for i in range(32)]))
    input_circuit.append(oracle_gate, quantum_registers)
    return input_circuit

def append_diffusion_gate(input_circuit: QuantumCircuit,
                          quantum_registers: List[int]) -> QuantumCircuit:
    '''
    Takes an input circuit and appends a diffusion gate to the registers.
    The quantum registers must be length 5.
    '''
    if len(quantum_registers) != 5:
        return None
    for i in quantum_registers:
        input_circuit.h(i)
    diffuse = -1*np.eye(32)
```

```
    diffuse[0,0] = 1
    input_circuit.append(UnitaryGate(diffuse), quantum_registers)
    for i in quantum_registers:
        input_circuit.h(i)
    return input_circuit

def get_most_likely_outcome(quantum_circuit: QuantumCircuit) -> int:
    backend = Aer.get_backend('qasm_simulator')
    job_sim = backend.run(transpile(quantum_circuit, backend), shots=5024)
    result_sim = job_sim.result()
    counts = result_sim.get_counts(quantum_circuit)
    return int(max(counts, key = counts.get), 2)
```

Solution

In [55]:

```
### ======== BEGIN CODE =============
'''
Below are some suggested templates for functions that would be useful to impleme
You don't have to use them if you don't want

'''

def grovers_search_known_sols(f: Callable[[int], bool], num_sols: int) -> int:
    '''
    Runs Grovers search, assuming there are num_sols number of solutions (we'll
    Assumes that the input space is dimension 32.
    '''

def grovers_search(f: Callable[[int], bool]) -> int:
    '''
    Runs Grovers search with an unknown number of solutions by halving the guess
    Returns -1 if no answer is found.
    '''

### ======== END CODE =============

def find_min(f: Callable[[int], int]) -> int:
    reset_num_oracle_calls()
    y = random.randrange(0, 32)
    n = 5

    ### ======== BEGIN CODE =============

    T = random.randrange(0, int(2**n))

    while T > 0:
        qr = QuantumRegister(n, 'q')
        cr = ClassicalRegister(n, 'c')
        circuit = QuantumCircuit(qr, cr)

        circuit.h(qr)
        num_iters = int(math.sqrt(2**n/T))
        for counter in range(num_iters):
            circuit = append_oracle(circuit, lambda x: f(x) < f(y), qr)
            circuit = append_diffusion_gate(circuit, qr)

        circuit.measure(qr, cr)
        y_new = get_most_likely_outcome(circuit)
```

```
        if f(y_new) < f(y):
            y = y_new

        T = T//2

    ### ======== END CODE =============

    return y
```

Test your code on the following examples.

In [56]:
```
f1 = lambda x: x - 16
print("Minimum of f2 is: ", find_min(f1))
print("Oracle calls used: ", num_oracle_calls)
```

```
Minimum of f2 is:  0
Oracle calls used:  13
```

In [59]:
```
f2 = lambda x: x**2 - 13*x + 3
print("Minimum of f2 is: ", find_min(f2))
print("Oracle calls used: ", num_oracle_calls)
```

```
Minimum of f2 is:  6
Oracle calls used:  13
```

In [61]:
```
f3 = lambda x: 1.0 - np.sin(x / 16)
print("Minimum of f3 is: ", find_min(f3))
print("Oracle calls used: ", num_oracle_calls)
```

```
Minimum of f3 is:  25
Oracle calls used:  11
```

Problem 4: Exploring Order Finding

In this problem, we'll explore the quantum algorithm for Order Finding (which is different from the one covered in class; in fact this one is closer to the way Peter Shor implemented it). Consider the following function

$$f(x) = 3^x \bmod 16$$

where x ranges from 0 to 15, so that x can be represented using 4 bits.

Problem 4.1

What is the period r of this function? Meaning, what value of r is such that $f(x+r) = f(x)$ for all x?

Solution

x f(x)

0	1
1	3
2	9
3	11
4	1
5	3
6	9
7	11
8	1
9	3
10	9
11	11
12	1
13	3
14	9
15	11

As we can see above, by calculating f(x) for all the x's possible, we can see a repeated pattern for $f(x)$. As $f(0+4) = f(0)$ and so on and so forth according to the table above, this translates to a generic fact that for this $f(x)$, $f(x+4) = f(x)$. This means that period r of the function is equal to 4.

Problem 4.2

Let's see how we could've figured this out using quantum computation! First, we create a quantum circuit with 8 qubits and 4 classical bits to store measurement outcomes.

Add gates to the order finding circuit to get to the following state:

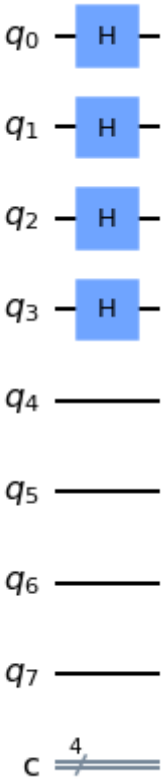
$$\frac{1}{\sqrt{16}} \left(\sum_{i=0}^{15} |i\rangle \right) \otimes |0\rangle$$

Note that here $|i\rangle$ and $|0\rangle$ refer to integers, but in your circuit they will need to be represented in binary using 4 bits.

In [2]:

```
order_finding_circuit = QuantumCircuit(8,4)

### ======== BEGIN CODE =============
order_finding_circuit.h([0, 1, 2, 3])
### ======== END CODE =============

order_finding_circuit.draw(output="mpl")
```

Out[2]:



Problem 4.2

We provide code that implements the U_f unitary, which acts on 8 qubits (4 for input, 4 for ancilla) as follows:

$$U_f \left|i\right\rangle \left|j\right\rangle = \left|i\right\rangle \left|j \oplus f(i)\right\rangle$$

Append the oracle unitary to your circuit to get the following state

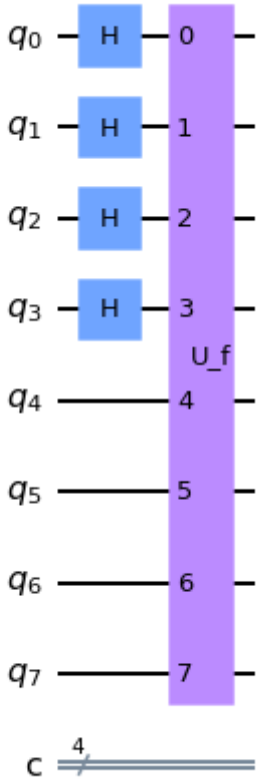
$$\frac{1}{\sqrt{16}} \left(\sum_{i=0}^{15} \left|i\right\rangle \otimes \left|3^i \bmod 16\right\rangle \right)$$

In [3]:

```
def reverse_bits(i: int) -> int:
    '''
    Reverses the bits of a number up to 5 bits (0 through 31)
    '''
    return int('{:08b}'.format(i)[::-1], 2)

#This function returns the UnitaryGate object that you want to append to your ci
def U_f() -> UnitaryGate:
    unitary = np.zeros([256, 256])
    for i in range(16):
        for j in range(16):
            input_basis_state = i * 16 + j
            exponentiated_j = pow(3, i, 16) ^ j
            output_basis_state = i * 16 + exponentiated_j
            unitary[reverse_bits(input_basis_state), reverse_bits(output_basis_s
```

```
    return UnitaryGate(unitary,"U_f")

### ======== BEGIN CODE =============
order_finding_circuit.append(U_f(), list(range(8)))
### ======== END CODE =============

order_finding_circuit.draw(output="mpl")
```

Out[3]:



Problem 4.3

Suppose you measured the second register (the last 4 qubits) of the state described in Problem 4.2. What are the possible outcomes and their probabilities? What are the corresponding post-measurement states?

Solution

The first four qubits (or the first register) is basically entangled state of $|+\rangle$ n times. When we put the unitary on the 8 qubits, the first 4 bits that are entangled are put through that unitary which gives us the formulation in problem 4.2 as follows:

$$\frac{1}{\sqrt{16}}\left(\sum_{i=0}^{15}|i\rangle\otimes\left|3^i \bmod 16\right\rangle\right)$$

Expanding the summation for all the elements gives us (each number is represented as 4 bits):

$$= \frac{1}{\sqrt{16}}(|0000\rangle \otimes |0001\rangle + |0001\rangle \otimes |0011\rangle + |0010\rangle \otimes |1001\rangle + |0011\rangle \otimes |1011\rangle + |0100\rangle \otimes$$
$$\otimes |0011\rangle + |1010\rangle \otimes |1001\rangle + |1011\rangle \otimes |1011\rangle + |1100\rangle \otimes |00$$

Pairing up the first register according to second register values (and normailizing the values inside the brackets) gives us:

$$= \frac{\sqrt{4}}{\sqrt{16}}\left(\frac{(|0000\rangle + |0100\rangle + |1000\rangle + |1100\rangle)}{\sqrt{4}}\right) \otimes |0001\rangle + \frac{\sqrt{4}}{\sqrt{16}}\left(\frac{(|0001\rangle + |0101\rangle + |10($$
$$+ \frac{\sqrt{4}}{\sqrt{16}}\left(\frac{(|0011\rangle + |0111\rangle + |10$$

Solving further this gives us:

$$= \frac{1}{\sqrt{4}}\left(\frac{(|0000\rangle + |0100\rangle + |1000\rangle + |1100\rangle)}{\sqrt{4}}\right) \otimes |0001\rangle + \frac{1}{\sqrt{4}}\left(\frac{(|0001\rangle + |0101\rangle + |1001\rangle$$
$$+ \frac{1}{\sqrt{4}}\left(\frac{(|0011\rangle + |0111\rangle + |1011\rangle$$

So, when we measure, the possible outcomes with the probabilities and corresponding post measurement states are as follows:

Outcome 1:

$|0001\rangle$ with probability of 25 and corresponding post measurement state as $\frac{(|0000\rangle+|0100\rangle+|1000\rangle+|1100\rangle)}{\sqrt{4}}$ which can also be written as $\frac{(|0\rangle+|4\rangle+|8\rangle+|12\rangle)}{\sqrt{4}}$

Outcome 2:

$|0011\rangle$ with probability of 25 and corresponding post measurement state as $\frac{(|0001\rangle+|0101\rangle+|1001\rangle+|1101\rangle)}{\sqrt{4}}$ which can also be written as $\frac{(|1\rangle+|5\rangle+|9\rangle+|13\rangle)}{\sqrt{4}}$

Outcome 3:

$|1001\rangle$ with probability of 25 and corresponding post measurement state as $\frac{(|0010\rangle+|0110\rangle+|1010\rangle+|1110\rangle)}{\sqrt{4}}$ which can also be written as $\frac{(|2\rangle+|6\rangle+|10\rangle+|14\rangle)}{\sqrt{4}}$

Outcome 4:

$|1011\rangle$ with probability of 25 and corresponding post measurement state as $\frac{(|0011\rangle+|0111\rangle+|1011\rangle+|1111\rangle)}{\sqrt{4}}$ which can also be written as $\frac{(|3\rangle+|7\rangle+|11\rangle+|15\rangle)}{\sqrt{4}}$

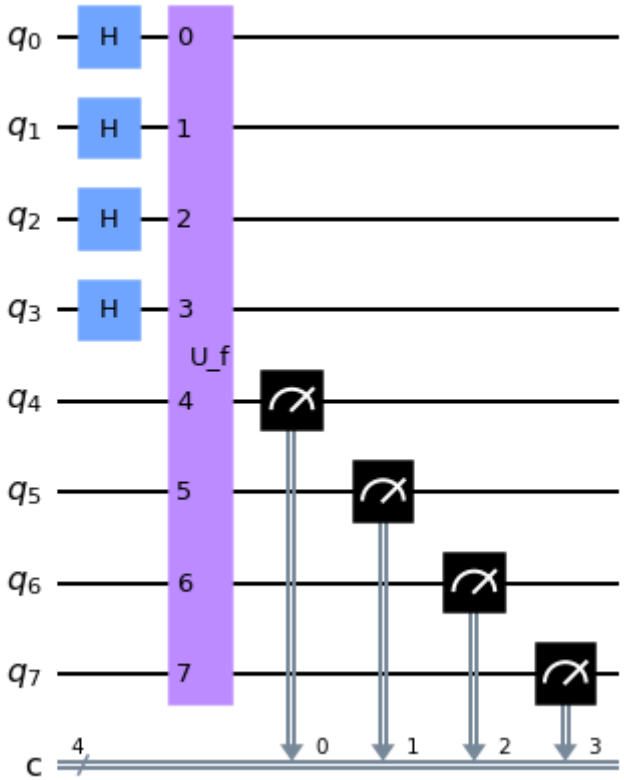
Problem 4.4

Let's simulate the measurement of the second register and collect measurement statistics. Write code to output the empirical probability of each measurement outcome (for example, the integer 1 appears $X\%$ of the time, the integer 3 occurs $Y\%$ of the time, etc.).

Note that the convention in Qiskit is to use little-endian representation of integers: the least significant bit goes first (so the ordering goes like $|0000\rangle$, $|1000\rangle$, $|0100\rangle$, $|1100\rangle$, . . .).

In [4]:
```
order_finding_circuit.measure([4,5,6,7], [0,1,2,3])
order_finding_circuit.draw(output="mpl")
```

Out[4]:



In [10]:
```
backend = Aer.get_backend("statevector_simulator")
result = execute(order_finding_circuit, backend=backend, shots=1024).result()
result.get_counts()
# Note that the bitstrings are reverse order
```

Out[10]:
```
{'1101': 254, '1001': 247, '1100': 243, '1000': 280}
```

In [11]:
```
### ======== BEGIN CODE =============
total_count = 1024
counts_data = result.get_counts()

for bin_num, count in counts_data.items():
    counter = 0
    final_num = 0
    for char in bin_num:
        final_num += int(char) * (2**counter)
        counter += 1
```

```
    prob = round((count / total_count) * 100, 2)
    print(f"{final_num} occurs for {prob}% of the time!")

### ======== END CODE =============
```

```
11 occurs for 24.8% of the time!
9 occurs for 24.12% of the time!
3 occurs for 23.73% of the time!
1 occurs for 27.34% of the time!
```

Problem 4.5

Now let's get the post-measurement state *conditioned* on the second register measuring $|j\rangle$ for $j = 3$. First, suppose we measure the second register and obtain outcome $|3\rangle$. What will the post measurement state on the first register be?

Solution

When the second register is measured to be $|3\rangle$ then, the post measurement state as calculated in problem 4.3 will be:

$\frac{(|0001\rangle+|0101\rangle+|1001\rangle+|1101\rangle)}{\sqrt{4}}$ which can also be written as $\frac{(|1\rangle+|5\rangle+|9\rangle+|13\rangle)}{\sqrt{4}}$

Problem 4.6

Now let's validate empirically that the post measurement state is what we found above. We've already got some skeleton code for you set up; you should fill in the rest. The result is that the variable `postmeasurement_state` should be the post measurement state of the 8 qubits, conditioned on the second register being in the state $|3\rangle$ (or, in binary, $|1100\rangle$).

In [12]:
```
j = 3
j_in_rev_binary = '1100' # Note this bitstring is in reverse order. Reversing it

#just repeating this 100 times until you get the result you want (the number 100
for k in range(100):
    result = execute(order_finding_circuit, backend=backend, shots=1).result()
    if list(result.get_counts().keys())[0] == j_in_rev_binary:
        break

postmeasurement_state = result.get_statevector()
```

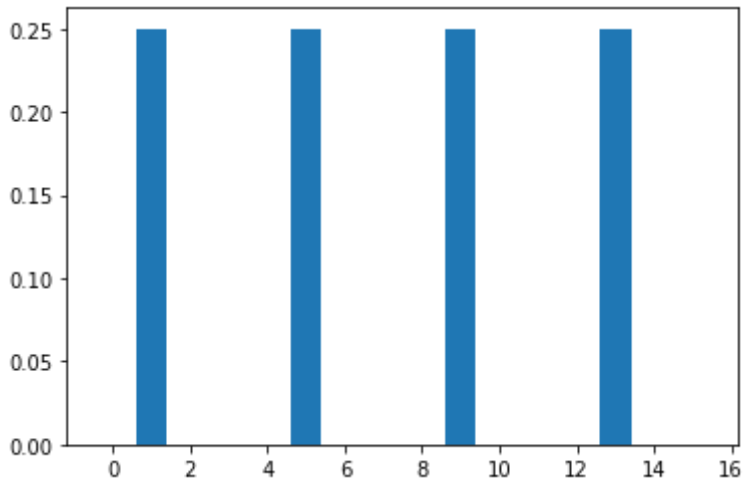
The variable `postmeasurement_state` is a vector of dimension 1024. We are interested in the state of the first register (the first 5 qubits), so let's extract that into a 32-dimensional vector `first_register`.

We then plot the squares of the amplitudes of `first_register`.

In [13]:
```
first_register = []

for i in range(16):
```

```
    index = reverse_bits(i*16 + j)
    ampl = postmeasurement_state[index]
    first_register.append(ampl)

plt.bar(range(16),[np.absolute(f)**2 for f in first_register])
```

Out[13]: <BarContainer object of 16 artists>



Does the plot agree with your theoretical calculations of the post-measurement state?

Solution

Yes, the plot agrees with the theorecical calculations of the post-measurement state!

Problem 4.7

Let $|\psi\rangle$ denote the post-measurement state corresponding to the variable `first_register`. Suppose we apply the 4-qubit Fourier Transform unitary F_{16} to $|\psi\rangle$ to get the state $\left|\hat{\psi}\right\rangle$.

First, compute by hand the Fourier transform F_{16} of $|\psi\rangle$, and determine an algebraic expression for the amplitudes of $\left|\hat{\psi}\right\rangle$.

If we then measured $\left|\hat{\psi}\right\rangle$ in the standard basis, what outcomes are we most likely to get?

Hint: Your answers to Problem 1 may be helpful.

Solution.

When we use the recursive structure for F_N taught in class and multiply the same with the state $|\psi\rangle$, we will get that:

$$F_{16}\,|\psi\rangle = \frac{1}{4}(|0\rangle + |4\rangle + |8\rangle + |12\rangle)$$

Taking some insight from Problem 1:

F_{16} is the change of basis matrix. Following from this,

Any jth column of F_{16} can be written as:

$$\frac{1}{4} * \{1, \omega_{16}^{j}, \omega_{16}^{2j}, \omega_{16}^{3j}, \ldots, \omega_{16}^{15j}\}^T$$

Since, F_{16} is symmetric, any jth row of F_{16} can be written as:

$$\frac{1}{4} * \{1, \omega_{16}^{j}, \omega_{16}^{2j}, \omega_{16}^{3j}, \ldots, \omega_{16}^{15j}\}$$

Also, since the ith entry is the dot between ith row of F_{16} and $|\psi\rangle$, we can formalize the entry as:

$$F_{16}|\psi\rangle_i = \frac{1}{4}\sum_{k=0}^{15} \omega_{16}^{ik} |\psi_k\rangle$$

Futher, if we measure the outcome of 3, then the post measurement will give us:

$$F_{16}|\psi\rangle = \frac{1}{4}\sum_{k=0}^{15} \left(\omega_{16}^{k} + \omega_{16}^{5k} + \omega_{16}^{9k} + \omega_{16}^{13k}\right)|k\rangle$$

Simplifying the same gives us:

$$F_{16}|\psi\rangle = \frac{1}{4}(|0\rangle + |4\rangle + |8\rangle + |12\rangle)$$

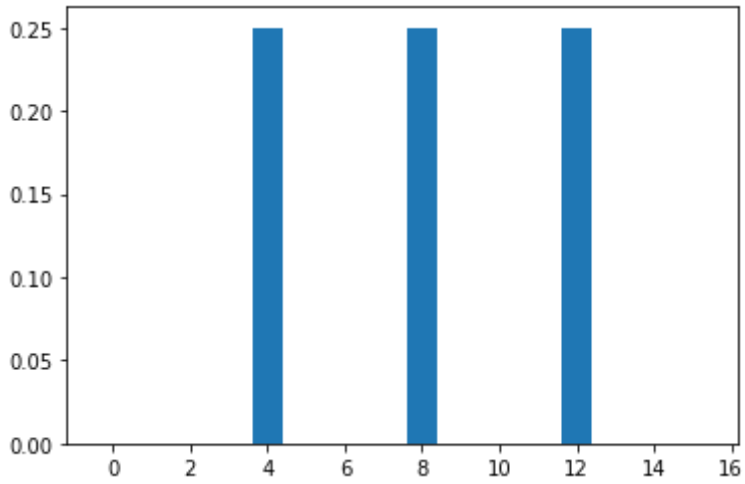
Therefore, we get the states $|0\rangle$, $|4\rangle$, $|8\rangle$, and $|12\rangle$ with equal probability of $\frac{1}{4}$.

We now plot the squares of the amplitudes of $\left|\hat{\psi}\right\rangle$ below. Does it match your math above?

Yes, It Does!

In [14]:
```
import numpy.fft as fft

mean = sum(first_register)/len(first_register)
fourier = fft.fft(np.array(first_register) - mean,16, axis = 0)
# have to add in the normalization because numpy doesn't do it
fourier = fourier/np.sqrt(16)

# plot the absolute value squared of the fourier transform
plt.bar(range(16),[np.absolute(f)**2 for f in fourier])
```

Out[14]:
```
<BarContainer object of 16 artists>
```



Problem 4.8

Suppose you had multiple copies of the state $\left|\hat{\psi}\right\rangle$. Explain how, by measuring this state in the standard basis multiple times, one can determine the period r of the function $f(x)$? This should use the algebraic expression for the amplitudes that you obtained in Problem 4.7.

Solution.

When we have multiple copies of the state $\left|\hat{\psi}\right\rangle$, we would get the outputs as 0, 4, 8 and 12 with equal probability. By measuring the qubits again and again for the copies, we would get either $|0\rangle$, $|4\rangle$, $|8\rangle$, or $|12\rangle$ with equal probability. Using this data that we have measured again and again for the copies, we can directly get the period of the function by taking the greatest common divisor of the outputs that are non-zero or we can just take any 2 non-zero outputs and take their gcd. That greatest common divisor will the period of the function f.