Quantum Computation & Cryptography
Lab sheet 1

The basics

The basic types we're going to work with are Ket and Qubit (actually Qubits). Why do we have both Ket and Qubits? Here's a helpful analogy: a Graph data structure is something like the figure below, a set of nodes connected by some configuration of edges. However, it is clear that there is a difference between having the Graph of Figure 1 and having the list $[n_1, n_2, n_3, n_4, n_5, n_6]$ of nodes (where n_i is of some type Node).

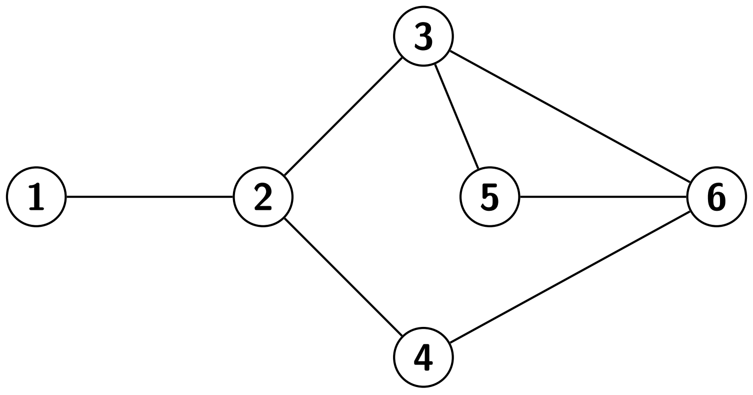


Figure 1: Simple graph

Similarly, you can think of Ket as a grouping of Qubits together with the relations between the qubits (in fact it's even more similar to a graph if you think of the edges as *entanglement* between various qubits). When making a new Ket you will typically specify the number of qubits you want. Here's an example:

```
let k = Ket(5)
let qs = k.Qubits
show "%s" (k.ToString())
```

The variable k is a Ket of 5 qubits. All Kets are initialized to the all 0's state, so k is $|00000\rangle$. The variable qs stores the list of qubits of k.

Gates and measurements act on Qubits types. Some examples of gates are H for Hadamard, X for the Pauli X operation, CNOT for the controlled-NOT operation etc. For measurement, the function is called M. All of these take as input lists of qubits, but only act on either the first or first and second qubits (for

CNOT). If you want to act with the same operation on multiple qubits, use the >< operator. Here's an example:

```
let k = Ket(5)                  // Create a 5-qubit ket (initial state is |00000>)
let qs = k.Qubits               // qs is the list of qubits for the ket k
H qs                            // Hadamard on the first qubit
CNOT qs                         // CNOT on first and second qubits
M >< qs                         // measure all 5 qubits
show "First qubit after measurement %d" qs.[0].Bit.v
show "Second qubit after measurement %d" qs.[1].Bit.v
```

Notice that for a Qubit, q, we get its bit value, after measurement, as $q.Bit.v$. After the qubits have been measured we can no longer apply quantum operations to them, since the qubits have turned into classical bits. We can, however, reanimate the qubits to whatever values we want and then apply operations to them (for example to reanimate a qubit to its collapsed value: $q.ReAnimate(q.Bit)$). We can also reset all the qubits to the original all 0's state by assigning qs to $k.Reset(5)$. Check the Liquid documentation to get more details on these types and on the operations that can be performed.

Time to start coding! We'll start with some simple examples from the source file **Lab1.fsx**.
Important! At the beginning of the source file you will see the line:

```
#r @"/home/andru/Liquid/bin/Liquid1.dll"
```

Change the path to wherever your *Liquid1.dll* file is located. It will be in the *bin* directory of where you unzipped the Liquid project.

Examples

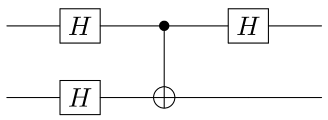
1. Examine the function *ex1*. It only takes a number, n, as input. The function does the following n times: initialize a Ket of one qubit in the state $|0\rangle$, apply a Hadamard operation it, measures it in the computational basis and records the measurement outcome. The measurement statistics are displayed (how many 1's and how many 0's were observed). Test this function with different values of n.

2. Now look at *ex2a*. This function simply applies a $\pi/4$ rotation around the Z axis on the input qubit (using the predefined R operation). Examine the resulting state when this function is applied to $|0\rangle$, $|1\rangle$ and $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Switch to *ex2b*. This function is performing a $\pi/4$ rotation around the X axis (using the predefined rotX operation). Again, examine the resulting state for $|0\rangle$, $|1\rangle$ and $|+\rangle$.

3. The function *ex3* is our first multi-qubit example. Here, the function is taking a list of qubits as input but is only acting on the first two, call them q_1 and q_2. The function applies H to q_1 and then does a $CNOT$ operation with q_1 as the control and q_2 as the target. Test this function with the 4 possible computational basis state inputs (print the resulting ket state but also perform measurements on the two qubits to see what happens). Now test the function in the case where the first input is $|+\rangle$ and the second is is $|0\rangle$. Explain this behaviour.

4. Our next function, *ex4* is sort of a generalization of the previous example. Again we take a list of qubits as input, but this time we will be working with all of them. The function starts by applying H to the first qubit and then proceeds to $CNOT$ the first qubit with each of the remaining qubits. Test this function with varying numbers of input qubits and see what happens.

5. In the last example function, *ex5*, we implement a function from a given circuit. The circuit is the following:



In this example you are also shown (in the *Lab_1()* main function) how to generate a circuit from a "*quantum function*" [1]. Like before, test this function. Print the output as a ket state for various input combinations and also gather measurement statistics and examine the generated circuit.

Warm-up problems

1. Write a function that receives a single qubit, q, as input and applies to it (in this order) a H, X, H. Test this function on inputs $|0\rangle$ and $|1\rangle$ and on superposition states. Does the function behave like a familiar operation you've seen? Now write a similar function which applies the operations H, Z, H and perform the same tests. Is this one familiar?

2. Using your result from the previous problem, implement a controlled-Z operation, CZ for two qubits.

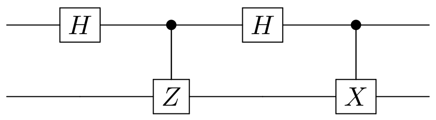
3. Write a function that takes as input a number n and returns an n-qubit state which is the equal superposition of all n-qubit computational basis states (each state in the superposition will have the same amplitude and phase). Example: $n = 3 \rightarrow \frac{1}{2\sqrt{2}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$.

4. Write a function that takes as input an n-qubit state and acts on computational basis states in the following way: if the basis state contains an even number of 1's in its binary representation do nothing, otherwise flip the phase. Example: $|000\rangle \rightarrow |000\rangle$, $|100\rangle \rightarrow -|100\rangle$, $|11\rangle \rightarrow |11\rangle$. Test this function on the superposition created at 3.

Circuits

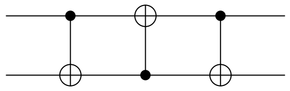
Implement the functions from the given circuits and test their behaviour. Try to figure out what they do. You should also generate the circuits from your functions to check that they correspond to the given circuits.

1.

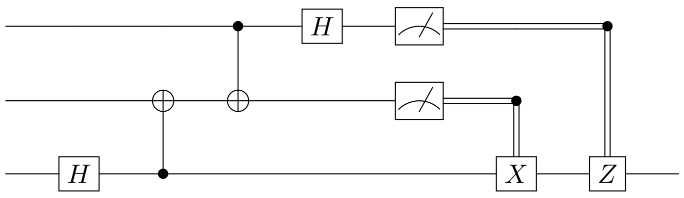


[1]To test this on Linux/Mac, run from command line: *mono Liquid.exe -s "Lab_1.fsx" "Lab_1()"*

2.



3.



Problems

1. Given n qubits, write a function that computes the mod 2 sum (parity) of the first $n-1$ qubits and adds this result mod 2 (the $\oplus$, *xor*, operation) to the n'th qubit. So, for example, if the input state is the computational basis state $|x_1 x_2 x_3 y\rangle$ (where x_1, x_2, x_3 and y are bits) the output state should be $|x_1 x_2 x_3 z\rangle$, where $z = x_1 \oplus x_2 \oplus x_3 \oplus y$. Examples: $|10\rangle \rightarrow |11\rangle$, $|11\rangle \rightarrow |10\rangle$, $|101\rangle \rightarrow |100\rangle$, $|110\rangle \rightarrow |110\rangle$, $|101101010\rangle \rightarrow |101101011\rangle$, $\frac{1}{\sqrt{2}}(|010\rangle + |111\rangle) \rightarrow \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle)$.

2. Write another parity function, like the previous one, but which works for the $(|+\rangle, |-\rangle)$ basis, where $+$ is 0 and $-$ is 1. Examples: $|-+\rangle \rightarrow |--\rangle$, $|-+-\rangle \rightarrow |-++\rangle$, $\frac{1}{\sqrt{2}}(|+++\rangle + |---\rangle) \rightarrow \frac{1}{\sqrt{2}}(|+++\rangle + |--+\rangle)$. Test this function on computational basis states as well! Generate the circuits for both this and the previous problem.

3. Write a function that takes as input an n-qubit state and acts on computational basis states in the following way: if the basis state is $|00...0\rangle$ flip its phase, otherwise do nothing. Test the function on the superposition state from before. You should implement this function explicitly writing the associated unitary as a sparse matrix as in the given *rotX* function.

4. Implement a similar function which flips the phase only for the state $|x\rangle$, where x is given as input (in any representation you like: number, binary representation as list of booleans etc).