Creating Arbitrary Superpositions

Problem Statement

Design a general circuit that accepts vectors with random positive integral values of size n with m bits in length for each element and finds the superposition of indices such that the elements of the vector at those indices have different values for every two adjacent bits in their binary representation.

Example

So, if we were to provide an example — consider the vector $[1, 5, 7, 10]$. Now, I shall describe the computation that would lead us to our solution.

Classical Part:

- First, we convert the elements of the array into their respective binary representation (as required).

$$[1, 5, 7, 10] \rightarrow [0001, 0101, 0111, 1010]$$

- Now, in the list of binary representations we search for those numbers with different values for every adjacent bit pairs. In this case, they are $[0101, 1010]$ that is 5 and 10 respectively.
- Now, we need to obtain their indices in accordance with the original array.

$$\text{indices} = [1, 3]$$

- Finally, we return the binary representation of these indices.

$$[01, 11]$$

Quantum Part:

- Given, the list of indices in their binary representation consider each element of the list to denote a quantum state. Now we have a list of quantum states. Thus, in this case we have $[|01\rangle, |11\rangle]$.
- Create a state that is equal superposition of all such states in the above list. The final superposition obtained in this case is shown below.

$$\frac{1}{\sqrt{2}}(|01\rangle + |11\rangle)$$

Solving for the Classical Part

For the classical part of the problem, given an arbitrary array of elements we have to return a list of suitable indices. This is easy enough.

Algorithm:

Suppose we have an array arr, then we iterate over each of its elements and convert them into their required binary representation and for every such binary representation, we do the following:

- Let s be the string denoting the binary representation of $arr[i]$.
- Then, if $s[j] \neq s[j-1] \; \forall \; j \in [1, ..., \text{len}(s) - 1]$ then we know that we have found a binary representation where every two adjacent bits are different.
- We append the index i to a list, say L.
- Finally after all iterations, we obtain L as the required list of suitable indices.

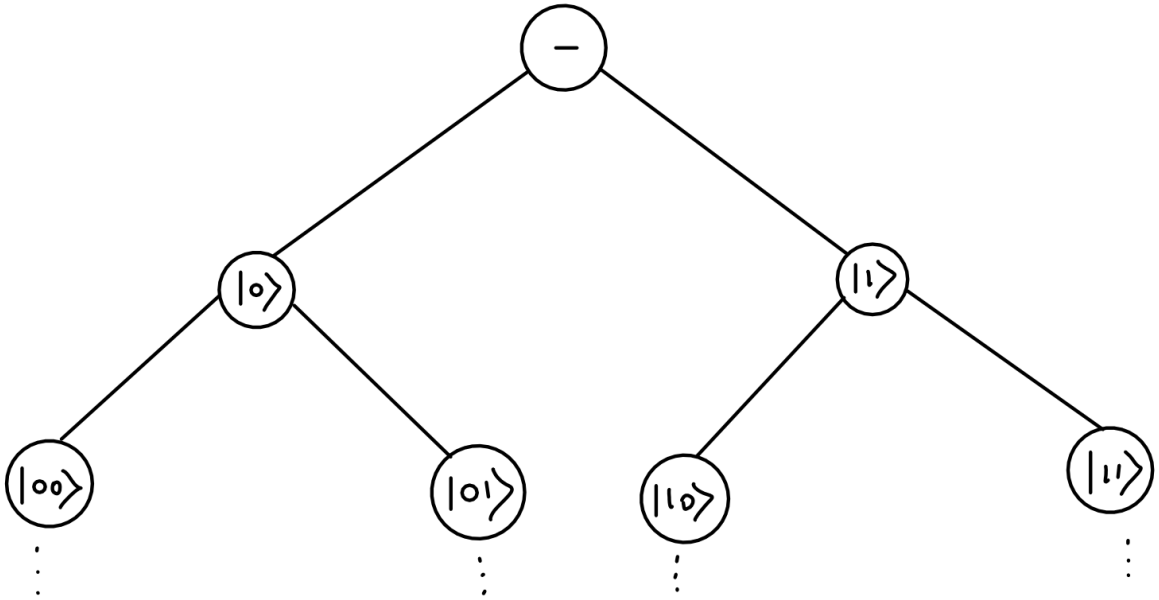
Solving for the Quantum Part

Now, upon obtaining the list of indices, we form an array (called params) of length $2^{\text{binary len}(n-1)}$ where n is the number of elements of arr (the original vector or array of random positive integers).

Now, params will be an array representation of the required superposed state we expect to get. For example, in the example we had shown above with $arr = [1, 5, 7, 10]$, our params array will be as follows.

$$\text{params} = [0, \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}]$$

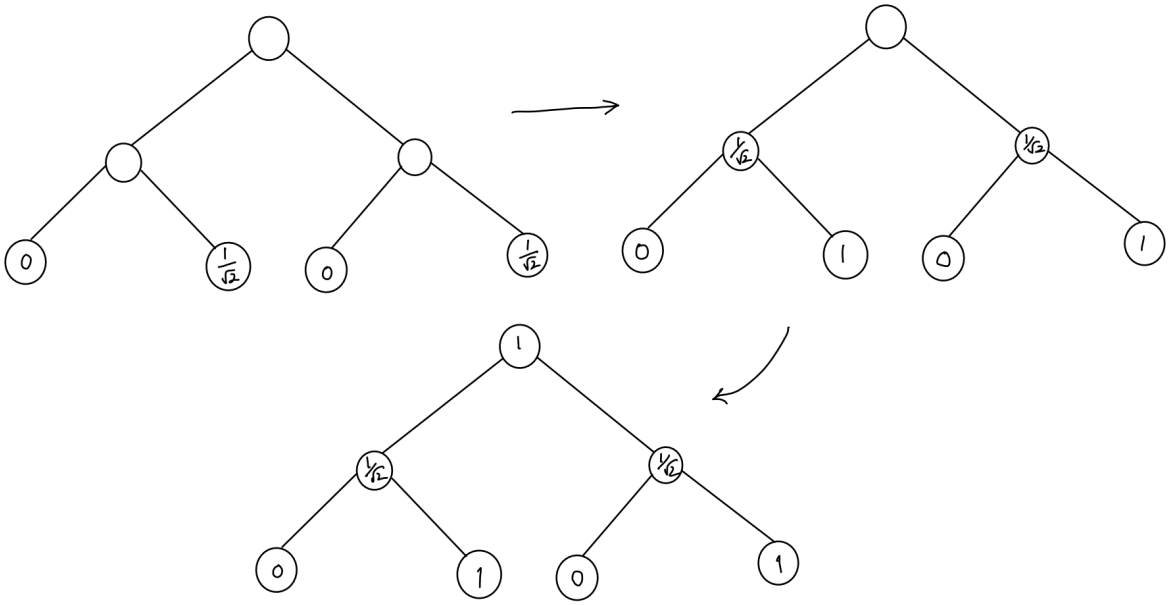
Now using this array we create a binary tree (which I later learned is similar to the bifurcation graph of a QRAM) of the following structure.



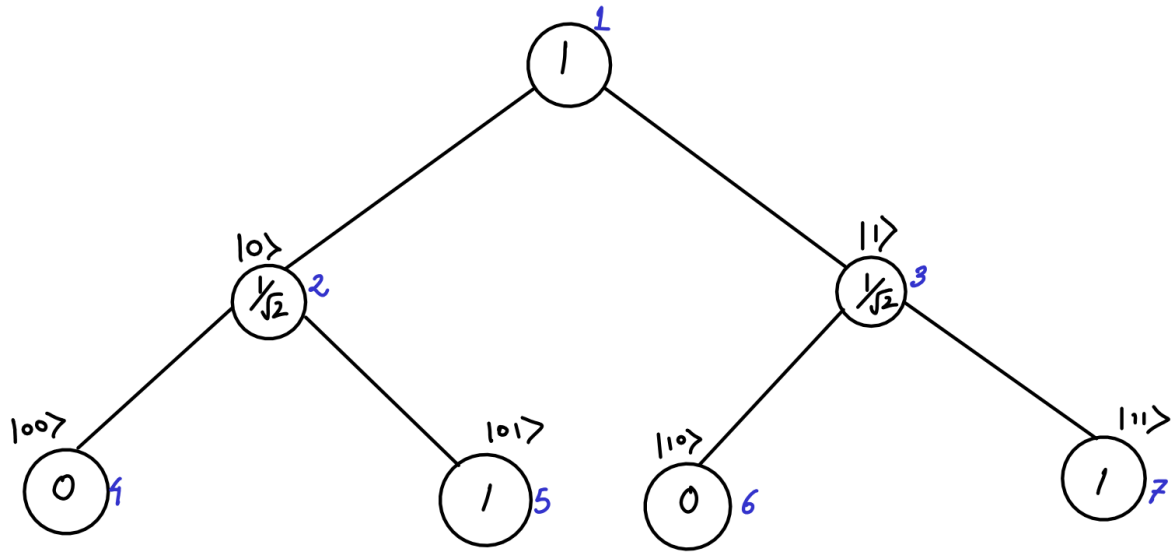
So basically, as we go lower in the tree we can represent larger states. For every parent node $|v\rangle$ we have $|v\rangle \otimes |0\rangle$ and $|v\rangle \otimes |1\rangle$ as its two children nodes.

Now, each node (which represents a state) stores a value (amplitude) k such that there is a probability of k^2 to reach it from its parent node. Thus, we can use this tree structure to represent any arbitrary superposition of its leaves.

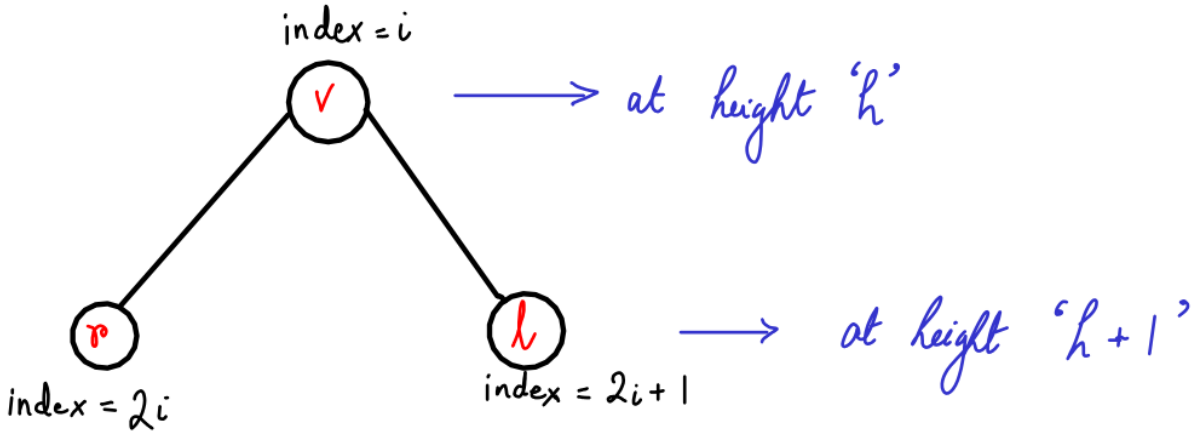
We build this tree bottom up and the amplitudes are shifted from children nodes to parent node, similar to as described below.



So finally we have the following complete tree.



This tree represents the rolled back probability amplitude distributions and tphe structure is used to apply controlled rotations on the qubits to obtain the desired state represented by the overall amplitude distribution in leaf nodes (namely $[0, \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}]$).



Computation at every node

Suppose, we are at index i which falls at height h. Based on the value l, the angle of rotation will be decided. When we are at node i, the rotational gate applied is $R_y(\theta)$ on the $q[h+1]$ (the qubit at height $h+1$).

$$R_y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$$

Here, $\theta = 2\cos^{-1}(l)$. Whether or not $R_y(\theta)$ is applied, depends on the control qubits (all qubits till the height h represented by $q[:h+1]$) and control values (binary representation of i without the most significant bit).

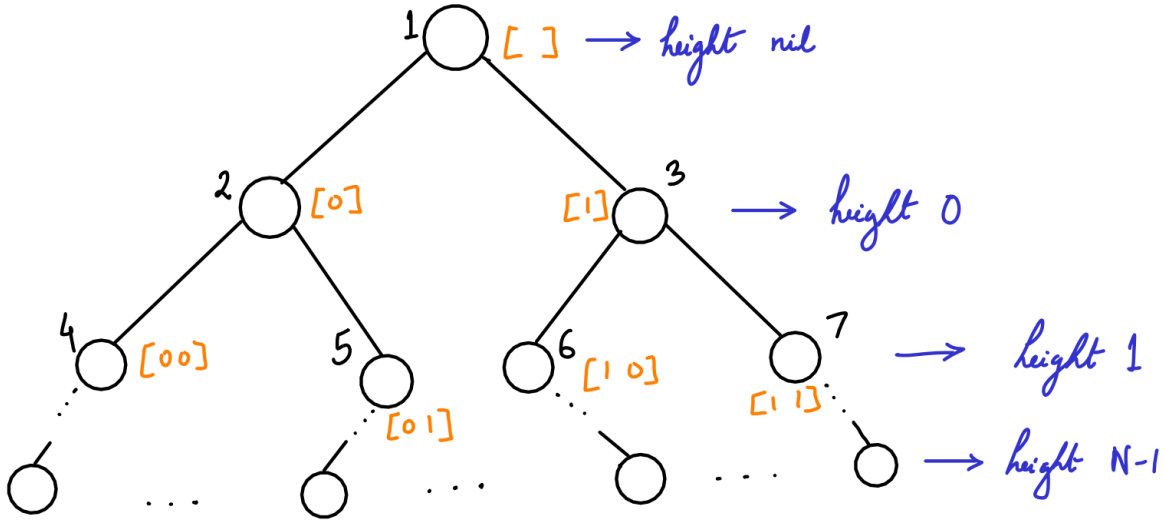
The $R_y(\theta)$ gate is applied only when the values of the list $q[:h+1]$ are same as that of the list of control values.

For example, if $q[:h+1] = [|+\rangle, |0\rangle]$ and control values $= [1, 0]$ then only 50% of times the associated rotation would be applied, if we were to simulate the circuit execution. The below code shows the implementation of the above idea of controlled rotation in cirq.

```
cirq.YPowGate(exponent=angle)
  .on(qubits[height])
  .controlled_by(*qubits[:height], control_values=bits)
```

Conclusion

Controlled rotations like as explained above are applied at every node other than the leaf nodes. Thus, the last controlled rotations are applied on the qubit at the height of the leaves.

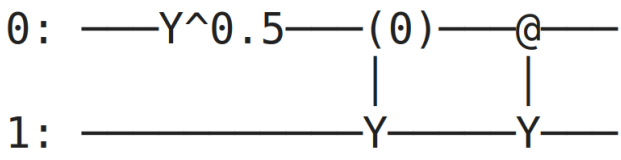


Here, we have a total number of qubits $= N$ and total nodes $= 2^0 + 2^1 + ... + 2^N$. Respective list of control values for computation at every node is shown in orange.

Epilogue

After all operations are performed, the final circuit is obtained. The value of the state vector can be easily tested upon simulating the circuit.

In the example with original vector $[1, 5, 7, 10]$ the circuit obtained is as follows.



Upon simulating the circuit, we obtain the representation of the required state vector (quantum state with required superposition defined according to the problem).

$$[0j,\ (-0.5+0.5j),\ 0j,\ (-0.5+0.5j)]$$

On removing global phase, we get the above quantum state as $[0, \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}]$ which is basically what we wanted.

$$[0, \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}] \equiv \frac{1}{\sqrt{2}}|01\rangle + |11\rangle$$

Moreover, the documentation provided in the code(s) is quite extensive and can be referred to for any further doubts or query regarding the implementation of the above idea.

Additional Study

Creating arbitrary superpositions is a subproblem. It comes under the bigger problem of creating any desired quantum state. It would be fun to provide some light on known practices to solve this problem as well.

Qiskit (by IBMQ) uses a method proposed by Shende et. al. which in short assumes to have started with the desired quantum state and reduces that state to $|00....0\rangle$ using a circuit. As a result, the reverse of this circuit would provide us with the necessary initialization circuit.

TLDR of the Idea

Given any state $|a\rangle$, we can construct $R_y(-\theta)R_Z(-\phi)$ such that we obtain the following.

$$R_y(-\theta)R_Z(-\phi)|a\rangle = re^{i\gamma}|0\rangle$$

Now, when there are more than one qubit, say n qubits, we factorize the state vector

$$\begin{aligned}|\psi\rangle &= k_1|0...00\rangle + k_2|0...01\rangle + k_3|0...10\rangle + \cdots + k_{2^n}|1...11\rangle \\ &= |0...0\rangle(k_1|0\rangle + k_2|1\rangle) + \cdots + |1...11\rangle(k_i|0\rangle + k_{2^n}|1\rangle) \\ &= |0...0\rangle(|a\rangle_1) + \cdots + |1...11\rangle(|a_{2^{n-1}}\rangle)\end{aligned}$$

such that $R_y(-\theta_j)R_Z(-\phi_j)|a\rangle_j = r_j e^{i\gamma_j}|0\rangle$ for a certain index j.

$$U = \begin{bmatrix} R_y(-\theta_1)R_Z(-\phi_1) & & & \\ & R_y(-\theta_2)R_Z(-\phi_2) & & \\ & & \cdots & \\ & & & R_y(-\theta_{2^{n-1}})R_Z(-\phi_{2^{n-1}}) \end{bmatrix}$$

Therefore, the above unitary U can be implemented as a "quantum multiplexor" gate (since it is a block diagonal matrix) such that upon applying it to $|\psi\rangle$ we get the following result.

$$U|\psi\rangle = \begin{bmatrix} r_1 e^{i\gamma_1} \\ ... \\ r_{2^{n-1}} e^{i\gamma_{2^{n-1}}} \end{bmatrix} \otimes |0\rangle$$

In short, this is how `<circuit>.initialize(<desired_vector>, <list of qubits>)` is accurately implemented in Qiskit.