Problem Set 2 - Coding

Due: December 14, 2025

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.

Write your collaborators here:

Recommended Environment to Run This Notebook

We highly recommend that you use the qBraid platform to run this Jupyter notebook. This supports Qiskit, and furthermore to render your Problem Set solutions to PDF, you have to do the following:

1. File > Save and Export Notebook As > HTML
2. Save the HTML file somewhere on your local computer
3. Open the HTML file using your favorite browser, and Print to PDF. We recommend using Landscape mode so the Python code shows up better.

In [ ]:
```
!pip install qbraid > /dev/null
!pip install 'qbraid[braket]' > /dev/null
!pip install qbraid-cli > /dev/null
!pip install 'qbraid-cli[envs]' > /dev/null
!pip install qiskit > /dev/null
!pip install qiskit_aer > /dev/null
!pip install numpy > /dev/null
!pip install matplotlib > /dev/null
```

In [ ]:
```
import qbraid
from qbraid.runtime import QbraidProvider, BraketProvider, load_job

from qiskit import QuantumCircuit, transpile
```

```
from qiskit.circuit.library.standard_gates import XGate, YGate, ZGate, HGate
from qiskit.visualization import plot_histogram

from qiskit_aer import AerSimulator
from qiskit_aer.noise import depolarizing_error

import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import os
```

Problem 0

CHSH game on a quantum computer

Note: *this is just a warmup to test that you can run jobs on actual quantum hardware; you don't have to write any code for this problem*.

In this demo we implement the CHSH nonlocal game on a quantum device and compare its performance to the best possible classical strategy. The game involves two parties, Alice and Bob, who receive random input bits $x, y \in \{0, 1\}$ and must output bits $a, b \in \{0, 1\}$ without communicating. They "win" a round if

$$a \oplus b = x \wedge y.$$

Classically, no matter what strategy Alice and Bob agree on in advance, their average success probability is at most $3/4$. Using an entangled Bell state and appropriate measurement bases, a quantum strategy can reach an ideal success probability of $\cos^2(\pi/8) \approx 0.854$, strictly higher than the classical bound.

By running this game on hardware and plotting the observed success rates for each input pair (x, y), as well as the overall average success, we can:

- Directly visualize the quantum violation of the classical (local hidden-variable) limit.
- See how close real devices come to the ideal quantum prediction, and how noise and decoherence reduce the observed advantage.

Set up

Below this part be careful when running cells since some jobs may be submitted to actual quantum computers which will consume your qBraid credits. To set up your connections run the following commands.

In [ ]:
```
%load_ext qbraid_magic
```

In [ ]:
```
!qbraid jobs enable braket -y
!qbraid jobs state
```

In [ ]:
```
provider = QbraidProvider(api_key='YOUR API KEY HERE') # Found at https://account.qbraid.com/ > account > Y
provider.save_config()
provider.get_devices()
```

Below the two devices selected are IQM's Emerald and Rigetti's Ankaa 3, you can use these if you want or from the above list you may select any two available devices to benchmark on. Note that different devices cost different amounts and that you have a limited number of credits. **It's your resposibility to manage the amount of credits you consume.**

In [ ]:
```
iqm_device = provider.get_device('iqm_emerald')
rigetti_device = provider.get_device('rigetti_ankaa_3')
```

In [ ]:
```
def chsh_circuit(x, y):
    """
    Build a 2-qubit circuit for the CHSH game:

      - Prepare |Φ+> = (|00> + |11>)/sqrt(2).
      - Alice (qubit 0) uses setting A_x:
          x = 0: Z basis
          x = 1: X basis (H then Z)
      - Bob (qubit 1) uses setting B_y:
          y = 0: (Z + X)/√2  (angle +π/4 from Z)
          y = 1: (Z − X)/√2  (angle −π/4 from Z)

    All measurements are carried out in the Z basis after rotations.
    """

    qc = QuantumCircuit(2, 2)
```

```
    # Prepare Bell state
    qc.h(0)
    qc.cx(0, 1)

    # Alice's measurement basis
    if x == 1:
        qc.h(0)  # X basis

    # Bob's measurement basis via rotation into Z basis
    if y == 0:
        qc.ry(-np.pi / 4, 1)  # (Z + X)/root(2)
    else:
        qc.ry(+np.pi / 4, 1)  # (Z - X)/root(2)

    # Measure:
    # classical bit 1 <- Alice (qubit 0)
    # classical bit 0 <- Bob   (qubit 1)
    qc.measure(0, 1)
    qc.measure(1, 0)

    return qc


def chsh_success_for_setting(counts, x, y):
    """
    Given measurement counts for a fixed input setting (x, y),
    compute the CHSH game success probability.

    CHSH game:
      - Inputs: x, y in {0, 1}.
      - Outputs: a, b in {0, 1} from measurement outcomes.
      - Winning condition: a xor b = x and y.

    Here counts has keys 'ab' where:
      a = Alice's bit (classical bit 1, leftmost char),
      b = Bob's bit   (classical bit 0, rightmost char).
    """
    shots = sum(counts.values())
    wins = 0

    for bitstring, c in counts.items():
        # bitstring is 'ab':
```

```
        a_bit = int(bitstring[0])  # Alice
        b_bit = int(bitstring[1])  # Bob

        # Win if a ⊕ b = x ∧ y
        if (a_bit ^ b_bit) == (x & y):
            wins += c

    return wins / shots


def plot_chsh_success(success, avg_success):
    """
    Plot CHSH success probabilities for each setting and compare
    to the optimal classical strategy (3/4) and ideal quantum value.
    """
    settings = [(0, 0), (0, 1), (1, 0), (1, 1)]
    labels = [f"(x={x}, y={y})" for (x, y) in settings]
    values = [success[(x, y)] for (x, y) in settings]

    classical_bound = 0.75                      # optimal classical average success
    ideal_quantum = np.cos(np.pi / 8) ** 2      # = 0.854, ideal CHSH quantum strategy

    fig, ax = plt.subplots()
    ax.bar(labels, values)
    ax.axhline(classical_bound, linestyle="--", linewidth=1,
               label="Classical bound (3/4)")
    ax.axhline(ideal_quantum, linestyle=":", linewidth=1,
               label="Ideal quantum (~0.854)")

    ax.set_ylim(0.5, 1.0)
    ax.set_ylabel("Success probability")
    ax.set_title(f"CHSH game success; average quantum = {avg_success:.3f}")
    ax.legend()
    plt.tight_layout()
    plt.show()



shots = 500
settings = [(0, 0), (0, 1), (1, 0), (1, 1)]

circuits = [chsh_circuit(x, y) for (x, y) in settings]
```

```
jobs = rigetti_device.run(circuits, shots=shots)
```

The above code will likely run quickly. It only *creates* the job. The jobs are probably still running by the time the cell is finished.

If you re-run the above cell, it will create more jobs and spend more credits.

The below code will likely execute slowly. It waits for the jobs to finish. You may re-run this without spending more credits.

In [ ]:
```
results = [job.result() for job in jobs]
print(results)

# Compute per-setting success probabilities
success = {}
for idx, (x, y) in enumerate(settings):
    print(type(results[0]))
    counts = results[idx].data.get_counts()
    success[(x, y)] = chsh_success_for_setting(counts, x, y)

# Average success assuming inputs (x, y) are chosen uniformly at random
avg_success = sum(success.values()) / 4.0

print("Per-setting CHSH success probabilities:")
for (x, y), p in success.items():
    print(f"  P_win(x={x}, y={y}) = {p:.3f}")
print(f"Average quantum success = {avg_success:.3f}")
print("Classical optimal average success = 0.750")

plot_chsh_success(success, avg_success)
```

Problem 2

Benchmarking Single Qubits

In this subproblem, we will benchmark individual qubits on the quantum devices you just selected. A typical way to benchmark a qubit is to run a sequence of randomly chosen single-qubit gates $g_1, g_2, \ldots, g_k$, and then running the reverse sequence of $g_k^{-1}, \ldots, g_1^{-1}$ so that overall the effect should be the identity. Of course, each gate will incur some noise, so the

state of the qubit will drift over time. One can measure the noise by measuring the qubit at the end of the sequence to see if it stayed in the |0⟩ state.

You will write code to perform the following: for $k = 5, 10, 15, 20, 25$, for each qubit $q = 0, 1, 2, \ldots, 4$, pick a sequence of gates $g_1, \ldots, g_k$ where each g_i is chosen randomly from the gate set X, Y, Z, H. Then, on qubit q run the sequence (g_i) forward and then in reverse, and measure the qubit for a total of 10, 20,..., 50 gates being applied. Do this 1000 times and calculate the percentage p_k of times that the qubit ends back in the |0⟩ state.

In [ ]:
```
'''
This function takes as input
    - k : half-length of random gate sequence

  Returns:
    - the QuantumCircuit object corresponding to the circuit
'''
def benchmark_qubit(k):
    qc = QuantumCircuit(1, 1)
    ########## WRITE CODE TO GENERATE THE RANDOM SEQUENCE OF LENGTH k, and ITS REVERSAL, ON QUBIT q #######



    ############ END CODE BLOCK ############
    # measure qubit q, and store it in classical register [0]
    qc.measure_all()
    return qc
```

In [ ]:
```
qc_1 = benchmark_qubit(5) # k = 5
qc_2 = benchmark_qubit(10) # k = 10
qc_3 = benchmark_qubit(15) # k = 15
qc_4 = benchmark_qubit(20) # k = 20
qc_5 = benchmark_qubit(25) # k = 25
qc_batch = [qc_1, qc_2, qc_3, qc_4, qc_5]

qc_1.draw() # this will draw you circuit so you can verify visually
```

The next two cells submit jobs to the quantum computers

In [ ]:
```
job_iqm = iqm_device.run(qc_batch, shots=1000)
```

In [ ]:
```
job_rigetti = rigetti_device.run(qc_batch, shots=1000)
```

This next part is a little laborsome. While the jobs all submitted in a single line, qBraid breaks the batch into separate job ids. From the qBraid website go to Quantum Jobs and find your jobs. They'll likely still be queued at this point but you can copy paste the job ids (labelled as 'taskArn') from each job into the arrays below.

In [ ]:
```
# Example job id: arn:aws:braket:eu-north-1:592242689881:quantum-task/015a3437-07a4-43ff-b60a-00f2bc0be29f

job_ids_iqm = ["YOUR_JOB_ID_1", "YOUR_JOB_ID_2", ...] # Put jobs that ran on IQMs machine here
job_ids_reg = ["YOUR_JOB_ID_1", "YOUR_JOB_ID_2", ...] # Put jobs that ran on Rigettis machine here
```

Once all jobs have status COMPLETED indicated by a green checkmark on the website run the cell below which will load each jobs results into their respective result arrays

In [ ]:
```
# Run this cell to load each jobs data
results_iqm = []
results_reg = []

for i in range(len(job_ids_iqm)):
    temp = load_job(job_ids_iqm[i], "braket")
    results_iqm.append(temp.result())

for i in range(len(job_ids_reg)):
    temp = load_job(job_ids_reg[i], "braket")
    results_reg.append(temp.result())
```

In [ ]:
```
# This will plot a histogram of the data from rigetti
plot_histogram([result.data.get_counts() for result in results_reg])
```

Next using plot the results on a graph comparing IQM and Rigetti's (or whichever two you picked if you changed them) quantum computers with the y-axis as the percentage of times the qubit returned correctly and the x-axis as k.

Find a best function $f(k)$ (linear, quadratic, exponential,...) that fits the plots. For example, if the plot of p_k against k looks linear, then you should come up with parameters a, b such that p_k is close to $ak + b$. If it looks like exponential decay, then you should find an approximate $f(k) = ae^{bk} + c$ for some parameters a, b, c.

This function can be used to give a simple model for how noise accumulates on each qubit from single-qubit gates.

Problem 3

Reliable quantum memory with Shor's code

Consider a situation where we have a reliable quantum computer that can *operate* on qubits with negligible error, but cannot *store* qubits for long periods of time. So we have another system called a quantum *memory* that can store qubits with some small error rate ϵ. Let's see how we can use Shor's code to increase the reliability of our storage mechanism. We're going to compare the reliability of storing a single qubit in memory versus storing 9 error-corrected qubits.

Our assumption is that no errors will occur during computation. Only the storage/retrieval process includes an error channel.

Your objectives:

- Fill in the correction procedure for Shor's code. Note that we're not using any ancillas! Try to come up with a way to implement the majority vote corrections using only CNOT, Toffoli, and H gates between the 9 data qubits. (Hint: look up "Quantum error correction" on Wikipedia.)
- Assume that the probability of an error having occurred during storage is exponentially distributed as follows where t is in seconds:

  $$p(\text{no error}) = e^{-t}$$

  If we want to store a logical qubit for one hour with a 99% chance of success, how often must we run Shor's correction procedure? Find the largest T such that running Shor's correction procedure every T seconds results in a final success probability over 99%.

Note that the quantity "total variation distance" (TVD) corresponds to the probability of a logical error in the whole circuit. This is given by the `calculate_total_error_prob` function.

In [ ]:
```
def build_physical_circuit(error_prob: float = 0.0) -> QuantumCircuit:
    qc = QuantumCircuit(1, 1)
    qc.z(0)
    if error_prob > 0:
        error = depolarizing_error(error_prob, 1)
        qc.append(error, [0])
    qc.measure(0, 0)
```

```
    return qc


def build_shor_circuit(error_prob: float = 0.0) -> QuantumCircuit:
    qc = QuantumCircuit(9, 1)

    # Encode
    qc.cx(0, 3)
    qc.cx(0, 6)
    for q in [0, 3, 6]:
        qc.h(q)
    blocks = [(0, 1, 2), (3, 4, 5), (6, 7, 8)]
    for ctrl, t1, t2 in blocks:
        qc.cx(ctrl, t1)
        qc.cx(ctrl, t2)

    qc.barrier()

    # Logical Z (Transversal X)
    for q in range(9):
        qc.x(q)

    qc.barrier()

    # Noisy Storage
    if error_prob > 0:
        error = depolarizing_error(error_prob, 1)
        for q in range(9):
            qc.append(error, [q])

    qc.barrier()

    # Error Correction
    ################## YOUR CODE GOES HERE
    for ctrl, t1, t2 in blocks:
        qc.cx(ctrl, t1)
        qc.cx(ctrl, t2)
        qc.ccx(t1, t2, ctrl)

    for q in [0, 3, 6]:
        qc.h(q)
```

```
    qc.cx(0, 3)
    qc.cx(0, 6)
    qc.ccx(3, 6, 0)
    ####################################

    qc.barrier()

    # Decode/Measure
    qc.measure(0, 0)
    return qc


def calculate_total_error_prob(counts: dict[str, int], shots: int, ideal_outcome: str) -> float:
    ideal_count = counts.get(ideal_outcome, 0)
    p_accuracy = ideal_count / shots
    return 1.0 - p_accuracy
```

In [ ]:

```
backend = AerSimulator()

# Verify ideal
qc_check = build_shor_circuit(error_prob=0.0)
res_check = backend.run(qc_check, shots=1000).result()
counts_check = res_check.get_counts()
print(f"Ideal check (expect {{'0': 1000}}): {counts_check}")

shots = 10000
p_values = np.geomspace(0.01, 0.3, 30)
ideal_outcome = "0"

phys_tvds = []
shor_tvds = []

print(f"Running simulation with {shots} shots per point...")

for p in p_values:
    # Physical
    qc_phys = build_physical_circuit(error_prob=p)
    res_phys = backend.run(qc_phys, shots=shots).result()
    phys_tvds.append(calculate_total_error_prob(res_phys.get_counts(), shots, ideal_outcome))

    # Shor
    qc_shor = build_shor_circuit(error_prob=p)
```

```
    res_shor = backend.run(qc_shor, shots=shots).result()
    shor_tvds.append(calculate_total_error_prob(res_shor.get_counts(), shots, ideal_outcome))

plt.figure(figsize=(10, 6))
plt.loglog(p_values, phys_tvds, "o--", label="Physical Qubit")
plt.loglog(p_values, shor_tvds, "s-", label="Shor Code")
plt.xlabel("Storage Error Probability (p)")
plt.ylabel("Total Variation Distance (TVD)")
plt.title("Memory Stability: Single Qubit vs Shor Code")
plt.legend()
plt.grid(True, which="both", ls="-", alpha=0.5)
plt.show()
```

Optionally, you may compare these error probabilities to those of real hardware. Uncomment and run the below code to see how often the encoding/decoding cycle for Shor's 9-bit code executes successfully. In the above simulations, we assumed this process was error-free. How good is that assumption today?

In [ ]:
```
# ----------------------------------------------------------------------
# Uncomment this block to run the ideal Shor code on real hardware
# backend "iqm_device" and compute the TVD vs the ideal code.
#
# Note that this must be run after having defined an iqm_device
# backend in the previous problem.
#
# hw_shots = 10000
# qc_shor_ideal = build_shor_circuit(error_prob=0.0)
# qc_shor_ideal_t = transpile(qc_shor_ideal, iqm_device)
# job_hw = iqm_device.run(qc_shor_ideal_t, shots=hw_shots)
# res_hw = job_hw.result()
# hw_tvd = calculate_tvd(res_hw.get_counts(), hw_shots, ideal_outcome)
# print(f"iqm_device TVD vs ideal code: {hw_tvd:.2f}")
# ----------------------------------------------------------------------
```

In what regime is error correction preferable? Provide intution for the existence of a regime where error correction worsens performance.

How often should we run error correction?

Given a target error rate (measured by TVD in this case), how do we know how often we should run error correction? We first

have to make some assumption about the underlying error process. For this problem, assume that the probability $p_E(t)$ of error after t seconds during the storage process is exponentially distributed. That is,

$$p_E(t) = 1 - \exp\left(\frac{-t}{\tau}\right)$$

Start by calculating the slope and intercept of the line of best fit comparing `log_p` and `log_tvd`. This will be helpful in the next part.

In [ ]:
```
# Fit error curve with a power law.
# We'll do this by fitting a line on a log-log scale.

################## YOUR CODE GOES HERE
shor_tvds_arr = np.array(shor_tvds, dtype=float)

# Guard against any zero TVD values (cannot take log of 0)
mask = shor_tvds_arr > 0
log_p = np.log10(p_values[mask])
log_tvd = np.log10(shor_tvds_arr[mask])

# Linear regression on log10(TVD) vs log10(p)
slope, intercept = np.polyfit(log_p, log_tvd, 1)

print("Log-log linear fit for Shor code:")
print(f"log10(TVD) = {slope:.3f} * log10(p) + {intercept:.3f}")
print(f"TVD = 10^{intercept:.3f} * p^{slope:.3f}")
################## YOUR CODE ABOVE
```

Now, incorporating both the exponential error model and the specific values we just found for estimated TVD given a storage error probability, we will calculate how frequently we must perform error correction to stay under 1% TVD from the ideal distribution.

In [ ]:
```
def shor_logical_error_prob(p_phys: float) -> float:
    """
    Approximate per-period logical failure probability for the Shor code
    given per-qubit physical error probability p_phys, using the fitted
    power-law model TVD(p) ≈ 10**intercept * p**slope.
    """
    if p_phys <= 0:
```

```
        return 0.0

    log_tvd = slope * np.log10(p_phys) + intercept
    tvd = 10.0 ** log_tvd
    # Clip to [0, 1] for numerical stability
    return float(np.clip(tvd, 0.0, 1.0))


def success_probability_for_period(
    T: float,
    tau: float,
    T_total: float = 3600.0
) -> float:
    """
    Overall success probability for storing a logical qubit for T_total seconds,
    when Shor error correction is run every T seconds, given mean physical
    error time tau (seconds), under an exponential error model.
    """
    ###################### YOUR CODE GOES HERE
    if T <= 0 or tau <= 0:
        return 0.0

    # Physical error probability over interval T under exponential model
    p_phys = 1.0 - np.exp(-T / tau)

    # Per-period logical failure probability
    p_logical_fail = shor_logical_error_prob(p_phys)

    if p_logical_fail >= 1.0:
        return 0.0

    # Number of QEC periods in T_total
    n_periods = T_total / T

    return (1.0 - p_logical_fail) ** n_periods
    ###################### YOUR CODE ABOVE


def find_max_correction_interval(
    tau: float,
    T_total: float = 3600.0,
    target_success: float = 0.99
```

```
) -> tuple:
    """
    For a given tau (mean time between physical errors, in seconds), find the
    largest correction interval T such that the overall success probability
    over T_total seconds is at least target_success.

    T is scanned on a log grid: T ∈ [1e-4, 1e4] seconds, then restricted to
    T <= T_total.

    Returns (T_max, P_success(T_max)). If no such T exists, returns (None, None).
    """
    T_candidates = np.logspace(-6, 4, 1000)
    T_candidates = T_candidates[T_candidates <= T_total]
    if T_candidates.size == 0:
        return None, None

    success_vals = np.array([
        success_probability_for_period(T, tau, T_total)
        for T in T_candidates
    ])

    mask_ok = success_vals >= target_success

    if not np.any(mask_ok):
        return None, None

    # Largest T that still meets the success criterion
    T_max = T_candidates[mask_ok].max()
    P_succ_max = success_probability_for_period(T_max, tau, T_total)
    return float(T_max), float(P_succ_max)
```

Run the plotting code below to see if your code works and gives reasonable results.

In [ ]:
```
T_total = 3600.0        # one hour
target_success = 0.99   # ≥ 99% success over T_total

# Choose a range of tau values for the hardware (in seconds)
taus = np.logspace(-1, 3, 10)

T_max_values = []
```

```
for tau in taus:
    T_max, P_succ = find_max_correction_interval(
        tau,
        T_total=T_total,
        target_success=target_success,
    )
    # Use NaN if no feasible T exists for this tau
    if T_max is None:
        T_max_values.append(np.nan)
    else:
        T_max_values.append(T_max)

T_max_values = np.array(T_max_values)

plt.figure(figsize=(8, 5))
plt.loglog(taus, T_max_values, "o-")
plt.xlabel(r"Mean time between physical errors $\tau$ (s)")
plt.ylabel(r"Max correction interval $T_{\max}$ (s)")
plt.title(r"$T_{\max}$ vs. $\tau$ for 1 hour storage at $\geq 99\%$ success")
plt.grid(True, which="both", ls="-", alpha=0.5)
plt.show()
```

If we can only run error correction about every 1 ms, how good do the underlying qubits for storage need to be? That is, what is the required mean time between physical errors?

Problem 4

In this question, you will use implement Grover's algorithm and run it on noisy hardware.

Grover's oracle

Build a grover's oracle by filling in the following method to try to find the "solution" by searching bit strings of length 3 (i.e. the set [000, 111]) to try to find the string "101". Note that we chose a string that is symmetric so that we may disregard big endianess and little endianess representations. Use an MCX gate (read more here if needed: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.MCXGate) targeted on the ancilla qubit with control on the "input" qubits or any series of one or two qubit gates. Make sure to reverse any bit-flips or computations so that the "input" qubits are reset appropriately.

Alternatively, you can use the PhaseOracleGate and input a SAT formula whose solution is "101" (https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.circuit.library.PhaseOracleGate), or you can directly implement a SAT oracle.

In [ ]:
```
def grover_unitary() -> QuantumCircuit():
    qc = QuantumCircuit(3+1)
    ###############################################################################
    # your code here
    # qc.mcx([list of control bits], target bit) # this an example of how to call mcx

    ###############################################################################
    return qc
```

Grover's Diffuser

Build Grover's diffuser by rotating the input qubits out of the hadamard basis bit flipping them and then apply a MCP(multi-control phase) gate with angle π so that it functions as a MCZ gate with target on the last qubit in the "input" qubits. Reset everything as appropriate. Note, this does not depend on the oracle and is the same for every Grover's subroutine.

In [ ]:
```
def grover_diffuser() -> QuantumCircuit():
    qc = QuantumCircuit(3)
    ###############################################################################
    # your code here

    # 1. apply an H gate to each qubit

    # 2. apply an X gate to each qubit

    # 3. apply the MCZ gate
    #qc.mcp(np.pi, control_qubits, target_qubit) #applies MCZ

    # 4. undo the X and H gates you applied

    ###############################################################################
    return qc
```

Grover's Algorithm

Calculate the number of iterations of Grover's Algorithm needed for this problem using k iterations calculated by the following

equation $k = \frac{\pi}{2\theta} - \frac{1}{2}$ where $\theta = 2sin^{-1}\sqrt{\frac{m}{N}}$. Then run a circuit for k iterations where k denotes the optimal number of iterations. Plot the histograms of the measurement and compare them with the solution.

In [ ]:
```
grover_of = grover_unitary().to_gate()
grover_of.name = "Oracle"
grover_ud = grover_diffuser().to_gate()
grover_ud.name = "Diffuser"

qc_g = QuantumCircuit(3+1,3)
################################################################################
# your code here

# 1. initialize by setting each input qubit to |+> and the ancilla to |->

# 2. iterate the correct amount of times with the oracle and diffuser
# example of how to append the circuits you made earlier
# qc_g.append(grover_of, range(3+1))
# qc_g.append(grover_ud, range(3))

################################################################################
for i in range(3):
    qc_g.measure(i, i)

qc_g.draw()
```

Run on simulator

Run the Grover's circuit we just constructed on a simulator. How does this compare with the ideal predicted success probability ($\sin^2((2k+1)\theta)$)?

In [ ]:
```
from qiskit import transpile
from qiskit_aer import AerSimulator

shots = 1024
backend_sim = AerSimulator()

# helper method for simulator runs
def fetch_counts(qc:QuantumCircuit(), backend, shots):
```

```
    compiled_qc = transpile(qc, backend, optimization_level=0)
    job_sim = backend.run(compiled_qc, shots=shots)
    results = job_sim.result()
    return results.get_counts()

plot_histogram(fetch_counts(qc_g, backend_sim, shots))
```

Predict performance of Grover's on noisy hardwares given single-qubit benchmarking from Problem 2.

For simplicity, assume that the multi-qubit gates have independent single qubit gate errors, which we estimated in Problem 2 (for example, a MCX gate applied on 4 qubits is counted as 4 independent single qubit errors). Note this is a huge simplification and actual error for multi-qubit gates would be higher.

Predict the success probability of Grover's algorithm for the two types of hardwares given your estimate for the single qubit error rates from Problem 2.

Note: if you got an unreasonable value for the estimated error in Problem 2, you can also use the stated single qubit error rate from the hardware specs (99.75% single qubit gate fidelity for Rigetti)

Now, run Grover algorithm on actual hardware. How do the results compare with your prediction?

In [ ]:
```
n_shots = 1000
job_iqm = iqm_device.run([qc_g], shots=n_shots)
job_rigetti = rigetti_device.run([qc_g], shots=n_shots)
```

In [ ]:
```
job_id_iqm =  # same as in problem 2, copy this from your qbraid quantum jobs dashboard
job_id_rig =
```

In [ ]:
```
# Run this cell to load each jobs data
temp = load_job(job_id_iqm, "braket")
result_iqm = temp.result()

temp = load_job(job_id_rig, "braket")
result_rig = temp.result()
```

In [ ]:
```
# This will plot a histogram of the data from rigetti
plot_histogram(results_rig.data.get_counts())
```

In [ ]:
```
import numpy as np

def success_prob(res):
    counts = res.data.get_counts()
    shots = sum(counts.values())
    p = counts.get('101', 0) / shots
    return p

p_iqm = success_prob(result_iqm)
p_rig = success_prob(result_rig)
```

Problem 5

After you're all done, please remember to fill out the

1. QBraid survey (https://docs.google.com/forms/d/e/1FAIpQLSdrferJftIdWIHN32D9892gjlmQ8zwd9JGDqxKbKHn5WR5POQ/viewform). This will be extremely helpful information for QBraid, who generously provided the compute credits that allowed you to complete this assignment.

2. Course evaluations. This will be extremely helpful feedback for the instructor and TAs!

In [ ]: