# ASparse State Preparation - Best Case Scenario

## Algorithm_1

The objective of this algorithm is to create a circuit C such that given a sparse state input |x>, we can generate this sparse state by applying C to $|0>^n$. We implement the algorithm proposed by Gleinig and Hoefler in the two parts they designate Algorithm_1 and Algorithm_2.

The implementation for Algorithm_1 is given in the challenge. Algorithm_1, given an input S of quantum states, classically computes a list of operations, ops, which details which gates should be applied to reach our target state. The purpose of each list is as follows:

- ops1
    - List that determines which gates should be used, and their order. ops1[i] has a value of:
        - 1 to denote a NOT gate
        - 2 to denote a CNOT gate
        - 3 to denote RY gates surrounding another gate
        - 4 to denote the final bitstring has been returned
- ops2
    - List that gives the index of the qubit where a NOT gate should be applied
- ops3
    - List that gives the indices of the control and target qubits where a CNOT gate should be applied
- Ops4
    - List that gives the phase of the RY gate, the index of the target qubit, and the indices of the control qubit if the operation is a series of CNOT gates
- ops5
    - Bitstring output of algorithm_1 after all circuit operations have been applied
- n9
    - n9[i] represents the number of times that the gate represented by op1[i] is carried out

However, it differs from the pseudocode presented in the article. Namely, Algorithm_1 recursively calls itself until the basis state reaches 1. This replaces the while loop in Algorithm_2, which simplifies the superpositions of the states until the S reaches 1. After completion of Algorithm_1, we utilize the quantum functions that correspond with the value in ops1.

## Quantum Functions

- unitary_control(qubit: QArray[QBit], contrl: QArray[QBit], target: QParam[int])
    - This function inputs a target quantum state, a control quantum state, and an index of the quantum state of the control X gate to be applied. It then applies the control X gate to the target qubit at the imputed index.
- y_rotation(theta: QParam[float], reg: QArray[Qbit], target: QParam[int])
    - This function rotates a Qbit in array “QArray” at index “target” about the y-axis a given “theta” degrees from its current position.
- my_unitary (q: QArray[QBit], w:QParam[float], target: QParam[int])
    - This function uses the within_apply function block to first compute y_rotation(w, q, target), then applies a Pauli-X gate to the qubit in q at index target, before “uncomputing” y_rotation by implementing y_rotation(-w, q, target). The significance of the “uncompute” is explained later.
    - 
- My_controlled_unitary(q: QArray[QBit], w:QParam[float], ctrl:QArray[QBit], target: QParam[int])
    - Similarly to my_unitary, this function uses the within_apply function block, however instead of applying a Pauli-X gate, it calls a controlled X gate. It first computes y_rotation(w, q, target), then applies unitary_control(q, ctrl, target), before “uncomputing” y_rotation by implementing y_rotation(-w, q, target). The significance of the “uncompute” is explained later.

## Algorithm_2

Since the while loop from the article’s pseudocode is handled in the recursive calls to Algorithm_1, Algorithm_2 mainly consists of parsing the data stored in ops1-5 and n9 to build the quantum circuit to create a sparse state. Parts of Algorithm_2 was also implemented and given with the assignment. Below is the pseudo code that summarizes the part of the function that we implemented.

```
//Add extra not gates to reach $|0>^n$
for state in sparse_states do:
    for index in state[0] do:
        If state[0][index]==1 do:
            ops1 += 1
            ops2 += n-1-index
        end if
    end for
end for
Reverse ops1-5 and n9

for op in ops1 do:
```

```
If op == 1 do:
    Apply NOT gate to qubit ops2[0]
    ops2.pop(0)
end if
Else If op == 2 do:
    Apply CNOT gate with control qubit ops3[0][0] to target qubit ops3[0,1]
    ops3.pop(0)
end if
Else If op == 3 do:
    Theta = arctan(|sqrt(ops4[0][0])| / sqrt(|ops4[0][1]))
    If ops4[0].length == 4 do:
        controls = ops4[0][2]


end if
```

## Roles of Within_Apply and Phase Kickback

Throughout the implementation of Algorithm 2, one of the key things we had to watch out for was phase kickback. Specifically, with the prevalence of CX gate throughout the program, our goal was to reduce the dimensionality by one each time, however, we must be careful about our implementation as for each phase we introduce by rotation around an axis, a phase would appear on the control kickback due to the nature of quantum systems. To prevent this, we add in a 'compute' and an 'uncompute' unit that forms the general operation $UVU^\dagger$such that we can apply an initial operation that assists us in developing the algorithm but can later be undone. In our case, our 'compute' and 'uncompute' operation is the $RY(\theta)$ gate, where we apply $RY(\theta)$, CX or X gate, then $RY(-\theta)$. When visualized on a bloch sphere, it is clear that such operation will always lead the initial state to rotate around the X axis by  radians without accidentally introducing unwanted phase, ensuring the probability of meas$\pi$uring $|0\rangle$ and $|1\rangle$ is flipped. This implementation is seen in our my_unitary and my_control_unitary functions.