Assignment 2

***Due**: Monday, Sept 30th, 2024 @ 11:59PM*

CPSC 447/547 Introduction to Quantum Computing (Fall 2024)

1 Introduction

Welcome to Assignment 2 for CPSC 447/547 (Introduction to Quantum Computing). As usual, collaboration is encouraged; if you discussed with anyone besides the course staff about the assignment, *please list their names* in your submission.

Getting Started.

This assignment has two parts, a *written portion* and a *programming portion*. The tasks that are marked by "($\star$ pts)" are optional. Typesetting your solutions to the written portion is not mandatory but highly encouraged. See the instructor's note on Ed for details about Latex for quantum computing. Some basic familiarity with Python and object-oriented programming is required to complete the programming portion of this assignment. No Python packages, except for `math` and `numpy`, are allowed. To start,

- Create a folder for Assignment 2, e.g., `A2/`
- Download the starter files for this assignment to that folder from the course website:
  - `written.tex`
  - `A2.py`
  - `requirement_A2.py` (Do not modify)
- Write your solutions in `A2.py` (for programming tasks) and in `written.tex` or on paper (for written tasks).
- Debug and test your solution locally by running '`python3 A2.py`' on command line. This will check for any violation of the requirements and run correctness tests. Feel free to add more tests in `A2.py`. Do not hardcode your solutions for each public test cases.

Submission. Once you have completed and are ready to submit, upload two files to Gradescope (accessed through Canvas): `written.pdf` and `A2.py`. Gradescope will immediately show the results from running the requirement test and public test cases. If your file fails the tests in `requirement_A2.py`, a **0** score will be assigned.

After the deadline, your written solution will be graded manually by our course staff; your programming solution will be graded using our auto-grading script that contains private test cases. Late submissions (for up to two days) will receive a 50% penalty.

WRITTEN PORTION
This portion of the assignment has a total of 49 *points.*

2 Measurements

Recall from lecture, a quantum measurement, represented by an observable O, will probablistically collapse a quantum state to one of the eigen-basis states of O. Specifically, by the spectral theorem, we can write $O = \sum_i \lambda_i |e_i\rangle\langle e_i|$ and associate the measurement event with a random variable x, where $x = \lambda_i$ if the measurement outcome is its associated eigen-basis $|e_i\rangle$.

Task 2.1 (9 pts)

In this task, we perform measurements by the observable $\sigma_x = (+1)|+\rangle\langle+| + (-1)|-\rangle\langle-|$, where $|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$. For each of the following quantum states, first write it in the form of $|x\rangle = \alpha|+\rangle + \beta|-\rangle$, for some complex numbers α, β. Or if $|x\rangle$ is a two-qubit state, then we want to measure the first qubit, so write the state in the form of $|x\rangle = \alpha|+\rangle \otimes |\psi_+\rangle + \beta|-\rangle \otimes |\psi_-\rangle$, for some single-qubit states $|\psi_+\rangle, |\psi_-\rangle$. Then answer what is the probability that the measurement outcome is $|+\rangle$?

(a) $|x\rangle = |0\rangle$

(b) $|x\rangle = \frac{1+i}{2}|0\rangle + \frac{1-i}{2}|1\rangle$

(c) $|x\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$

Task 2.2 (12 pts)

Suppose we are given two qubits, A and B, in the following state: $|\psi_{AB}\rangle = \frac{1}{\sqrt{3}}|00\rangle + \frac{1}{\sqrt{3}}|01\rangle + \frac{1}{\sqrt{3}}|10\rangle$. For each of the following measurement scenarios (a-c), calculate the expectation value when measuring $|\psi_{AB}\rangle$.

(a) Only measure qubit A in the x basis, that is, with observable $O = \sigma_x \otimes I$.

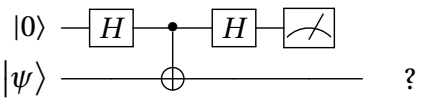
(b) Only measure qubit B in the x basis, that is, with observable $O = I \otimes \sigma_x$.

(c) Measure both qubits *jointly* in the $x \otimes x$ basis, that is, with observable $O = \sigma_x \otimes \sigma_x$.

(d) Following part (a), if the measurement outcome is $|+\rangle$, what is the state of qubit B after the measurement?

Task 2.3 ($\star$ pts)

Suppose we are given a qubit and a promise that the state of the qubit $|\psi\rangle$ is either $|+\rangle$ or $|-\rangle$. In this question, we will learn how to figure out which state the qubit is in without directly measuring $|\psi\rangle$. The key here is to use an ancillary qubit and two-qubit gates. Let's take a look at the following quantum circuit:



(a) Suppose $|\psi\rangle = |+\rangle$, then what is the probability that the top qubit measures to be $|0\rangle$? If non-zero, what is the state of the bottom qubit, given this measurement outcome?

(b) Suppose $|\psi\rangle = |+\rangle$, then what is the probability that the top qubit measures to be $|1\rangle$? If non-zero, what is the state of the bottom qubit, given this measurement outcome?

(c) Suppose $|\psi\rangle = |-\rangle$, then what is the probability that the top qubit measures to be $|0\rangle$? If non-zero, what is the state of the bottom qubit, given this measurement outcome?

(d) Suppose $|\psi\rangle = |-\rangle$, then what is the probability that the top qubit measures to be $|1\rangle$? If non-zero, what is the state of the bottom qubit, given this measurement outcome?

(e) Suppose $|\psi\rangle = \sqrt{1-\epsilon}|+\rangle + \sqrt{\epsilon}|-\rangle$ for some small ϵ, i.e., a slight perturbation from $|+\rangle$. Then what is the probability of measuring $|0\rangle$ for the top qubit?

3 Sharing Entanglement

Entanglement is one of the most powerful yet elusive properties in quantum computing. In lectures, we have studied entanglement between two parties, such as Alice and Bob. We saw how to use entanglement to accomplish something like quantum teleportation. The concept of entanglement can be generalized to three parties or more. In this question, we will explore what does it mean for more qubits to be entangled.

Task 3.1 (6 pts)

Suppose Alice, Bob, and Carol each holds on to one of the three qubits in the quantum state $|\psi_{ABC}\rangle$, respectively. For each of the following scenarios of $|\psi_{ABC}\rangle$, first answer whether or not $|\psi_{ABC}\rangle$ is entangled (i.e., is it not possible to write $|\psi_{ABC}\rangle = |\psi_A\rangle \otimes |\psi_B\rangle \otimes |\psi_C\rangle$), then write down what is the joint state of Alice and Bob's qubits after Carol measured her qubit (conditioned on her measurement outcome). Finally, does Carol's measurement breaks the entanglement between Alice and Bob? Answer whether Alice's qubit and Bob's qubit are still entangled, if not, write down the product form $|\psi_A\rangle \otimes |\psi_B\rangle$.

(a) "GHZ state": $|\psi_{ABC}\rangle = \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|111\rangle$

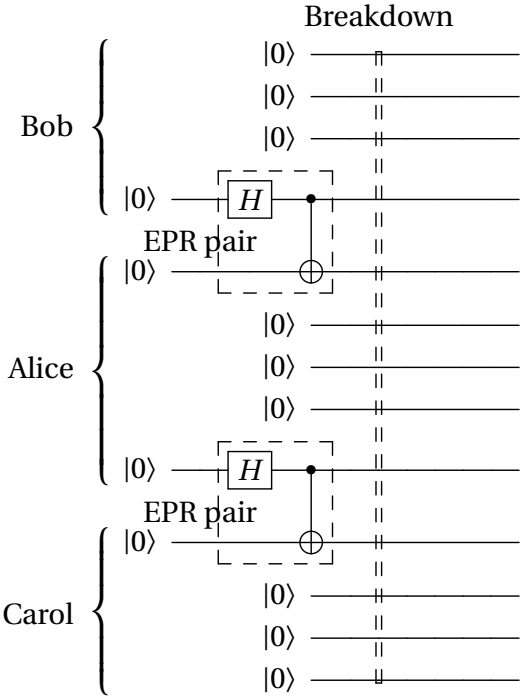
(b) "W state": $|\psi_{ABC}\rangle = \frac{1}{\sqrt{3}}|001\rangle + \frac{1}{\sqrt{3}}|010\rangle + \frac{1}{\sqrt{3}}|100\rangle$

Task 3.2 (10 pts)

Suppose Alice lives in Chicago, Bob lives in Los Angeles, and Carol lives in New Haven. They can communicate by classical channels (such as sending classical bits via phone calls or radio signals) as well as quantum channels (such as sending qubits implemented by photons via optical fibers). One day, all quantum channels broke down... So they have to resort to using quantum teleportation for communicating qubits. Fortunately, right before the breakdown, Alice prepared two EPR pairs; she sent one of the qubit in the first EPR pair to Bob and one of the qubit in the second EPR pair to Carol. However, Bob and Carol do not share entangled qubits with each other before the breakdown. In this case, are we able to make an EPR pair shared between Bob and Carol despite having no quantum channel between them?

(a) Describe how you would accomplish this in fewer than five sentences.

(b) Complete the quantum circuit below to implement your algorithm. Note that no multi-qubit gates are allowed across different people's qubits after the breakdown point. Some qubits are initialized for each person; you do not have to use all of them.



4 Gate Equivalences

Task 4.1 (12 pts)

For the following sequences of gates, give their single gate equivalents. Write your answers using gates from the following set, with an appropriate phase factor:

$$\{I, X, Y, Z, H, S, S^\dagger, T, T^\dagger\}.$$

For example, $-Z$ or $e^{i\pi/16}Y$ are valid answers. For reference, we list the gate definitions here:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}, T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}, T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}$$

(a) XYZ

(b) $HZHZ$

(c) $TXT^\dagger X$

(d) $TZT^\dagger Z$

(e) $SXS^\dagger$

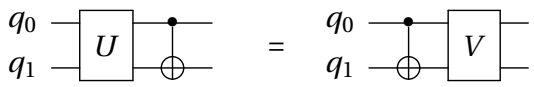
(f) $S^\dagger(TH)Z(TH)^\dagger$

Task 4.2 ($\star$ pts)

For the following scenarios of U, solve for a V such that the circuit equality holds. Write your answers using gates from the following set, with an appropriate phase factor:

$$\{I, X, Y, Z, H, S, S^\dagger, T, T^\dagger, CNOT_{0,1}, CNOT_{1,0}, SWAP_{0,1}\}.$$

Note that at most one gate per qubit is allowed. For example, $-(Z_0X_1)$ is a valid answer. To avoid ambiguity, we use subscripts to indicate the qubit that the gate acts on. For example, $Z_0(HS)_1 = (Z \otimes H)(I \otimes S)$ and $CNOT_{0,1}$ has a control qubit q_0 and a target qubit q_1.



(a) $U = X_0 I_1$

(b) $U = I_0 X_1$

(c) $U = X_0 Z_1$

(d) $U = (ZX)_0 I_1$

(e) $U = SWAP_{0,1}$

PROGRAMMING PORTION
This portion of the assignment has a total of 51 *points.*

5 Quantum Compiler

The programming portion of this assignment is to implement a mini quantum compiler! In lectures, we discussed one of the topics in quantum compilation, namely circuit synthesis (which deals with decomposing an arbitrary multi-qubit unitary into a sequence of one- or two-qubit quantum gates). The role of a quantum compiler goes beyond circuit synthesis; in this task, we will implement **register allocation**, which is another important task for the compiler.

What is a register? In our context, a quantum register is a list of qubits used by a quantum circuit. When executing the quantum circuit on an actual quantum computer, we need to map each qubit in the quantum register to a physical qubit in the quantum hardware.

Let's use an example to illustrate. Suppose a quantum computer (backend) physically has 5 qubits, labeled $q_0, q_1, \ldots, q_4$. We want to run a quantum circuit on 4 qubits. Now we have a choice to make: we can run the circuit on $\{q_0, q_1, q_2, q_3\}$ or on $\{q_0, q_1, q_2, q_4\}$, etc. Typically, we consider many constraints (such as quality of qubits, fidelity of gates, connectivity, etc) such that we would prefer one mapping over the other. In this question, we will simplify the problem by ignoring these constraints – for simplicity, we'd always prefer using the *lower-indexed qubits*. Thus, in the example earlier, the mapping $\{q_0, q_1, q_2, q_3\}$ wins.

Task 5.1 (12 pts)

A module needs to take a subset of the qubits in `QuantumRegister` as input. Therefore, we will implement the following two functions in the `QuantumRegister` class (and similarly for the `ClassicalRegister` class):

- `select(self, ids)`: it takes a list of indices (`ids`) and return a new `QuantumRegister` which includes only the qubits indexed by `ids`. If `ids` is empty, raise an `Exception('ids must be non-empty!')`. *Hint:* please read carefully the new `__init__` definition for the `QuantumRegister` for information about different ways to initialize a quantum register.

  For example, if a register contains a qubit array `[0,2,4,5]`, then `select` on `ids = [1,3]` is expected to return a quantum register object containing qubits `[2,5]`.

- `__add__(self, other)`: it concatenates two quantum registers. We assume `reg1 + reg2` results in a larger array containing qubits in `reg1` followed by qubits in `reg2`. You may assume that qubit indices in `reg1` and `reg2` do not overlap. *Hint:* make sure the returned `QuantumRegister` has the updated values for `size`, `array`, and `new_q`.

  For example, if the qubit array of `self` is `[0,3]` and that of `other` is `[1,4]`, then `__add__` is expected to return a register containing a qubit array `[0,3,1,4]`.

Note: `new_q` is a list of indicators for whether the qubits in this register are newly initialized, or referenced from existing qubits initialized outside of the circuit module. This is useful later when we need to keep track of the allocated qubits. For `ClassicalRegister`, there is an analogous list, namely `new_c`, to track.

Task 5.2 (6 pts)

In the `QuantumCircuit` class, implement the `allocate(self, ids, new)` function for using part of the qubits in the existing quantum circuits when constructing a new quantum circuit on top of it. It returns a tuple of `QuantumRegister` and `ClassicalRegister`.

`ids` is a list of indices of qubits from `self.qubits` to be included in the new register and `new` is a (non-negative) integer specifying the number of new qubits in addition to the existing ones. That is, the new register should contain the selected qubits from `self.qubits` followed by some new qubits. Similarly, `new` classical bits should be allocated for the `ClassicalRegister`. *Hint:* you might want to use the `select` function you defined in Task 5.1. The next Task 5.3 is a synthetic quantum program for you to test the implementation of `allocate`.

Task 5.3 ($\star$ pts)

To **test your implementation** for register allocation, you can run it on the provided modular quantum program in the starter code, called `synthetic_algo()`. It implements a quantum circuit as shown in Figure 1. Notice that there are repetitive modular structures in the circuit, indicated by the dashed lines. It is implemented by several nested function calls to sub-circuits, including `moduleA()` and `moduleB()`. The programs should print as QASM in the following form, where the nested modules are indented:

```
======<CPSC 447/547 QASM>======
Qreg: 4, Creg: 4
h qreg0 ,
h qreg1 ,
h qreg2 ,
```

USER PROGRAM

```python
def moduleA(qcA):
    qcA.cx(0, 2)
    qcA.cx(1, 0)
    qcA.cx(1, 2)
    qcA.tdg(0)
    # Allocate qubits for module B
    qids = [1]
    (qreg, creg) = qcA.allocate(qids, 1)
    # Initialize circuit for module B
    qcB = QuantumCircuit(qreg, creg)
    moduleB(qcB)
    qcA._append(qcB, qids, qids)
    return

def moduleB(qcB):
    qcB.t(0)
    qcB.cx(0, 1)
    qcB.cx(1, 0)
    return

def synthetic_algo():
    n = 4 # number of qubits
    qc = QuantumCircuit(n, n)
    for i in range(n):
        qc.h(i)
    for i in range(0, n, 2):
        qids = [i, i + 1]
        # Allocate qubits for module A
        (qreg, creg) = qc.allocate(qids, 1)
        # Initialize circuit for module A
        subcirc = QuantumCircuit(qreg, creg)
        moduleA(subcirc)
        # Add the module to the parent circuit
        qc._append(subcirc, qids, qids)
    return qc
```

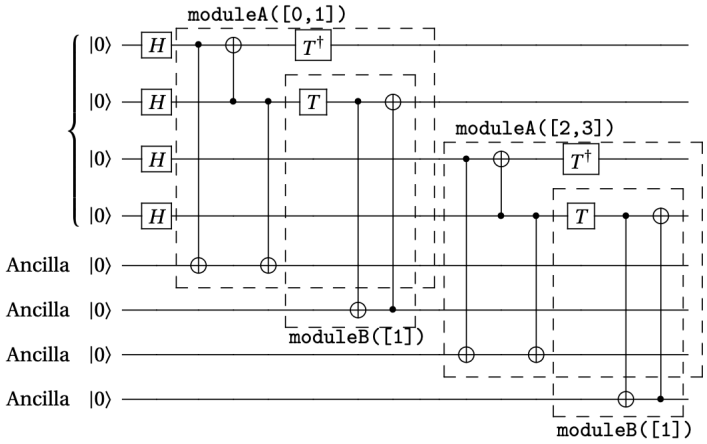


Figure 1: An example quantum circuit for `synthetic_algo()`. This is provided in `A2.py` for testing, as described in Task 5.3. The dashed lines in the quantum circuit indicate the boundaries of a module. "Ancilla" qubits are scratch qubits initialized and used within a module and are not accessed from outside the module.

```
h qreg3 ,
module qreg0 qreg1 , creg0 creg1
    Qreg: 3, Creg: 3
    cx qreg0 qreg0 ,
    cx qreg1 qreg0 ,
    cx qreg1 qreg0 ,
    tdg qreg0 ,
    module qreg1 , creg1
        Qreg: 2, Creg: 2
        t qreg1 ,
        cx qreg1 qreg0 ,
        cx qreg0 qreg1 ,
module qreg2 qreg3 , creg2 creg3
    Qreg: 3, Creg: 3
    cx qreg2 qreg0 ,
    cx qreg3 qreg2 ,
    cx qreg3 qreg0 ,
    tdg qreg2 ,
    module qreg3 , creg1
        Qreg: 2, Creg: 2
        t qreg3 ,
        cx qreg3 qreg0 ,
        cx qreg0 qreg3 ,
================================
```

Task 5.4 (10 pts)

Let's now use the `QuantumCircuit` class to implement a circuit for **gate teleportation**. In this task, we will learn how to use entanglement as a resource to perform remote gate between distant qubits. Suppose Alice and Bob each has a qubit and they would like to perform a CNOT

gate between them, but they are too far away physically to perform the gate directly. Fortunately, they have shared a copy of the EPR state, $|\psi_{AB}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ prior to the experiment. Now our goal is to help Alice and Bob implement the quantum programs for them to run locally to realize this CNOT gate remotely. Implement the quantum program for a remote (teleported) CNOT gate in `A2.py`.

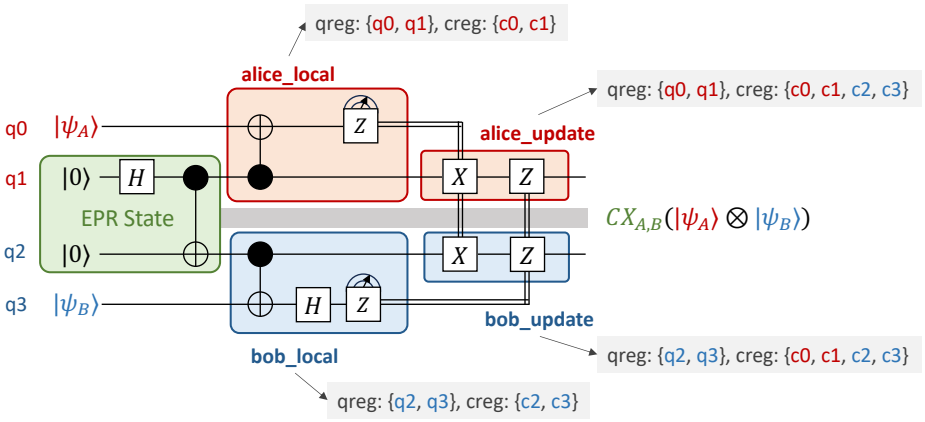


Figure 2: Remote CNOT gate between Alice and Bob, using a pre-generated EPR pair. In this modular program, Alice never accesses Bob's qubits and Bob never accesses Alice's. This can be seen from the fact that Alice's subcircuits use a quantum register containing only `{q0, q1}`. Similarly, Bob's quantum register has only `{q2, q3}`. In contrast, a shared classical register containing `{c0, c1, c2, c3}` are used for both `alice_update` and `bob_update`.

- In the `remoteCNOT()` function, we have initialized a quantum circuit with 4 qubits, which we label as $\{q_0, q_1, q_2, q_3\}$. We also initialized q_1 and q_2 in the EPR state $|q_1 q_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Then we create a quantum register containing q_0, q_1 for Alice and create a quantum register containing q_2, q_3 for Bob. From this point, Alice will not have access to Bob's qubits and vice versa.
- You job is to create a (sub)circuit `alice_qc` that would allow Alice to perform gates and measurements locally in `alice_local(alice_qc)`, and similarly for Bob.
- Once Alice and Bob finish their measurements, we will then create a `ClassicalRegister` containing both Alice's and Bob's measured bits and use that for `alice_update` as well as for `bob_update`.
- Also helpful is the gate defined as `conditional(self, cbits, gate, qubits)` in the `QuantumCircuit` class. It implements a gate conditioned on classical measurement results. For example, `qc.conditional([0,1], 'x', [2])` performs the X gate named on qubits of indices `[2]`, if the `cbits` stored at indices `[0,1]` of the classical register are all `True`. When printed in QASM, we append an `'_if'` to its name to indicate that it's a conditional gate.

To validate your implementation, you can check whether your quantum program `remoteCNOT()` prints the same output in QASM:

```
======<CPSC 447/547 QASM>======
Qreg: 4, Creg: 4
h qreg1 ,
```

```
cx qreg1 qreg2 ,
module qreg0 qreg1 , creg0 creg1
    Qreg: 2, Creg: 2
    cx qreg1 qreg0 ,
    measure qreg0 , creg0
module qreg2 qreg3 , creg2 creg3
    Qreg: 2, Creg: 2
    cx qreg2 qreg3 ,
    h qreg3 ,
    measure qreg3 , creg1
module qreg0 qreg1 , creg0 creg1 creg2 creg3
    Qreg: 2, Creg: 4
    x_if qreg1 , creg0
    z_if qreg1 , creg3
module qreg2 qreg3 , creg0 creg1 creg2 creg3
    Qreg: 2, Creg: 4
    x_if qreg2 , creg0
    z_if qreg2 , creg3
==============================
```

Task 5.5 (8 pts)

Next, we will consider the **qubit mapping** problem in the quantum compilation tool flow. In `Backend`, implement the `alloc(self, n)` function for assigning available qubits from the backend, where `n` is a (non-negative) integer for the number of new qubits to allocate. If the backend allows this operation, update the state of the `Backend` instance, including its `num_q`, `in_use` and `all_qubits`, and return a list of indices (of length `n`) for those new qubits; otherwise raise an `Exception` following the explanation below.

`Backend` is a class containing information about the quantum hardware, including number of qubits (`num_q`), number of qubits that have already been allocated (`in_use`), a list of all `Qubits` (`all_qubits`), a name (`label`), and an indicator for whether the backend has a variable | infinite | idealized size (`variable==True`) or a fixed | finite | realistic size (`variable==False`).

We assume to allocate lower-indexed qubits first. So, if backend is finite and `(in_use + n)` exceeds `num_q`, raise an `Exception('Allocation error: not enough qubits!')`, otherwise update the status of the backend and return the list of indices.

Task 5.6 (15 pts)

Implement `compile_fully_connected(self, backend, qubits_mapping, cbits_mapping)`, as part of the `QuantumCircuit` class, to perform code flattening and qubit mapping on the input modular program. Here, input argument `backend` is the intended hardware backend to execute the circuit (assumed to be fully connected, i.e., a complete graph), `qubits_mapping` is an optional existing mapping for indexing qubits, and `cbits_mapping` is an optional mapping for indexing classical bits. A mapping is a `Dictionary` of key-value pairs, where the key is the index of a qubit in the quantum register of the circuit and the value is the index of the qubit in the backend. The function should return a flattened and mapped circuit as a list of gates – "flattened" means that the returned list should contain only quantum gates (i.e., without subcircuits) and "mapped" means that the qubits in the circuit are indices to the backend. Raise an `Exception('Backend too small!')` if the number of qubits needed in the circuit exceeds that of the backend. Note that a separate `cbit_mapping` is used for indexing classical register, because the size of a quantum register can be different from that of a classical register, as seen

in Task 5.4. You can assume that `qubits_mapping` and `cbits_mapping` are consistent, that is, when `qubits.new_q[i]==True`, we can allocate a new qubit from the backend (`[qubit_idx] = backend.alloc(1)`) and then set `qubits_mapping[i] = qubit_idx` and `cbits_mapping[i] = qubit_idx`.

Detailed instructions can be found in the starter codes `A2.py`. The way one can implement this quantum circuit compiler is similar to implementing a compiler for a classical circuit. In **Task 5.6.1**, you will first allocate qubits for the circuit. Then in **Task 5.6.2**, you will recursively compile the circuits. For reference, the expected output of `compile_fully_connected()` for the quantum circuit in Task 5.3 is the following:

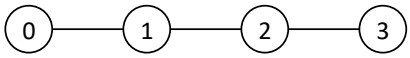
```
======<CPSC 447/547 QASM>======
Qreg: 8, Creg: 8
h qreg0 ,
h qreg1 ,
h qreg2 ,
h qreg3 ,
cx qreg0 qreg4 ,
cx qreg1 qreg0 ,
cx qreg1 qreg4 ,
tdg qreg0 ,
t qreg1 ,
cx qreg1 qreg5 ,
cx qreg5 qreg1 ,
cx qreg2 qreg6 ,
cx qreg3 qreg2 ,
cx qreg3 qreg6 ,
tdg qreg2 ,
t qreg3 ,
cx qreg3 qreg7 ,
cx qreg7 qreg3 ,
==============================
```

And the expected output for `remoteCNOT().compile_fully_connected()` in Task 5.4 is the following:

```
======<CPSC 447/547 QASM>======
Qreg: 4, Creg: 4
h qreg1 ,
cx qreg1 qreg2 ,
cx qreg1 qreg0 ,
measure qreg0 , creg0
cx qreg2 qreg3 ,
h qreg3 ,
measure qreg3 , creg3
x_if qreg1 , creg0
z_if qreg1 , creg3
x_if qreg0 , creg0
z_if qreg0 , creg3
==============================
```

Task 5.7 (⋆ pts)

Implement the `compile(self, backend, mapping)` function as in the previous task, but this time without assuming that the intended hardware backend is fully connected. Note that you can still assume that the backend is connected (that is no disconnected components). This means we need to update our definition of `Backend` to include information, and then implement an algorithm to resolve the connectivity constraints:



$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 3: An example backend of 4 qubits (left) and its adjacency matrix (right). Here `cx(0,3)` cannot be performed directly, because qubit 0 and qubit 3 are not connected.

- In `Backend`, add an input argument `adj_matrix`, which is the adjacency matrix describing the connectivity graph of qubits in the backend. Thus, for a backend of `num_q` qubits, `adj_matrix` is a size `num_q`×`num_q` matrix with type `np.ndarray`. Lastly, do not forget to initialize a field for `adj_matrix`. Note: an infinite/variable backend can be assumed to be fully connected, in which case `adj_matrix` can be set to `None`.
- In additional to all requirements in Task 5.6, implement an algorithm, in `compile(self, backend, mapping)`, that replaces any two-qubit gates (e.g., `cx(0,3)` on the backend shown in Figure 3) acting on qubits that are not directly connected by the following sequence of gates:

  1. Swapping one of the qubit to be next to the other qubit. For instance, apply `swap(0,1)`, then `swap(1,2)`.
  2. Apply the two qubit gates on the connected qubits. In our example, `cx(2,3)`.
  3. Finally, swapping the qubit back to its original location. For instance, apply `swap(1,2)`, then `swap(0,1)`.

  Therefore, `compile` function will return a list of gates, where every two-qubit gate acts only on a pair of qubits that are connected. Note that, since `swap` gate is not part of the instruction set in `QuantumCircuit`, we need to use the `cx` gates to implement it. You can assume that the original program does not use the three-qubit gate, `toffoli`, from the instruction set.