CSCI 4140
Original Three Algorithms

Mark Green

Faculty of Science

Ontario Tech

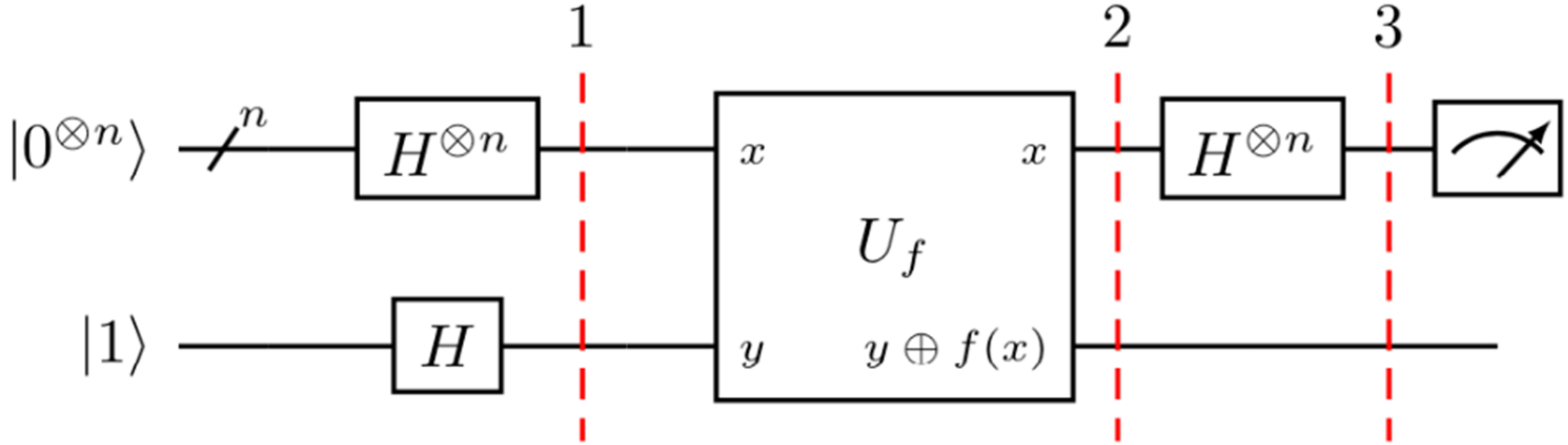
Introduction

- In this lecture we will examine the original three algorithms:
  - Deutsch-Josza Algorithm
  - Bernstein-Vazirani Algorithm
  - Simon’s Algorithm
- Examine how the algorithms are implemented, pick up some implementation strategies and tricks

Deutsch-Josza Algorithm

- This algorithm is given an unknown function, f(x), and needs to determine if its balanced or constant, it must be one of the two
- Our algorithm has n input qubits, x, that are used for f(x) and one input, y, that accumulates the result
- After executing f(x), the output on y is $y \oplus f(x)$
- The algorithm presented in class is shown on the next slide
- If f(x) is constant all the outputs will be zero, if it is balanced the output will be 1

Deutsch-Josza Algorithm



Deutsch-Josza Algorithm

- The first thing we need is circuits for constant and balanced functions,
- We call these oracles
- The one for a constant function is easy, it always produces 0 or 1, we can use a random number generator to determine which one

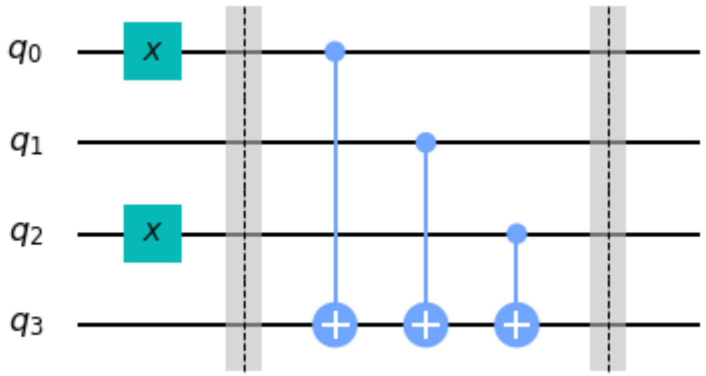
```
# n is the length of the first input register, the other one is length one
n = 3

const_oracle = QuantumCircuit(n+1)

# randomly choose between a zero and 1 output

output = np.random.randint(2)
if output == 1:
    const_oracle.x(n)

const_oracle.draw('mpl')
```

Deutsch-Josza Algorithm

- There are a number of ways to create a balanced function, one of them is shown on the next slide
- This version has a CNOT from each qubit in x to y
- You can try this with various inputs and you will see that it produces a balanced output
- The algorithm itself is fairly easy to code just following the circuit diagram from the lecture
- See the following slide

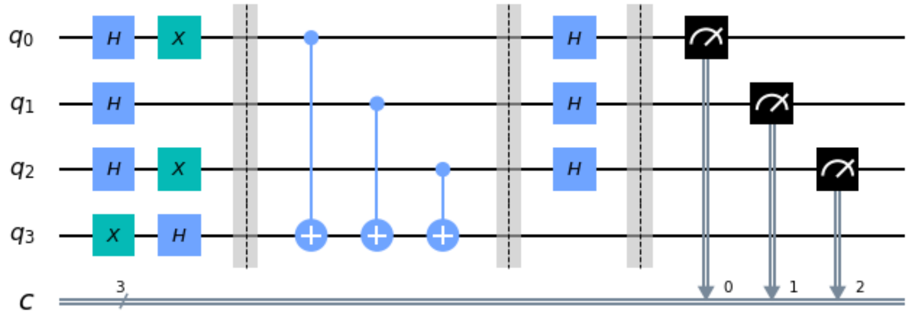
Deutsch-Josza Algorithm

```
balanced_oracle = QuantumCircuit(n+1)

# b_str determines the particular blanced function

b_str = "101"

# Place X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Use barrier as divider
balanced_oracle.barrier()

# Controlled-NOT gates
for qubit in range(n):
    balanced_oracle.cx(qubit, n)

balanced_oracle.barrier()
balanced_oracle.draw('mpl')
```



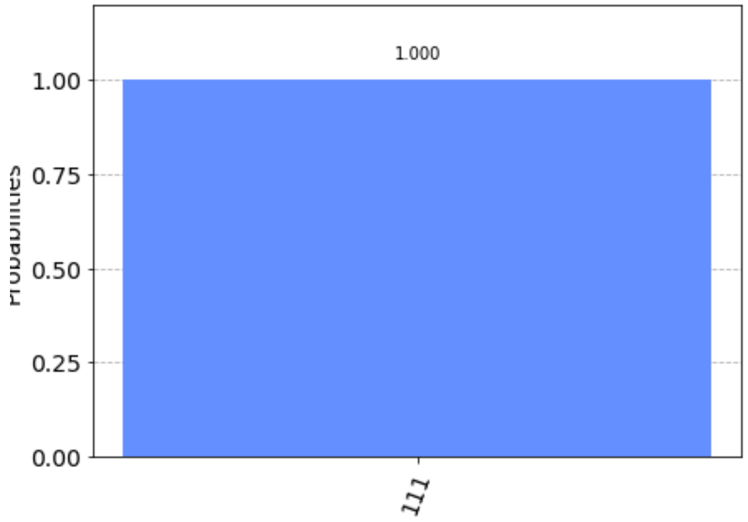
Deutsch-Josza Algorithm

```
dj_circuit = QuantumCircuit(n+1, n)

# First register is in |+> state apply H-gates
for qubit in range(n):
    dj_circuit.h(qubit)

# Second register in state |->
dj_circuit.x(n)
dj_circuit.h(n)

# Add oracle
dj_circuit += balanced_oracle

# Repeat H-gates
for qubit in range(n):
    dj_circuit.h(qubit)
dj_circuit.barrier()

# Measure
for i in range(n):
    dj_circuit.measure(i, i)

# Display circuit
dj_circuit.draw('mpl')
```

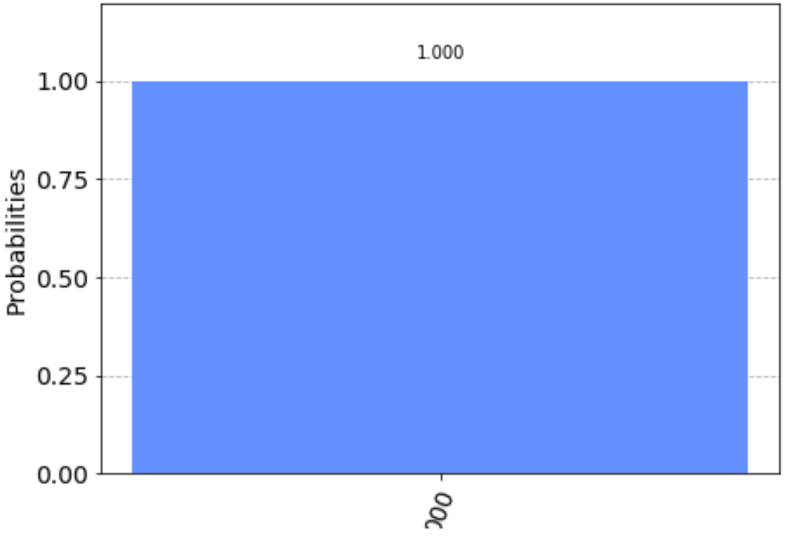


Deutsch-Josza Algorithm

- The big question now is how does it work?
- The results using the qasm simulator



Balanced



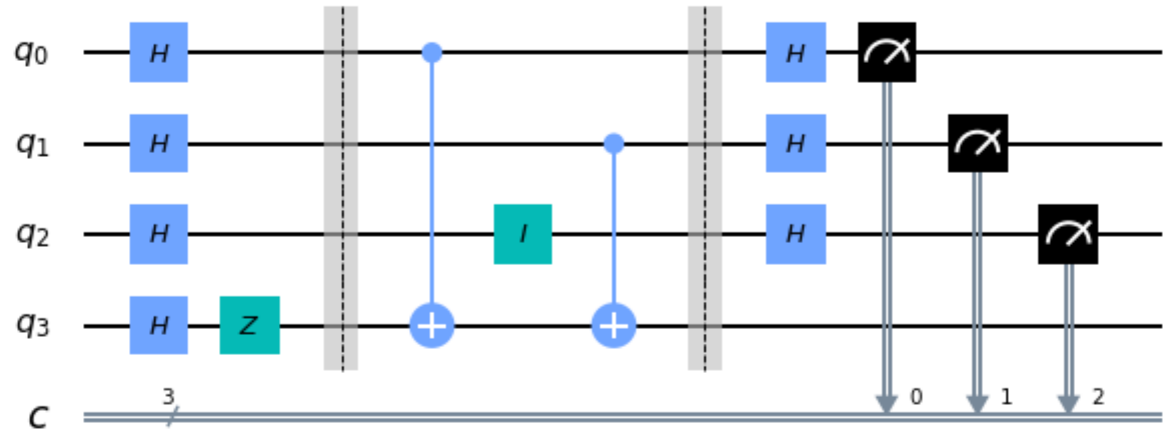
Constant

Bernstein-Vazirani Algorithm

- In this case we have the following function:

$$f(x) = s \cdot x \; mod \; 2$$

- Our problem is to determine the value of s
- Again the algorithm itself is fairly simple, the mathematics behind it isn't so simple



Bernstein-Vazirani Algorithm

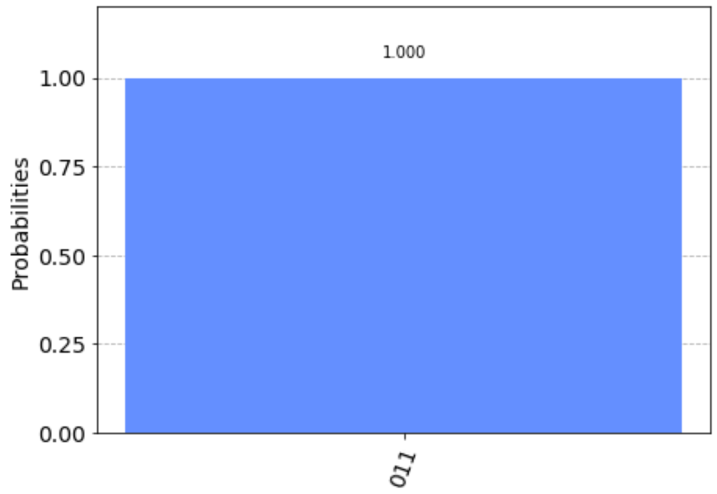
- Again the oracle is the hardest part of the circuit
- In the previous circuit the oracle is the center part of the circuit between the barriers and the value of s is 011
- The I gate is the identity
- To see if it works, you could put it in a circuit, try different input values and measure the result, again it’s a bit mysterious
- The code to produce the circuit is shown on the next slide

Bernstein-Vazirani Algorithm

```
n = 3 # number of qubits used to represent s
s = '011'   # the hidden binary string

# We need a circuit with n qubits, plus one ancilla qubit
# Also need n classical bits to write the output to
bv_circuit = QuantumCircuit(n+1, n)

# put ancilla in state |->
bv_circuit.h(n)
bv_circuit.z(n)

# Apply Hadamard gates before querying the oracle
for i in range(n):
    bv_circuit.h(i)

# Apply barrier
bv_circuit.barrier()

# Apply the inner-product oracle
s = s[::-1] # reverse s to fit qiskit's qubit ordering
for q in range(n):
    if s[q] == '0':
        bv_circuit.i(q)
    else:
        bv_circuit.cx(q, n)

# Apply barrier
bv_circuit.barrier()

#Apply Hadamard gates after querying the oracle
for i in range(n):
    bv_circuit.h(i)

# Measurement
for i in range(n):
    bv_circuit.measure(i, i)

bv_circuit.draw('mpl')
```

Bernstein-Vazirani Algorithm

- Again using the qasm simulator we get the following result:



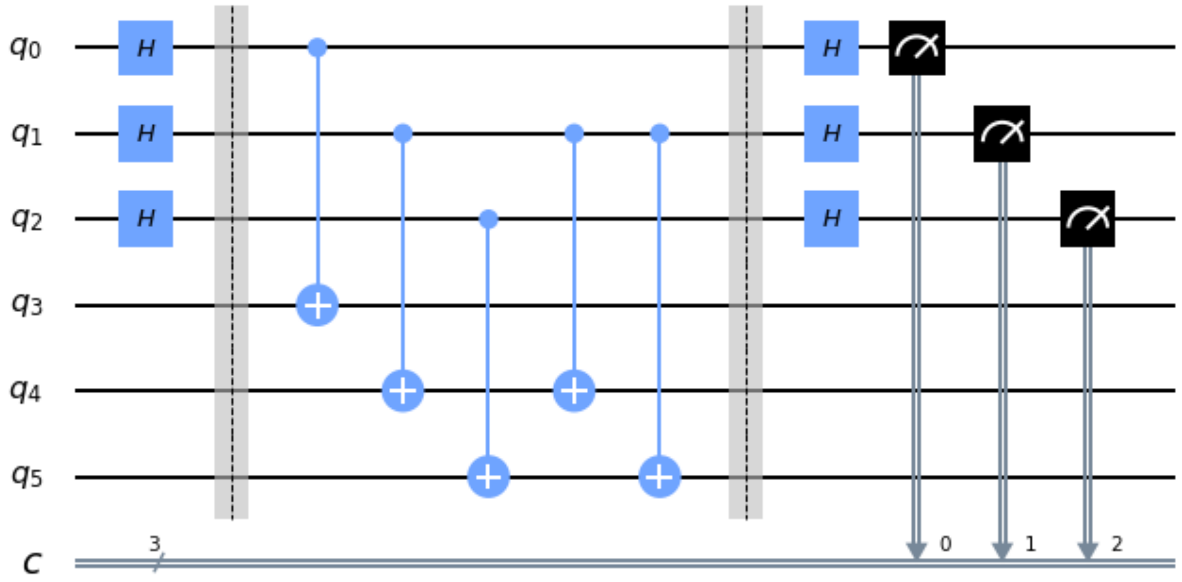
- Which does give us back the value of s that we provided

Simon's Algorithm

- This algorithm is a bit more complicated and we will rely on Qiskit to give us the oracle
- Problem: we are given a function f(x) with n bit input and n bit output
- We want to determine if f(x) is one-to-one or two-to-one
- We are interested in a particular type of two-to-one function that has the following property:
- $f(x_1) = f(x_2)$ if and only if $x_1 \oplus x_2 = b$, where b is called the period of the function
- The problem is really determining the value of b

Simon’s Algorithm

- We have the following quantum algorithm:



- Where the middle part is the oracle provided by Qiskit

Simon’s Algorithm

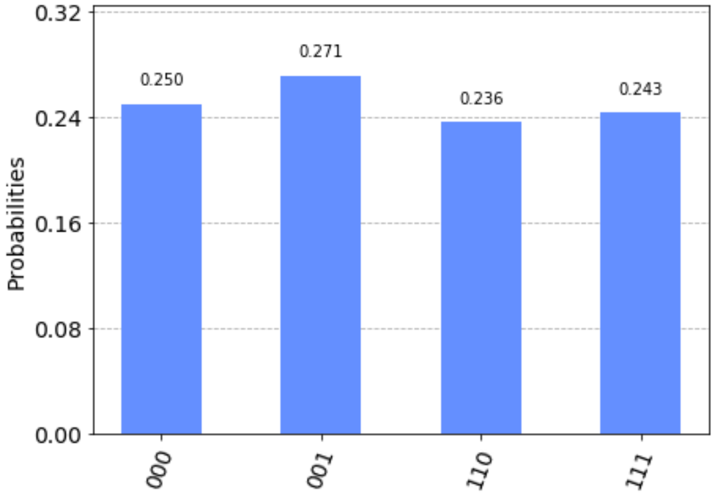
- If we run this algorithm we will get a bit vector that is orthogonal to b
- If we run the algorithm several times we will get several bit vectors that are orthogonal to b
- Eventually we will get enough to form a system of equations that can be solved on a classical computer
- This is a case of where we need both a quantum computer and a classical computer to solve the problem
- The code that produces the circuit is shown on the following slide

Simon’s Algorithm

```python
from qiskit_textbook.tools import simon_oracle

b = '110'

n = len(b)
simon_circuit = QuantumCircuit(n*2, n)

# Apply Hadamard gates before querying the oracle
simon_circuit.h(range(n))

# Apply barrier for visual separation
simon_circuit.barrier()

simon_circuit += simon_oracle(b)

# Apply barrier for visual separation
simon_circuit.barrier()

# Apply Hadamard gates to the input register
simon_circuit.h(range(n))

# Measure qubits
simon_circuit.measure(range(n), range(n))
simon_circuit.draw('mpl')
```

Simon’s Algorithm

- When we run this circuit with qasm we get the following result:



- We can choose any three of these vectors, construct a linear system of equations and solve for b

Summary

- We’ve examined the three algorithms that got quantum computing started
- The circuit and code for these algorithms is quite simple, even if the math behind them isn’t
- This gets us ready for tackling the quantum Fourier transform