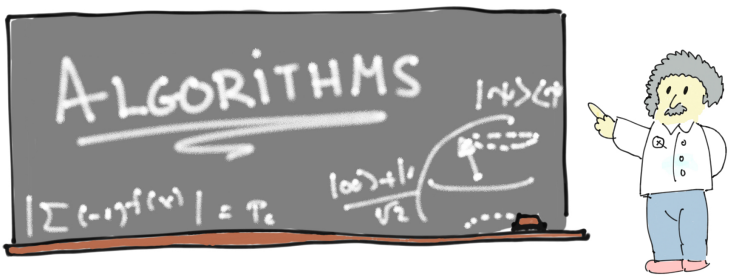

Algorithms: Adapting to the topology [200 points]

Version: 1

Algorithms

The **Algorithms** category challenges will serve to give you a sense of commonly-used routines that are helpful when using real quantum computers. The quantum algorithms that you will see in this category are procedures that mathematically demonstrate some advantage over traditional/classical solutions such as the Deutsch-Jozsa algorithm, applications of the Quantum Fourier Transform (QFT), Grover's algorithm, and Quantum Phase Estimation (QPE).

Although these algorithms have very specific applications, they are commonly related to real-world problems. One of the clearest examples of this is the well-known Shor factorization algorithm, which revealed that finding prime factors of a number is equivalent to finding the period of certain functions, which can be achieved through QPE. It is for this reason that it is so important to know these basic ideas of quantum computation. Let's get to it! And good luck!

Problem statement [200 points]

When we run an algorithm, we usually work without worrying about how the hardware underneath is built. We can afford to do this thanks to a set of post-processing steps that take place between the submission of your code and what the computer finally receives. In this challenge, we will analyze a specific case where a coded quantum circuit cannot be executed on a real quantum computer without some thinking.

A common limitation of real devices is that not all connections between qubits are allowed. For example, for certain hardware structures (or topologies), we cannot apply a CNOT gate between qubits 0 and 2. There must be some trick that allows us to work around this!

We will work with the topology in Figure 1.

Here, for example, we *can* implement a CNOT gate between qubits 0 and 1 or

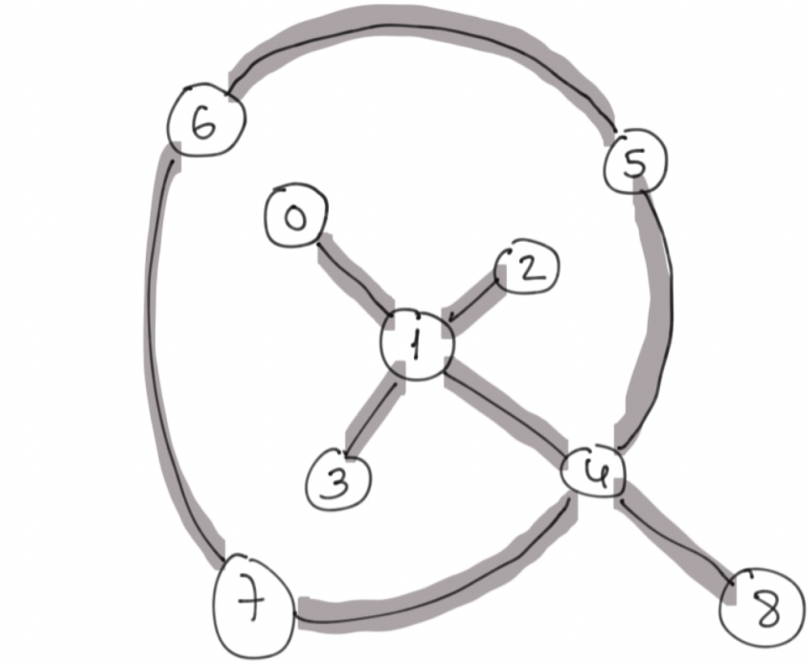

Figure 1: Topology

between qubits 4 and 8. However, connections between 0 and 5 or 4 and 6 are not allowed. This is not a problem, since we can construct pseudo-connections between these qubits by using the SWAP gate (represented by two blades joined together):

$$\text{SWAP}|\phi_i\rangle \otimes |\phi_j\rangle = |\phi_j\rangle \otimes |\phi_i\rangle$$

Using this, we can arrive at the following equivalence that relates CNOT and SWAP gates as seen in Figure 2:

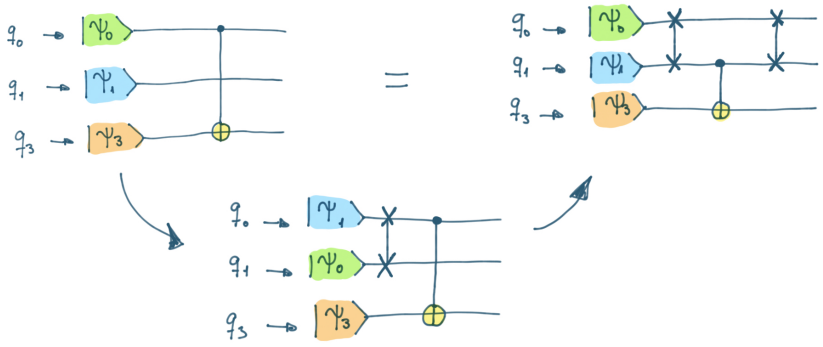

Figure 2: CNOT equivalence using SWAP gates

We want to make a connection between qubits 0 and 3, but since there is no direct connection, we can make a bridge through qubit 1. If we want to make a connection between qubits 2 and 8, we would have to use two qubits to facilitate the connection, as we can see in Figure 3.

Given a particular CNOT operator, you must determine the minimum number of auxiliary SWAP gates needed to make the desired pseudo-connection. You are provided with a fake "device" represented by a graph. The graph is provided in the form of a dictionary. The keys represent each node, and their associated values are lists corresponding to the nodes they are directly connected with.

The provided template `adapting_topology_template.py` contains a function called `n_swaps` that you need to complete. In this function, you will develop an algorithm for counting the minimum number of swaps needed to create the equivalent CNOT gate that is provided as input to the problem. This function needs to return an integer corresponding to the minimum number of swaps.

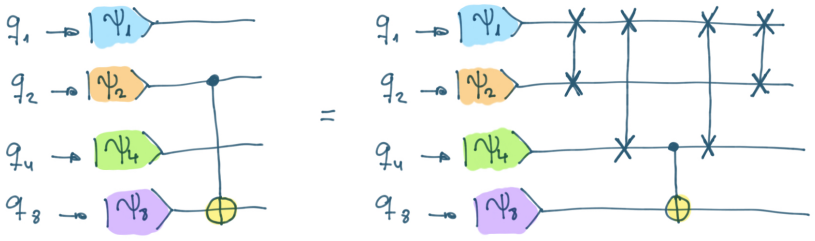

Figure 3: CNOT(2,8) equivalence

Input

- `list(int)`: A list of wires (integers) that correspond to the control and target qubits of a CNOT gate.

Output

- `int`: Minimum number of swaps to be used.

Acceptance Criteria

In order for your submission to be judged as "correct":

- The outputs generated by your solution when run with a given `.in` file must match those in the corresponding `.ans` file.
- Your solution must take no longer than the `60s` specified below to produce its outputs.

You can test your solution by passing the `#.in` input data to your program as stdin and comparing the output to the corresponding `#.ans` file:

```
python3 {name_of_file}.py < 1.in
```

WARNING: Don't modify the code outside of the `# QHACK #` markers in the template file, as this code is needed to test your solution. Do not add any print statements to your solution, as this will cause your submission to fail.

Specs
Time limit: **60 s**

Version History

Version 1: Initial document.