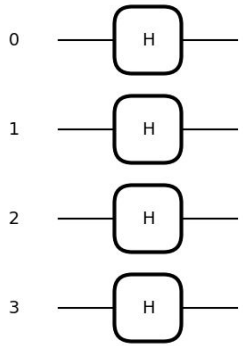
Introduction to Quantum Computing Pennylane: Oracles and promises

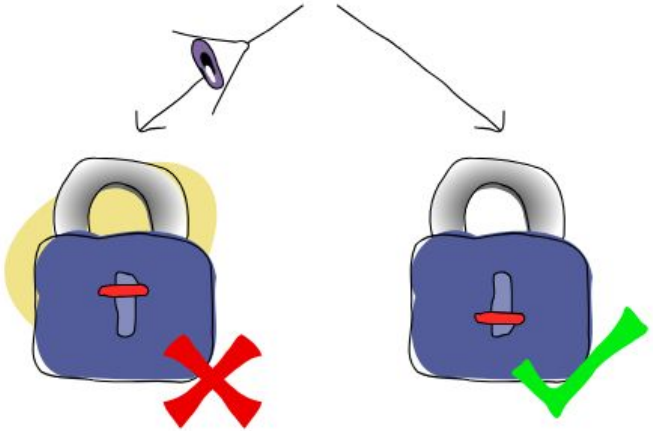
Chris Heunen

Exercise 1: Uniform superpositions




```
n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)

@pl.qnode(dev)
def uniformsuperposition():
    """Build a circuit that creates an n-qubit uniform superposition"""
    return pl

pl.drawer.use_style("black_white")
pl.draw_mpl(uniformsuperposition)();
```

A uniform superposition will include the solution string $|\mathbf{s}\rangle$. But by itself, this doesn't help us break the lock, since the superposition includes everything else as well! Once we open the quantum computer and look inside, we may observe the correct lock combination, but we might also see an incorrect combination, due to the probabilistic nature of quantum computation. Let's draw a cartoon of the case for the single-bit lock:



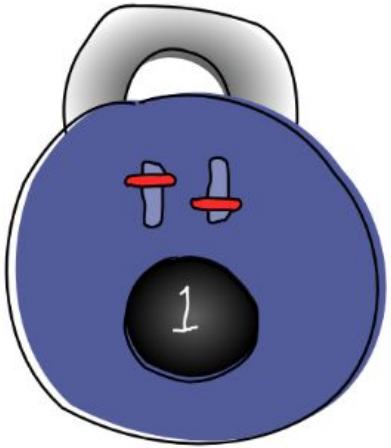
We still only have a 50% chance of landing on the correct answer when the computation finishes. More generally, if we start in the even superposition Eq. (1) for n bits, our chance of observing the correct answer is $1/2^n$. These are the same odds as a random *classical* guess! So, we see the basic dilemma (or rather, di^{n}lemma). A quantum superposition may look like it can result in an exponential number of things being done at once, but once we take a measurement, we will only get a random snapshot of what it's done.

Quantum algorithms are all about shuffling around this exponential collection of terms in the superposition so that, when we observe the system, our random snapshot has a high probability of showing us the thing we are looking for. And given that our algorithms involve randomness, we also should be comparing them to *random classical algorithms*, like guessing the combination.

Let's think a bit more about how this lock breaking actually works. Call the correct combination for an n-bit lock $\mathbf{s} \in \{0,1\}^n$. We can model the lock as a function f from n-bit strings $\mathbf{x} \in \{0,1\}^n$ to a bit $y \in \{0,1\}$, with $f(\mathbf{x}) = 0$ meaning "the combination doesn't work" and $f(\mathbf{x}) = 1$ meaning "the combination works". More concisely,

$$f(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} = \mathbf{s} \\ 0 & \text{otherwise.} \end{cases}$$

We encode this as a unitary operator, U_f, which acts on basis states as



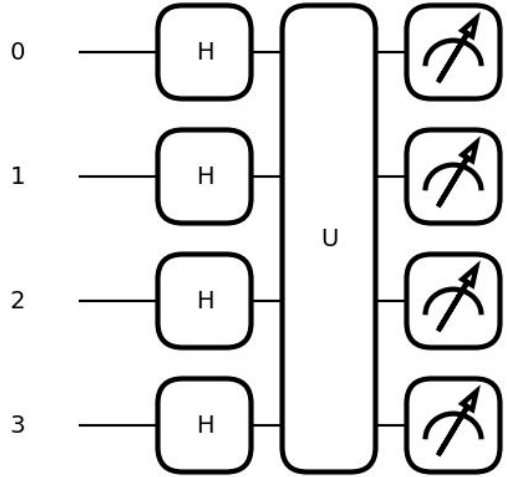
$$\begin{aligned} U_f|\mathbf{x}\rangle &= (-1)^{f(\mathbf{x})}|\mathbf{x}\rangle \\ &= \begin{cases} -|\mathbf{x}\rangle & \mathbf{x} = \mathbf{s} \\ |\mathbf{x}\rangle & \text{otherwise.} \end{cases} \end{aligned}$$

Exercise 2: Oracles

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```python
def oracle_matrix(key):
    """Create the unitary matrix corresponding to the binary key (list[int])
       e.g. key=[0,1,1] should give the diagonal matrix [1,1,1,-1,1,1,1,1]"""
    # Hint: use np.ravel_multi_index
    matrix = np.identity(2**len(key))
    return matrix

print(oracle_matrix([0,1,1,0]))
```

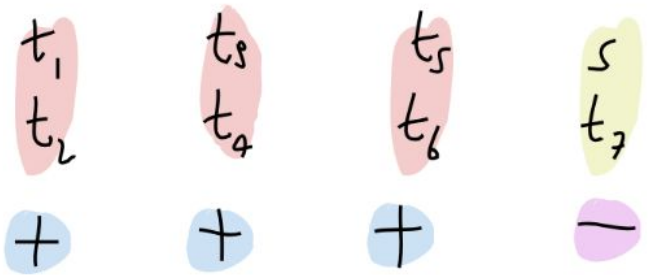
Exercise 2: Oracles




```
@pl.qnode(dev)
def oracle_circuit(key):
    # Hint: use pl.QubitUnitary
    return pl.probs(wires=range(n_bits))

pl.draw_mpl(oracle_circuit)([0,1,1,0]);
print(oracle_circuit([0,1,1,0]))
```

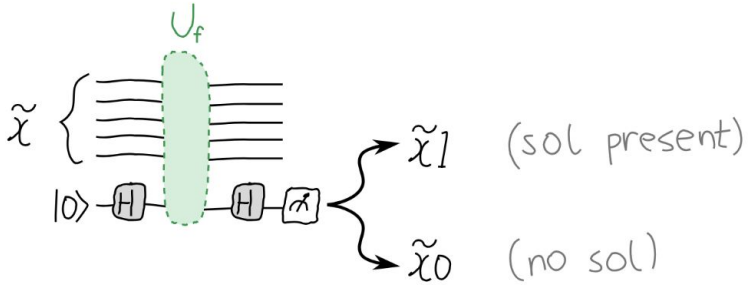
Applying the oracle by itself is not enough to improve our lock picker. The problem is that we introduce a phase change which is unobservable without further processing. In order to make some progress, we must *combine* states to induce a *relative* change of phase which is detectable. To illustrate, let's take a superposition of two states $|\mathbf{x}\rangle$ and $|\mathbf{y}\rangle$, and apply the oracle:

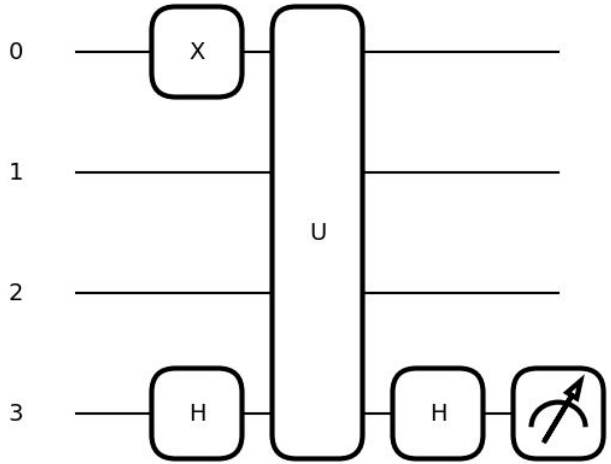
$$
\begin{aligned}
|\psi_{\mathbf{xy}}\rangle &= \frac{1}{\sqrt{2}}(|\mathbf{x}\rangle + |\mathbf{y}\rangle) \\
&\mapsto \frac{1}{\sqrt{2}}\Big((-1)^{f(\mathbf{x})}|\mathbf{x}\rangle + (-1)^{f(\mathbf{y})}|\mathbf{y}\rangle\Big).
\end{aligned}
$$



This leads to a rudimentary algorithm: test solutions in pairs. This is almost the brute force algorithm, except that instead of searching through 2^n combinations, we search through 2^{n-1} pairs, so we improve by a constant factor. Once we have identified the correct pair, we can just test both classically. This last step doesn't scale with n, so we have a very modest quantum speedup!

Exercise 3: Pair programming






```
n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)
@pl.qnode(dev)
def pair_circuit(probe, key):
    """Test whether probe (list[int]) contains a solution to key (list[int])"""
    return pl.probs(wires=n_bits-1)

pl.draw_mpl(pair_circuit)([1,0,0,1],[0,1,1,1]);
print(pair_circuit([0,1,1,1],[0,1,1,1]))
```

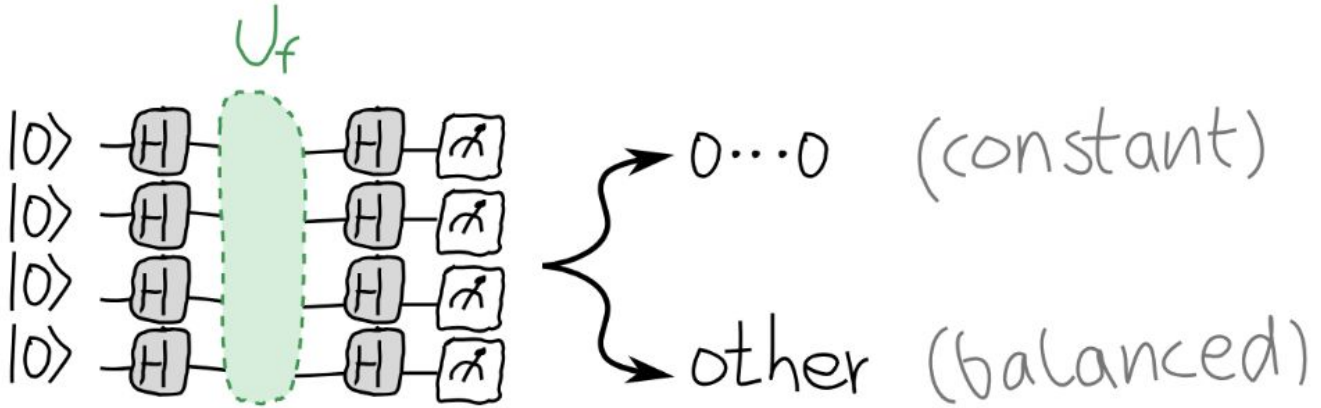
Exercise 3: Pair programming

```
secretkey = [0,1,0,1]

def pair_lock_picker(trials):
   keystrings = [np.binary_repr(n, n_bits-1) for n in range(2**(n_bits-1))]
   keys = [[int(s) for s in keystring] for keystring in keystrings]

   testnumbers = []

   for trial in range(trials):
       counter = 0
       for key in keys:
           counter += 1
           if np.isclose(pair_circuit(key, secretkey)[1], 1):
               break
       testnumbers.append(counter)
   return sum(testnumbers)/trials

trials = 500
output = pair_lock_picker(trials)

print(f"For {n_bits} bits, it takes", output, "pair tests on average.")
```

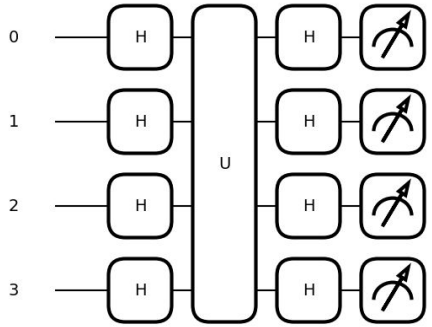
Let's return to the scenario where our lock has multiple secret combinations, with a set of solution strings S and non-solution strings T. Each solution $\mathbf{s} \in S$ contributes -1 in the sum, and each non-solution $\mathbf{t} \in T$ contributes $+1$, so that

$$\mathcal{A}_{\mathbf{0}} = \frac{1}{2^n}(|T| - |S|). \qquad (1)$$

Deutsch and Jozsa noticed something very clever. Call the situation where $|S| = |T|$ "balanced", i.e., there are just as many solutions as non-solutions. Then the amplitude (1) vanishes, and you will never observe $\mathbf{0}$! But if f is "constant", with $S = \varnothing$ or $T = \varnothing$, then $\mathcal{A}_{\mathbf{0}} = \pm 1$ and you will always observe $\mathbf{0}$! This leads to the **Deutsch-Jozsa algorithm**. If we are given the promise that f is either balanced or constant, then applying the same set of gates and observing will tell us which it is. It just depends on whether we see $\mathbf{0}$ or not!



Exercise 4: Deutsch-Jozsa



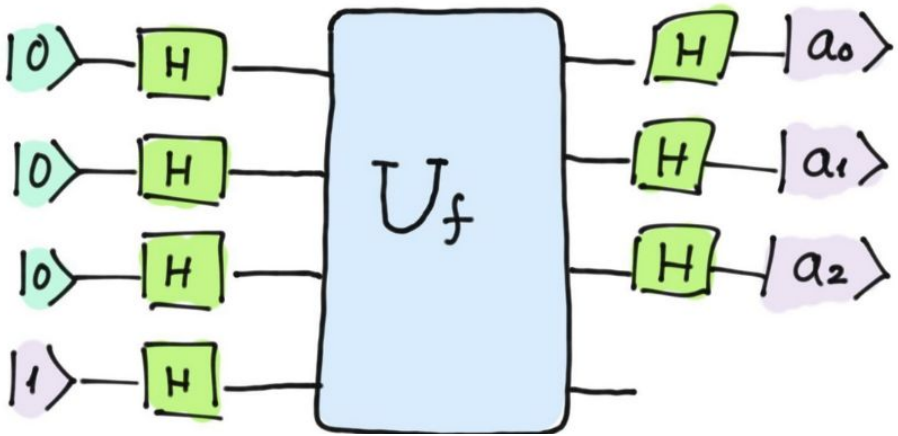

```
def oracle_matrix(keys):
    """Create the unitary matrix corresponding to the binary key (list[int])
       e.g. key=[0,1,1] should give the diagonal matrix [1,1,1,-1,1,1,1,1]"""
    # Hint: use np.ravel_multi_index
    matrix = np.identity(2 ** n_bits)
    return matrix

@pl.qnode(dev)
def deutschjozsa(keys):
    """Build the Deutsch-Jozsa circuit"""
    return pl.probs(wires=range(n_bits))

keys = ([[0,0,0,0],[0,0,0,1],[1,0,0,0],[0,1,0,0],[1,1,0,0],[0,0,1,0],[0,0,1,1],[0,1,1,0]])
if np.isclose(deutschjozsa(keys)[0],0):
    print("balanced")
else:
    print("constant")
```

$$f(\vec{x}) = \sum_{i=0}^{n-1} a_i x_i \pmod 2.$$

In general, U_f sends the state $|\vec{x}\rangle|y\rangle$ to the state $|\vec{x}\rangle|y + \vec{a} \cdot \vec{x} \pmod 2\rangle$.

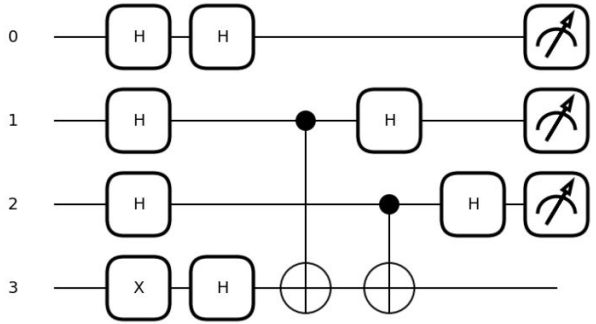
Suppose, for example, that $\vec{a} = [0, 1, 0]$. Then $U_f|111\rangle|0\rangle = |111\rangle|1\rangle$, since we are evaluating f at the point $\vec{x} = [1, 1, 1]$. The scalar product between the two values is 1, so the last qubit of the output will take the value 1.

The Bernstein–Vazirani algorithm makes use of this oracle according to the following circuit:



What we can see is that, by simply using Hadamard gates before and after the oracle, after a single run, the output of the circuit is exactly the hidden value of $\vec{a}$.

Exercise 5: Bernstein-Vazirani




```
dev = pl.device("default.qubit", wires = 4, shots = 1)

def oracle():
   """Encode the hidden value in a circuit"""
   pl.CNOT(wires=[1, 3])
   pl.CNOT(wires=[2 ,3])

@pl.qnode(dev)
def bernsteinvazirani():
   """Sample the Bernstein-Vazirani circuit to return the hidden value"""
   return pl.sample(wires = range(3))

pl.draw_mpl(bernsteinvazirani)()
a = bernsteinvazirani()
print(f"The value of a is {a}")
```