Assignment 2

***Written Part Due**: Monday, September 29th, 2025 @ 11:59PM*
***Programming Part Due**: Monday, October 13th, 2025 @ 11:59PM*

CPSC 4470/5470 Introduction to Quantum Computing (Fall 2025)

1 Introduction

Welcome to Assignment 2 for CPSC 4470/5470 (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 case.

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.

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 $O = \sigma_x \otimes I$.

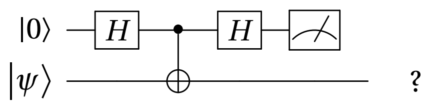
(b) Only measure qubit B in the x basis, that is, with observable $O = I \otimes \sigma_x$.

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

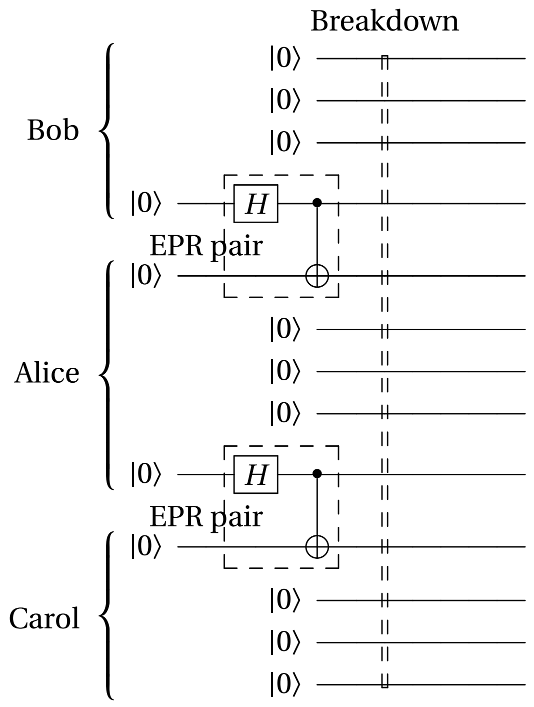
(b) "W state": $|\psi_{ABC}\rangle = \frac{1}{\sqrt{3}}|001\rangle + \frac{1}{\sqrt{3}}|010\rangle + \frac{1}{\sqrt{3}}|100\rangle$

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) $SXS^\dagger$

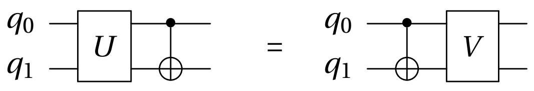
(f) $S^\dagger(TH)Z(TH)^\dagger$

Task 4.2 (⋆ 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:

$$\{I, X, Y, Z, H, S, S^\dagger, T, T^\dagger, \mathrm{CNOT}_{0,1}, \mathrm{CNOT}_{1,0}, \mathrm{SWAP}_{0,1}\}.$$

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 $\mathrm{CNOT}_{0,1}$ has a control qubit q_0 and a target qubit q_1.



(a) $U = X_0I_1$

(b) $U = I_0X_1$

(c) $U = X_0Z_1$

(d) $U = (ZX)_0I_1$

(e) $U = \mathrm{SWAP}_{0,1}$

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

5 Quantum Compiler

In lectures, we explored the **circuit synthesis** problem in quantum computing. We saw how an arbitrary multi-qubit unitary can be decomposed into one- and two-qubit gates. A quantum compiler, however, does much more. In this assignment, you will build parts of a **mini quantum compiler** that handles: register selection and concatenation (for grouping and composing qubits), register allocation (for initializing new qubits/cbits), module flattening and mapping (for turning nested modules into a flat, hardware-indexed gate sequence).

Why does this matter? Real devices have constraints, such as limited qubit count, connectivity, and gate set. The job of a compiler is to bridge logical circuits and physical hardware by choosing registers, mapping qubit indices, and restructuring circuits while preserving behavior.

What is a register? A **quantum register** is an ordered list of qubits used by a circuit. When running on a hardware, each logical qubit in the register must be mapped to a physical hardware qubit. To keep this assignment focused, we will ignore quality/fidelity/connectivity preferences in the selection of physical qubits, and simply *prefer lower-index hardware qubits* when we have a choice.

Like in Assignment 1, we will use `QuantumRegister` and `ClassicalRegister` data models. Importantly, a `QuantumRegister` tracks: `size` (number of qubits), `array` (list of qubit indices), `new_q` (Boolean list aligned with `array`, marking which qubits are *newly created* in the current context as opposed to references to existing qubits). A `ClassicalRegister` analogously tracks `new_c`. One of your tasks is, for example, to understand how could `new_q` and `new_c` help a compiler decide when to allocate versus reuse qubit resources, or when to free ancillary qubits.

Task 5.1 (12 pts)

Implement the following in `QuantumRegister` (and analogously in `ClassicalRegister`):

- `select(self, ids)`: it takes as input `ids` (list of integer positions into `self.array`) and returns a new `QuantumRegister` containing only the qubits at those positions, preserving order.
  - *Note:* `size, array, new_q` of the returned object must reflect the selection.
  - *Exception:* If `ids` is empty, raise `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.
  - *Example:* if `self.array == [0,2,4,5]`, then `select([1,3])` returns a register with `array == [2,5]` and the corresponding `new_q` entries for positions 1 and 3.
- `__add__(self, other)`: it concatenates two quantum registers, returning a new register whose `array` is `self.array + other.array`.
  - *Assume:* No overlapping indices across the two input registers.
  - *Note:* `size`, `array`, and `new_q` must be correctly updated.
  - *Example:* if `self.array == [0,3]` and `other.array == [1,4]`, then `__add__` is expected to return a register with `array == [0,3,1,4]`.

Why this matters: Selection lets modules slice registers; concatenation lets modules compose them. They are two essential compiler operations.

Task 5.2 (6 pts)

When writing a quantum program, it is convenient to call a function that contains a subcircuit. This new subcircuit might want to *reuse* some qubits from the current function but also request *fresh* ones. Let's see how we can support this in our compiler.

In `QuantumCircuit`, implement `allocate(self, ids, new)` that builds the registers for a new quantum circuit (sub)module:

- *Inputs:* `ids` (positions into the current register's existing `self.qubits` to *reuse*) and `new` (non-negative integer as the number of *new qubits* and new cbits to allocate *after* the reused ones).
- *Output:* `(qreg, creg)`, a tuple of `QuantumRegister` and `ClassicalRegister` where the selected existing qubits appear first, followed by the newly allocated ones.
- *Hints:* you might want to reuse the `select` function from Task 5.1. Also, ensure the `new_q` and `new_c` flags correctly mark only the appended items as "new".

The next Task 5.3 is a synthetic quantum program for you to test the implementation of `allocate`.

Task 5.3 ($\star$ pts)

Use the provided `synthetic_algo()` to test your `allocate`, `select`, and `__add__` implementations. It builds a circuit with repeated, nested modules as shown in Figure 1. Notice that there are repetitive modular structures in the circuit, indicated by the dashed lines. The program should print as a structured, properly indented QASM format. What to check:

- Register sizes at each module boundary (at line `Qreg: x, Creg: y`)
- Indentation showing module nesting
- Qubit indices within each indented module refer to their ids in local register.
- Reused vs new qubits align with your `new_q` flags
- For reference, the example output is provided below:

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

In this task, you 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

USER PROGRAM

```
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 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
```

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

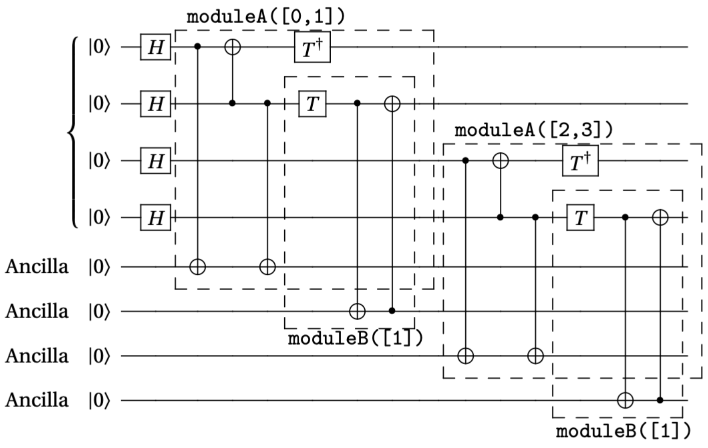


Figure 1: An example quantum circuit for `synthetic_algo()`. This is provided in `A2.py` for testing, as described in Task 5.3. The dashed lines in the quantum circuit indicate the boundaries of a module. “Ancilla” qubits are scratch qubits initialized and used within a module and are not accessed from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

run locally to realize this CNOT gate remotely. Implement the quantum program for a remote (teleported) CNOT gate in `A2.py`.

Implement `remoteCNOT()` in `A2.py` using the 4-qubit circuit pattern in Fig. 2. In the provided function, we have 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.

- Build `alice_qc` and `bob_qc` (local subcircuits only on their own registers).
  - Use `alice_local()` and `bob_local()` to perform their local gates and measurements.
  - After the measurements, build a shared `ClassicalRegister` containing both parties' outcomes
  - Then they each perform a local update in `alice_update()` and `bob_update()` respectively.
- *Hint:* Use `conditional(self, cbits, gate, qubits)` for updates. For example, the operation `qc.conditional([0,1], 'x', [2])` applies X gate to qubit indexed at `[2]` if the `cbits` stored at indices `[0,1]` are both `True`. When printed in QASM, these appear as `'x_if'` or `'z_if'`, etc.
- For reference, your QASM printout for `remoteCNOT()` should match the following example (order and indentation matter):

```
======<CPSC 447/547 QASM>======
Qreg: 4, Creg: 4
h qreg1 ,
cx qreg1 qreg2 ,
module qreg0 qreg1 , creg0 creg1
```

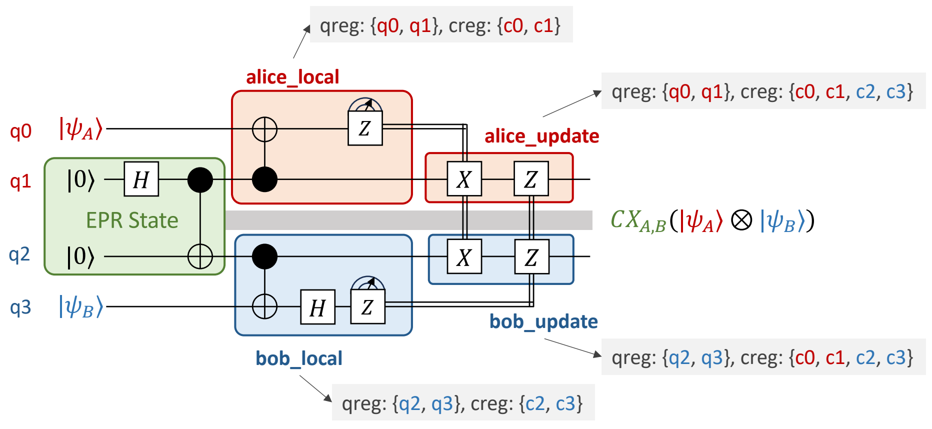

qreg: {q0, q1}, creg: {c0, c1}
alice_local
qreg: {q0, q1}, creg: {c0, c1, c2, c3}
alice_update
q0
$|\psi_A\rangle$
q1
$|0\rangle$
H
EPR State
q2
$|0\rangle$
q3
$|\psi_B\rangle$
Z
X
Z
H
Z
X
Z
$CX_{A,B}(|\psi_A\rangle \otimes |\psi_B\rangle)$
bob_update
qreg: {q2, q3}, creg: {c0, c1, c2, c3}
bob_local
qreg: {q2, q3}, creg: {c2, c3}

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`.

```
    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 a quantum compiler. In the setup, `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 Backend name (`label`), and a Boolean indicator: if `variable == True`, the backend has infinite size and if `variable == False`, the backend has a fixed, finite size.

In `Backend`, implement `alloc(self, n)` that assigns available hardware qubits.

- *Input:* It takes as input `n`, a (non-negative) integer for the number of new qubits to allocate.
- *Output:* Update the internal fields of `Backend`, including its `num_q`, `in_use` and `all_qubits`, and return a list of `n` indices for those new qubits just allocated.
- *Note:* Assume `n` is non-negative; if `n == 0`, it should return `[]` and make no changes.
- *Note:* Lower-indexed qubits should be allocated first.

- *Exception:* if `variable == False` (finite hardware) and `(in_use + n)` exceeds `num_q`, raise `Exception('Allocation error: not enough qubits!')`.

Task 5.6 (15 pts)

In `QuantumCircuit`, implement `compile_fully_connected(self, backend, qubits_mapping, cbits_mapping)` to perform code flattening and qubit mapping on the input modular program.

- *Input:* `backend` (the target hardware, assumed to be fully connected where gates between any qubits are allowed), `qubits_mapping` (an optional dict mapping for indexing qubits `{logical_qubit_idx -> backend_qubit_idx}`, and `cbits_mapping` (optional mapping for indexing classical bits).
- *Output:* It returns a flat list of gates (here "flat" means no sub-circuits) acting on qubits mapped to their backend indices.
- *Exception:* Raise `Exception('Backend too small!')` if the number of qubits needed in the circuit exceeds that of the backend.
- *Note:* If `qubits.new_q[i]==True`, call `[qubit_idx]=backend.alloc(1)` to obtain a fresh hardware index, then set both `qubits_mapping[i] = qubit_idx` and `cbits_mapping[i] = qubit_idx` to that index. You can assume that `qubits_mapping` and `cbits_mapping` are consistently indexed. However, 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.
- *Hint:* Additional instructions can be found in the starter codes `A2.py`. What to implement: In **Task 5.6.1**, you will first allocate qubits to ensure each (logical) qubit has a properly assigned physical hardware index. Then in **Task 5.6.2**, you will traverse the modules and recursively flatten the circuits, translating their gates to operations acting on mapped indices.
- 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 ,
===============================
```

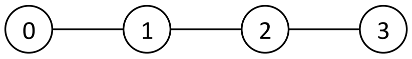
And the expected output for `remoteCNOT().compile_fully_connected()` in Task 5.4 is the following:

```
======<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)

Upgrade your compiler to respect a hardware backend that is not fully connected. Note that you can still assume that the backend is connected (that is no disconnected groups of qubits). This means we need to update our definition of `Backend` to include hardware topology information, and then implement a procedure to resolve the connectivity constraints. The goal is to ensure every two-qubit gate (e.g., `cx(u, v)`) acts on a connected pair of qubits. When `u,v` are not adjacent, route via SWAPs: 1. Bing qubits together along a shortest path: `swap` neighbor pairs until the control/target are adjacent. 2. Apply the intended two-qubit gates. 3. Swap back to restore original qubit placements.

- In `Backend`, it should now include `adj_matrix: np.ndarray` (with shape `num_q×num_q`), representing the adjacency matrix of an undirected connectivity graph between qubits in the backend. For a variable (infinite) backend, you may set `adj_matrix = None` and treat them as fully connected.
- *Output:* `compile()` returns a list of gates, where every two-qubit gate acts only on a pair of qubits that are connected.
- *Example:* `cx(0,3)` on the backend shown in Figure 3 acting on qubits that are not directly connected. So it should be translated to 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)`.
- Note: since `swap(a,b)` gate is not part of the instruction set in `QuantumCircuit`, it must be implemented using CNOTs (e.g., `cx(a,b); cx(b,a); cx(a,b)`) the `cx` gates to implement it. You can assume that the original program does not contain any three-qubit gate, e.g., `toffoli`.



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

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.