Detector de Números Primos

Sergio Jiménez Fernández - Curso Programación Software Cuántico

En este cuaderno se desarrolla la implementación del Trabajo de Fin de Curso. Este implica el desarrollo de una herramienta de Detección de Números Primos. Los contenidos de este cuaderno son los siguientes.

- Idea Principal
- Fase 1: Inicialización
- Fase 2: Producto Cartesiano
- Fase 3: Búsqueda
- Conclusión

In [ ]:
```
## NECESSARY IMPORTS ##
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2022-10-04 19:29:48,252: Credentials are already in
use. The existing account in the session will be replaced.
```

Idea Principal

Para afrontar este proyecto, en primer lugar, se sopesaron varias ideas como cuantificar algún ***test de primalidad clásico***, por ejemplo, el Test de Primalidad de Miller-Rabin o el Pequeño teorema de Fermat, o incluso intentar obtener una versión cuántica de algún algoritmo clásico como la _**criba de Eratosthenes**_. Pero, por motivos de complejidad no fueron posibles.

Finalmente, se consiguió un acercamiento mediante una de las aplicaciones del algoritmo de Grover que se muestran en ***este estudio***, basándose en los avances sobre la aplicación del algoritmo Quantum Bit String Comparator (QBSC) de _**David Sena et. al.**_. Proponen un método basado en multiplicar los números impares menores que el número impar a (el que se quiere comprobar si es primo o no). Si de esta multiplicación resulta que a está dentro de unos de los posibles resultados, **no** es primo. Notar que a forzamos que sea impar, porque si fuera par no sería primo pudiendo ser divisible entre 2.

Para ver el algoritmo con más detalle, veamos los tres principales pasos: 1) Inicialización: O lo que ellos conocen como *preparación de la base de datos*. En esta etapa codificaremos los

números impares menores que a en una superposición equiprobable. 2) Producto Cartesiano: Se realiza el producto cartesiano de este estado consigo mismo. Al final, obtendremos un estado con los resultados posibles de este producto cartesiano representado en otra superposición. 3) Búsqueda: Finalmente, solo falta buscar a dentro de la anterior superposición. Si este se encuentra, a **no** será primo. De lo contrario, hemos detectado un número primo.

Para nuestro caso, tenemos un *algoritmo híbrido* en el que las fases 1 y 3 se desarrollaran usando circuitos cuánticos usando la tecnología de IBM Quantum, mientras que la fase 2 se usará un algoritmo clásico para el producto cartesiano utilizando Python como lenguaje de programación.

Antes de pasar a la implementación del algoritmo, cabe destacar que este desarrollo difiere de la propuesta del estudio mencionado anteriormente. Esto se debe a que se plantearon varios errores que obligaron a tomar otras alternativas para cada etapa. Para el desarrollo de este algoritmo, supongamos como ejemplo que $a = 11$.

In [ ]:
```
a = 11
```

Fase 1: Inicialización

Para la inicialización, usaremos la estructura de datos que nos proporciona como salida el ***algoritmo qRAM***, en el que codificaremos la siguiente secuencia

x	$f(x)$
0	1
1	3
2	5
3	7
4	9
5	11
x	$2x+1$

Donde formalmente se conoce dirección a x y dato/valor a $f(x)$, emulando así al comportamiento de la RAM clásica.

Básicamente, para cada número natural, codificaremos el número primo asociado según la relación que vemos en la tabla. De esta manera, si quisieramos comprobar si el 11 es primo, tendíamos que implementar la secuencia de la tabla anterior. Para lo cual, necesitaríamos 3 cúbits para la dirección y 4 para el valor, por tanto, el número de cúbits necesarios es 7 ($n = 7$). Y, como necesitaremos 4 cúbits para al valor, también necesitaremos 4 registros clásicos.

In [ ]:
```
n_add = 3
n_val = 4
n = n_add + n_val
m = n_val
```

Habiendo calculado el número de cúbits necesarios para el algoritmo, podemos proceder a entender el circuito.

La primera dificultad que se nos plantea es la implementación de cada una de las columnas de la qRAM, pues cada una de ellas son en realidad una/varias puertas Pauli-X multicontroladas (ya sea por controles o por anticontroles). Y esto a su vez, podemos dividirlo en dos problemas: i) codificar el valor de la columna de la qRAM; ii) multicontrolarlo en función de la dirección. Ahora, podemos resolver cada subproblema.

Para solucionar el primer punto, tenemos el método `encode_values`, que recibe como entrada la representación binaria del valor que queremos codificar en un circuito a partir de Pauli-X's y devolverá un (sub)circuito. Y para el segundo punto, sencillamente usaremos la función `control` del subcircuito cuántico que acabamos de generar con `encode_values`, indicandole cuantos cúbits de control hay y el mapeo entre los qubits del subcircuito *guest* y el circuito *host*.

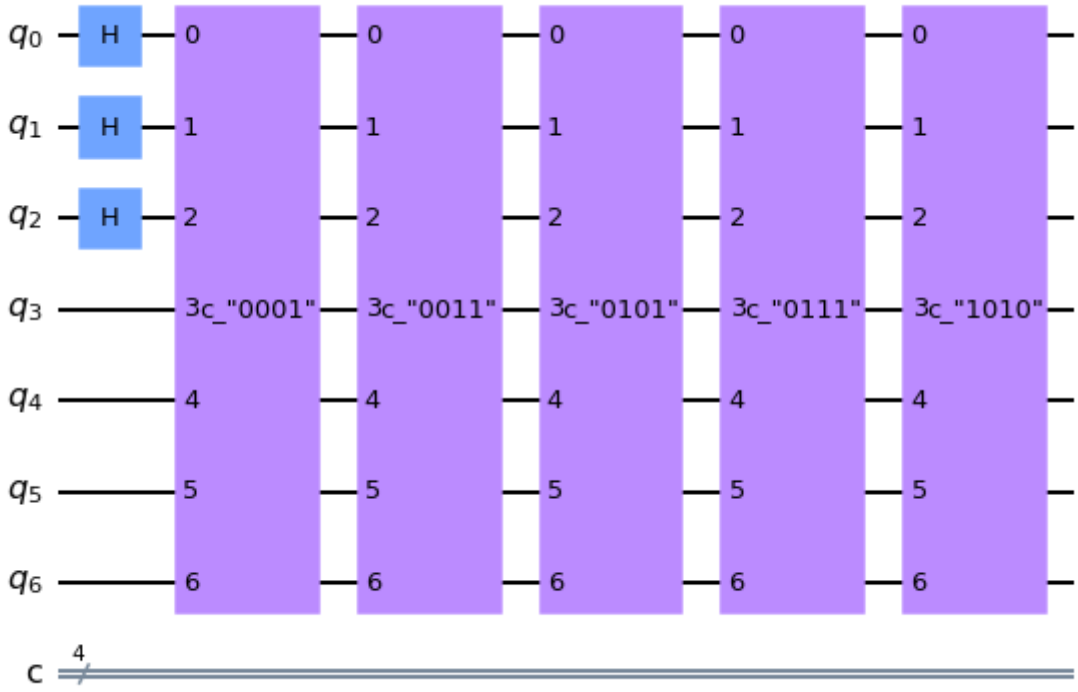
In [ ]:
```
def encode_values(odd_binary):
    '''Returns a quantum circuit with encoding of the odd number by using Pauli-X ga
    control_circuit = QuantumCircuit(n_val, name = f'"{odd_binary}"')

    for i in range(len(odd_binary)):
        if odd_binary[i] == "1":
            control_circuit.x(i)

    return control_circuit
```

Habiendo resuelto este problema, ahora solo queda iterar sobre las diferentes columnas y al final conseguiríamos un estado en el que tenemos representados los números impares menores que a en los cúbits $q3, q4, q5, q6$. En términos más formales:

$$|\gamma\rangle = \frac{1}{\sqrt{5}} \sum_{x=\{1,3,5,7,9\}} |x\rangle = \frac{1}{\sqrt{5}} \big(|0001\rangle + |0011\rangle + |0101\rangle + |0111\rangle + |1001\rangle \big)$$

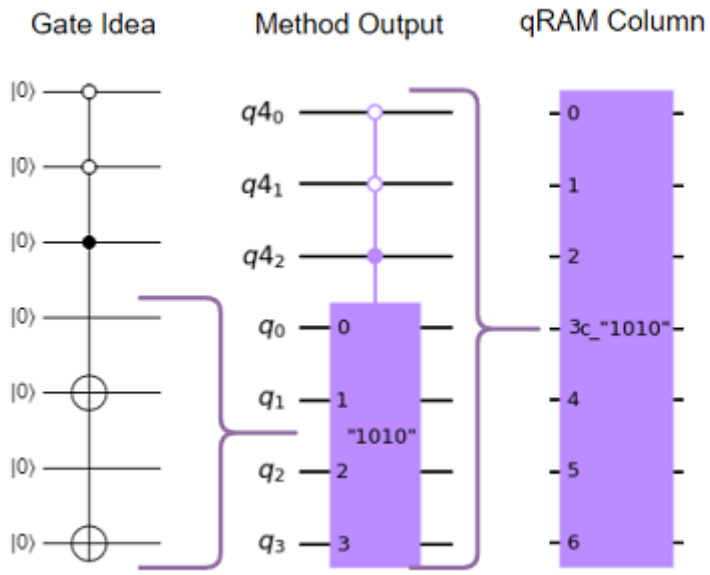
In [ ]:
```
qc = QuantumCircuit(n, m)
qc.h(range(n_add))

data = [1,3,5,7,10]

for i in range(len(data)):
    address_binary = format(i, "b")
    odd_binary = format(data[i], "b")

    while len(address_binary) < n_add:
        address_binary = f'0{address_binary}'

    while len(odd_binary) < n_val:
        odd_binary = f'0{odd_binary}'

    qram_column = encode_values(odd_binary).control(num_ctrl_qubits = n_add, ctrl_st
    qc.append(qram_column, list(range(n)))

qc.draw()
```

Out[ ]:



Apreciación

Como podemos ver en el circuito de la celda anterior, cada columna de la qRAM se representan como cajas negras. Realmente, el proceso de conversión es el que se expresa en la imagen inferior.



Donde vemos la progresión a cada paso de implementación, desde el concepto teórico de una columna, pasando por lo que devuelve el método `encode_values`, hasta lo que realmente se ve representado como salida en la celda anterior".

Fase 2: Producto Cartesiano

Fase 3: Búsqueda

Conclusión

In [ ]: