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

COMS 4281 Intro to Quantum Computing

Problem Set 3: Quantum Algorithms

Due: Nove m.

Collaboration aged (teams of at most 3). Please read the syllabus carefully for th collaboration. In particular, everyone must write their own solutions

Write your collaborators here:

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

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

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 $0 \leq y < 2^n$,

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

and recall the

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

Show that

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

Solution

Problem 1.4

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

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

where

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

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

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

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

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 f[illegible]e size of its range? Meaning, how many possible strings can be output by f?

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

Problem 2.3

Prove that your algorithm works.

Solution

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

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 does not need a bounded running time; you just need to show that the *expected* number of queries before the algorithm finds the minimum is $O(\sqrt{N})$.

Solution

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

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 [ ]:
```
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 apper
    '''
    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,
                          ntum_registers: List[int]) -> QuantumCircuit:
    '''
    Takes ar                    d appends a diffusion gate to the registers.
    The quar                    t be length 5.
    '''
    if len(q                    != 5:
        retu
    for i in                    s:
        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 [ ]:

```
### ======== BEGIN CODE ==========
'''
Below are some suggested templates for functions that would be useful to implem
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 gues
    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)

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

    return y
```


Test your code on some examples.

In [ ]:
```
f1 = lambda
print("Minim                ind_min(f1))
print("Oracl                num_oracle_calls)
```

In [ ]:
```
f2 = lambda                  3
print("Minimum of f2 is: ", find_min(f2))
print("Oracle calls used: ", num_oracle_calls)
```

In [ ]:
```
f3 = lambda x: 1+0 - np.sin(x / 16)
print("Minimum of f3 is: ", find_min(f3))
print("Oracle calls used: ", num_oracle_calls)
```


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

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 [ ]:
```
order_findir                    umCircuit(8,4)

### ========                      ========

### ========                      ======

order_findir                    tput="mpl")
```


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:

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

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

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

In [ ]:
```
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 c
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_
    return UnitaryGate(unitary,"U_f")

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

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

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


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

Problem

Let's simulate the second register and collect measurement statistics. Write code to probability of each measurement outcome (for example, the integer 1 a me, 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 [ ]:
```
order_finding_circuit.measure([4,5,6,7], [0,1,2,3])
order_finding_circuit.draw(output="mpl")
```


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


In [ ]:
```
### ======== BEGIN CODE =============


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


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

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 [ ]:
```
j = 3
```

```
j_in_rev_binary = '1100' # Note this bitstring is in reverse order. Reversing i

#just repeating this 100 times until you get the result you want (the number 1
for k in range(100):
    result = execute(order_finding_circuit, backend=backend, shots=1).result()

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

### ====== E                ===============

postmeasurem                 t.get_statevector()
```


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

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

In [ ]:
```
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])
```


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

Solution

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.

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

In [ ]:
```
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 a                ared of the fourier transform
plt.bar(rang                e(f)**2 for f in fourier])
```


Problem

Suppose you h f 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.

In [ ]: