**Summary**

**Time Complexity: O(N)** where N is the number of nodes in the binary tree. In the worst case, we might be visiting all the nodes of the binary tree.
**Space Complexity: O(N).** This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed binary tree could be N.
**Problem Pattern**

1. The lowest common ancestor is defined between two nodes p and q as the lowest node in the tree that has both p and q as descendants. Note that we allow a node to be a descendant of itself.
2. We would traverse the tree and once we reach the desired nodes p and q, we can backtrack and find the lowest common ancestor. How would you traverse the tree? Which algorithm is ideal? Consider DFS.
3. How would you know that you have found the lowest common ancestor? Think about how you could use flags to keep track of this.

**Problem Approach**

1. Traverse the tree in a depth first manner. The moment you encounter either of the nodes p or q, return some boolean flag. The flag helps to determine if we found the required nodes in any of the paths.
2. The least common ancestor would then be the node for which both the subtree recursions return a True flag. It can also be the node which itself is one of p or q and for which one of the subtree recursions return a True flag.
3. If the current node itself is one of p or q, we would mark a variable mid as True and continue the search for the other node in the left and right branches. If either of the left or the right branch returns True, this means the other node was found below.
4. If at any point in the traversal, any two of the three flags left, right or mid become True, this means we have found the lowest common ancestor for the nodes p and q.

## Problem Pseudocode

```
boolean recurseTree(TreeNode currentNode, TreeNode p, TreeNode
q) {

        // If reached the end of a branch, return false
        if (currentNode == null)
            return false

        // Left Recursion. If left recursion returns true, set
left = 1 else 0
        int left = this.recurseTree(currentNode.left, p, q) ? 1
: 0

        // Right Recursion
        int right = this.recurseTree(currentNode.right, p, q) ?
1 : 0

        // If the current node is one of p or q
        int mid = (currentNode == p || currentNode == q) ? 1 : 0

        // If any two of the flags left, right or mid become
True
        if (mid + left + right >= 2)
            this.ans = currentNode   // ans is storing the LCA
node
```

```
        // Return true if any one of the three bool values is
True

        return (mid + left + right > 0)

}
```

**Alternate Approaches**

1. Find paths from root to p & q and store them in separate vectors or arrays. Traverse both paths till the values in arrays are same. Return the common element just before the mismatch. Time and space complexity of this solution is O(n), but you will be traversing the tree twice.
2. If we have parent pointers for each node we can traverse back from p and q to get their ancestors. The first common node we get during this traversal would be the LCA node. We can save the parent pointers in a dictionary as we traverse the tree. The time & space complexity of this approach would also be O(n).