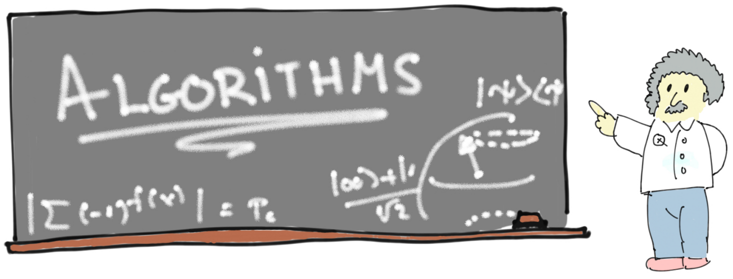

Algorithms: The Deutsch-Jozsa Problem [100 points]

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 [100 points]

In this challenge question, you will be working with the well-known Deutsch-Jozsa problem. There is no explicit *need* for a quantum solution to this problem, as there exist classical algorithms that work too. However, the quantum solution is exponentially more efficient. The Deutsch-Jozsa problem is simple in nature; we are given a black-box function $f : \{0,1\}^n \to \{0,1\}$ (i.e., a function that takes an n-bit string and maps it to a single bit) that we are guaranteed does one of the following things:

1. always outputs either 1 or 0 (f is said to be **constant**).
2. half of its domain yields an output of 0, the complementary half yields 1 (f is said to be **balanced**).

For instance, for $n = 2$, here is an example of a balanced function.

bit0	bit1	f(bit0, bit1)
0	0	1
0	1	0
1	0	1
1	1	0

Solving the Deutsch-Jozsa problem involves determining whether the black-box function f is constant or balanced in as few queries to the function as possible. It turns out that with a quantum computer, we only need *one* query to the function to determine whether it is constant or balanced, whereas in the worst case classical scenario, we need $2^{n-1}+1$ queries. In the quantum case, to solve this problem, we assume that we have some quantum circuit implementation of the function f given to us, called an "oracle" U_f. Essentially, the oracle U_f performs the following map: $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$. The first state $|x\rangle$ is the domain's register, and the second, $|y\rangle$, can be thought of as an ancillary qubit. The algorithm consists of executing the circuit in Figure 1.

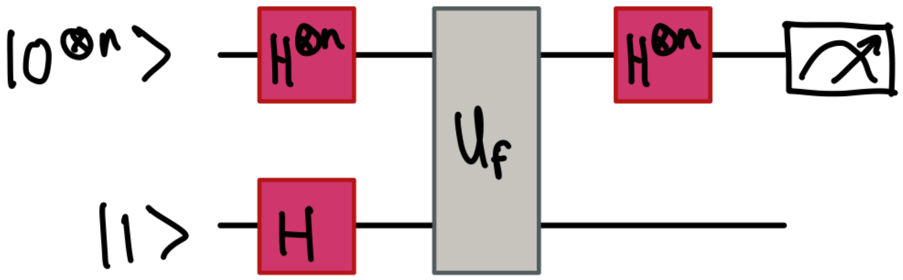

Figure 1: Quantum circuit implementing the Deutsch-Josza algorithm

The final measurement corresponds to measuring the probability that the first register is in the $|0^{\otimes n}\rangle$ state. Algebraically, the probability ends up being

$$P_0 = \left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \right|^2 .$$

We can see that if f is constant, it will give us probability 1 of seeing the state $|0\rangle^n$, whereas if f is balanced, we would have probability 0 of seeing that state.

In this exercise, we will solve the Deutsch-Jozsa problem for the case of an oracle that takes two qubits as input. Using an ancillary/auxiliary qubit (i.e., three qubits/wires in total), your code will do the following:

- Process the three qubits in a specific manner before they reach the oracle U_f.

- Process the qubits after the oracle.
- Measure the non-ancillary qubits and determine whether the oracle is constant or balanced.

The template file `deutsch_jozsa_template.py` contains one function aptly called `deutsch_jozsa`. In this function, you must define a `circuit` that implements some pre- and post-processing surrounding the `oracle()` to three qubits. Then you must devise a way to, given the output of `circuit()`, determine if the input `oracle()` was a constant or balanced function.

Input

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

Output

- `str`: "constant" or "balanced"

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.