Deutsch–Jozsa & Bernstein-Vazirani in pyQuil

In [1]:
```
import numpy as np
import itertools

from pyquil import Program, get_qc
from pyquil.gates import *
from pyquil.quil import DefGate
```

In [2]:
```
def create_Uf(f, n):
    """
    Function to create Ux matrix

    @param f: Input function that we want to encode
    @param n: Number of qubits. This should be 1+(len(input to f))

    @return: Numpy matrix Uf
    """
    dim = 2**n
    # creating a 2^n x 2^n zeros matrix.
    Uf = np.zeros((dim, dim), dtype=int)
    # This creates a list of the different permutations of n bits.
    lst_bitseq = list(map(list, itertools.product([0, 1], repeat=n)))
    for col, bitseq in enumerate(lst_bitseq):
        # applying the operation on the last helper bit.
        last_bit = bitseq[-1] ^ f(bitseq[:-1])
        final_bitseq = [bit for bit in bitseq]
        final_bitseq[-1] = last_bit
        # using the To-Form method discussed in class to help create the
Uf matrix.
        Uf[lst_bitseq.index(final_bitseq), col] = 1
    return Uf
```

In [3]:
```
def create_dj_circuit(f, n):
    """
    This function will create the program.

    @param f: Input function that we want to encode
    @param n: Number of qubits. This should be 1+(len(input to f))

    @return: Pyquil Program
    """
    uf = create_Uf(f, n)
    # Get the Quil definition for the new gate
    uf_definition = DefGate("UF", uf)
    # Get the gate constructor
    UF = uf_definition.get_constructor()

    p = Program()
    p += uf_definition
    p += X(n-1) # Make the last
    for i in range(0, n):
        p += H(i)
    p += Program("UF {}".format(' '.join(str(x) for x in list(range(0, n
)))))
    for i in range(0, n-1):
        p += H(i)
    return p
```

In [25]:
```
def run_circuit(f, n):
    """
    creates and runs a circuit

    @param f: Input function that we want to encode
    @param n: Number of qubits. This should be 1+(len(input to f))

    @return: result
    """
    p = create_dj_circuit(f, n)
    qc = get_qc('Aspen-4-16Q-A')
    result = qc.run_and_measure(p, trials=1)  # We only run 1 trial here
because it is deterministic with no noise
    return result
```

In [26]:
```
def run_dj(f, n):
    """
    Runs Deutsch–Jozsa on f and n

    @param f: Input function that we want to encode
    @param n: n is length of input to the function

    Prints either constant or balanced
    """
    res = run_circuit(f, n+1)
    sum = 0
    for i in range(0, n):
        sum += res[i][0]
    if sum == 0:
        print("Function is constant")
    else:
        print("Function is balanced")
```

In [27]:
```
def run_bv(f, n):
    """
    Runs Bernstein-Vazirani on f and n

    @param f: Input function that we want to encode
    @param n: n is length of input to the function

    Prints the value of a and b of f
    """
    res = run_circuit(f, n+1)
    b = f([0]*(n))
    a = ''
    for i in range(0, n):
        a += str(res[i][0])
    print("Function is defined with a={0}, b={1}".format(a, b))
```

Design

Present the design of how you implemented the black-box function U_f. Assess how visually neat and easy to read it is.

We know that Uf|x>|b> = |x>|b+f(x)> where Uf is the unitary matrix representing the blackbox function f. For n qubits, we first find all the different permutations of the bits in their two states (0,1) and then apply the operation f(x) xor'd with the last qubit (the helper bit) to get a new state for each of those permutations. Using this knowledge of knowing the initial qubit states and the resulting qubit states after applying the operation, we can create the Uf matrix using the "From - To" idea that the professor covered in class. I would it personally give it a 10/10 on how visually neat and easy to read it is.

Present the design for how you prevent the user of U_f from accessing the implementation of U_f. Assess how well you succeeded.

We can use Cython, Nuitka, Shed Skin or something similar to compile python to C code, then distribute the file containing the function as a python binary library (pyd) instead. We didn't apply this method for this homework, but I presume this would be enough to abstract the functionality of Uf creation. However for a determined user, you can't really hide the implementation using Python.

Present the design of how you parameterized the solution in n.

When creating the Uf matrix, we took n as an arguement to the create_Uf() function and accordingly created a unitary matrix $U_f \in \{1,0\}^{2^n \times 2^n}$ and the filled it using the method described in the first answer. For the quantum circuit, we created a string to define the n qubits that Uf would be applied to and passed it as an arguement to our Program object.

Discuss the number of lines and percentage of code that your two programs share. Assess how well you succeeded in reusing code from one program to the next.

We pretty much re-used the entire code for both these programs as the quantum circuit for both of them are exactly the same. All we had to change was the black-box functions for testing our code and some classical post-processing to print the answer

Look at the 4 cases below to see that there is a difference in execution time for different input of Uf

Especially notice that first case is faster than all the other cases

Constant function that always returns 0 on input of length 4

In [28]:
```
%%time
# from pyquil.api._devices import list_devices, list_lattices
# device_names = list_devices()  # Available devices are subject to chan
ge.
# lattice_names = list(list_lattices().keys())  # Available lattices are
subject to change.
# print(f"Available devices: {device_names}.\n")
# print(f"Available lattices: {lattice_names}.")

run_dj((lambda x: 0), 4)
```

```
Function is constant
CPU times: user 49.5 ms, sys: 4.41 ms, total: 53.9 ms
Wall time: 606 ms
```

Constant function that always returns 1 on input of length 4

In [29]:
```
%%time
run_dj((lambda x: 1), 4)
```

```
Function is balanced
CPU times: user 1.07 s, sys: 19.7 ms, total: 1.09 s
Wall time: 8.06 s
```

Balanced function that returns not of the partiy of bits on input of length 4

In [31]:
```
%%time
run_dj((lambda x: sum(x)%2 ^ 1), 4)
```

```
Function is balanced
CPU times: user 2.64 s, sys: 32.5 ms, total: 2.67 s
Wall time: 16 s
```

Balanced function that returns the partiy of bits on input of length 4

In [35]:
```
%%time
run_dj((lambda x: sum(x)%2), 4)
```

```
Function is balanced
CPU times: user 2.4 s, sys: 28.4 ms, total: 2.43 s
Wall time: 15.3 s
```

Now we will look at how long it takes as input size increases

I am only providing results for the case where we always return constant but it is easy to do this for the rest of the cases. Using this test case as it is the only example where I could successfully get result up to size of input 7, in other cases we are seeing a timeout after 4 or 5.

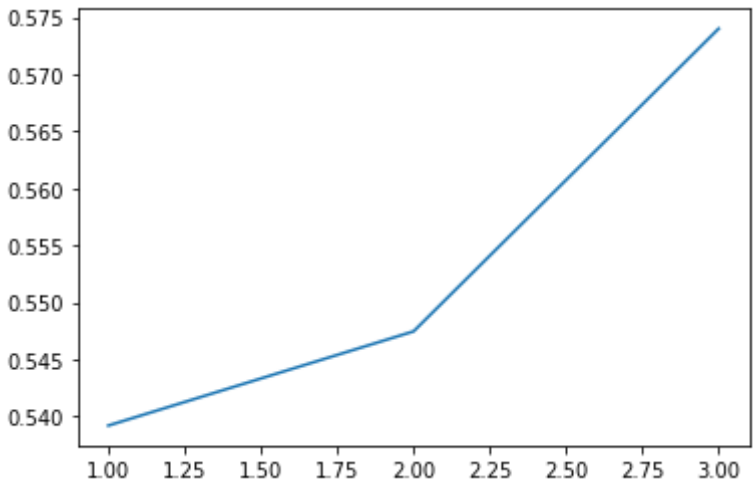
In [45]:
```
%matplotlib inline
import time
import matplotlib.pyplot as plt

time_it_took = []

for i in range(1,4):
    start = time.time()
    run_dj((lambda x: 0), i)
    end = time.time()
    time_it_took.append(end-start)

plt.plot(list(range(1,4)), time_it_took)
plt.show()
```

```
Function is constant
Function is constant
Function is constant
```



Now running some test to show that Bernstein-Vazirani also works

In [43]:
```
%%time
run_bv((lambda x: (np.dot([1, 0], x)%2) ^ 1), 2)
```

```
Function is defined with a=10, b=1
CPU times: user 65.7 ms, sys: 9.92 ms, total: 75.7 ms
Wall time: 751 ms
```

In [39]:
```
%%time
run_bv((lambda x: (np.dot([1, 0, 1, ], x)%2) ^ 0), 3)
```

```
Function is defined with a=101, b=0
CPU times: user 357 ms, sys: 10 ms, total: 367 ms
Wall time: 2.56 s
```

In [42]:
```
%%time
run_bv((lambda x: (np.dot([1, 0, 0, 1], x)%2) ^ 1), 4)
```

```
Function is defined with a=0011, b=1
CPU times: user 2.47 s, sys: 58.1 ms, total: 2.53 s
Wall time: 15.5 s
```

How to use our code?

Running this is straightforward. We have written two functions run_dj and run_bv. The arguments for both functions are the same. The first argument is the name of the function that we are using and the 2nd argument is length of the input to this function. Examples above

In [ ]:
```
```