Games: Find the car [400 points]

Version: 1

Games

Quantum computers help us solve many problems, but we can also use them to explore some of the most mysterious aspects of quantum mechanics. We learn so much better when we have fun, so why not use some crazy games and experiments to explore the boundaries of quantum theory? These fun coding challenges, ranging from entangling full-blown animals to using quantum circuits to cheat our way into victory, teach us a lot about why quantum computing is so powerful.

In the **Games** category, we will be exploring some of the weirdest quantum experiments proposed in the literature. These will enlighten us about how quantum mechanics is different from classical physics, but will also give rise to deeper philosophical questions. Let's have some fun!

Problem statement [400 points]

The objective of this challenge is to find out which of the four doors in Figure 1 hides a car behind it with 100% probability. We can ask where the car is through an oracle like the one in Figure 1. If the car is behind the queried door, the oracle will output a 1 on the third qubit. Otherwise, it will output a 0.

Let's see a code example for checking where the car is in a classical manner:

```
import pennylane as qml

def oracle_example():
    # the car is behind door |01>
    qml.PauliX(wires = 0)
    qml.Toffoli(wires = [0,1,"sol"])
    qml.PauliX(wires = 0)

def check_door(oracle):
```

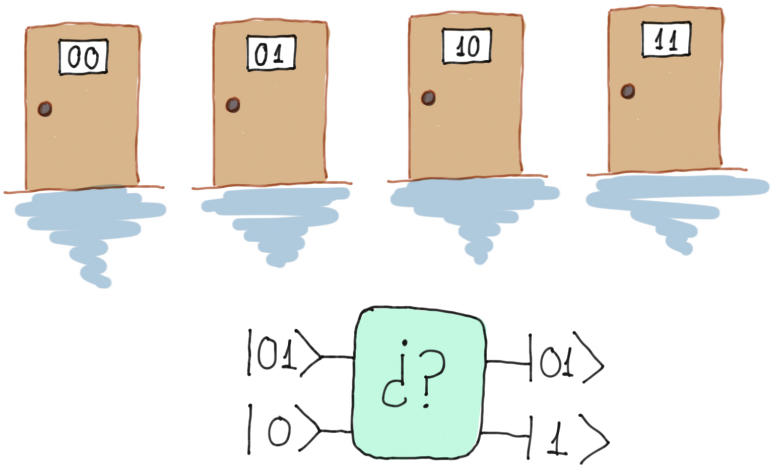

Figure 1: Where is the car?

```
    dev = qml.device("default.qubit", wires = [0,1, "sol"], shots = 1)
    @qml.qnode(dev)
    def circuit():

        # check behind door 10 by preparing state |10>
        qml.PauliX(wires = 0)

        # Let's apply the oracle
        oracle()

        return qml.sample()

    return circuit()


check_door(oracle_example)
```

In the code above, the car is located behind door number 1 (represented by $|01\rangle$), but we checked door number 2 (represented by $|10\rangle$). The output of the code is the list [1,0,0]. As the third qubit is in state 0, we deduce that the car is not behind door number 2.

If we try door number 1 ($|01\rangle$), we can see that the third qubit will change to 1.

```
def check_door(oracle):
```

```
    dev = qml.device("default.qubit", wires = [0,1, "sol"], shots = 1)
    @qml.qnode(dev)
    def circuit():

        # check behind door 01 by preparing state |01>
        qml.PauliX(wires = 1)

        # Let's apply the oracle
        oracle()

        return qml.sample()

    return circuit()
```

`check_door(oracle_example)`

Indeed, the output of the above code is the list [0,1,1].

Classically, and in the worst-case scenario, it takes three attempts to know where the car is. But with the help of a quantum computer, we can reduce the number of queries. One strategy to solve this problem is to use Grover's algorithm. However, we now impose an additional constraint: we can only use gates that affect a single qubit. The challenge is to determine where the car is with only two queries, i.e., by running two circuits that only call the oracle once.

In the provided template `find_the_car_template.py`, there are a few functions that you need to complete:

- `find_the_car`: two different circuits (your two allotted queries) that use the input oracle function are constructed and executed. The circuit outputs are used to determine the door concealing the car.
- `circuit1` and `circuit2`: design two circuits that query the input oracle function and, when both are run, provide you with sufficient information to determine the door concealing the car (they both return `qml.sample()`).

Input

- `list(int)`: A list of integers that define how the oracle is created.

Output

- `int`: An integer indicating the door concealing the car (0, 1, 2, or 3).

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.