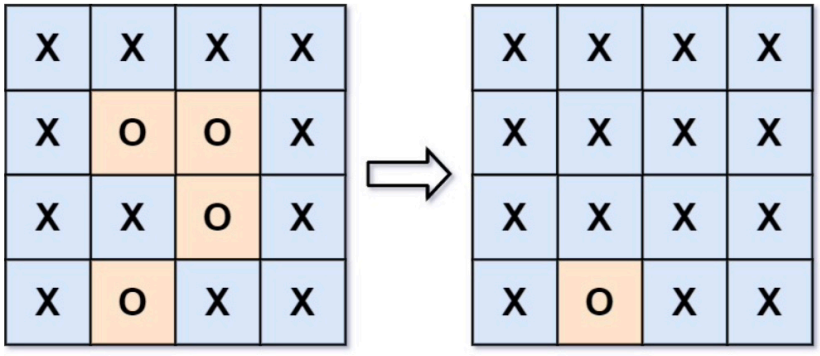
130. Surrounded Regions

Medium 2544 753 Add to List Share

Given an `m x n` matrix `board` containing `'X'` and `'O'`, *capture all regions surrounded by* `'X'`.

A region is **captured** by flipping all `'O'`s into `'X'`s in that surrounded region.

Example 1:




```
Input: board = [["X","X","X","X"],["X","O","O","X"],
["X","X","O","X"],["X","O","X","X"]]
Output: [["X","X","X","X"],["X","X","X","X"],
["X","X","X","X"],["X","O","X","X"]]
Explanation: Surrounded regions should not be on the
border, which means that any 'O' on the border of the board
are not flipped to 'X'. Any 'O' that is not on the border
and it is not connected to an 'O' on the border will be
flipped to 'X'. Two cells are connected if they are
adjacent cells connected horizontally or vertically.
```

Example 2:

```
Input: board = [["X"]]
Output: [["X"]]
```

Constraints:

- `m == board.length`
- `n == board[i].length`
- `1 <= m, n <= 200`
- `board[i][j]` is `'X'` or `'O'`.

```
class UF {
    int count;    // # of subsets
    int[] parent; // some trees
    int[] size;   // weight of trees

    UF(int n);

    // connect p with q
    void union(int p, int q);

    // determine whether p and q connected
    bool connected(int p, int q);

    // find the root for node x
    int find(int x);

    int count();
}
```

Key for Union-Find:

1. Use array parent store parent node for each node.
2. Use array size store weight for each tree. in order to keep balance after union.
3. In find function, compress path to keep height of trees as constant.

1. Union "O" at four sides with dummy.
2. Union the rest of "O"
3. Replace "O"s not connected with dummy

```cpp
class UF {
public:
    // construct function
    UF(int n) {
        this->_count = n;
        // point parent to self
        // init size as 1
        this->parent = new int[n];
        this->size = new int[n];
        for (int i = 0; i < n; i++) {
            this->parent[i] = i;
            this->size[i] = 1;
        }
    }

    // union p and q
    void _union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) {
            return;
        }

        // connect small tree below big tree
        if (this->size[rootP] < this->size[rootQ]) {
            this->parent[rootP] = rootQ;
        } else {
            this->parent[rootQ] = rootP;
        }

        this->parent[rootP] = rootQ;
        this->_count--;
    }

    // determine whether p an q in the same subset
    bool connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // return the number of subset
    int count() {
        return this->_count;
    }

private:
    int _count;
    int *parent;
    int *size;

    int find(int x) {
        while(this->parent[x] != x) {
            this->parent[x] = this->parent[this->parent[x]];
            x = this->parent[x];
        }
        return x;
    }
};
```

```cpp

class Solution {
public:
    void solve(vector<vector<char>>& board) {
        if (board.size() == 0) {
            return;
        }

        int m = board.size();
        int n = board[0].size();

        UF uf = UF(m * n + 1);
        int dummy = m * n;

        // connect O in first column and last column with dummy
        for (int i = 0; i < n; i++) {
            if (board[0][i] == 'O') {
                uf._union(dummy, 0 * n + i);
            }
            if (board[m - 1][i] == 'O') {
                uf._union(dummy, (m - 1) * n + i);
            }
        }

        // connect 0 in first row and last row with dummy
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O') {
                uf._union(dummy, i * n + 0);
            }
            if (board[i][n - 1] == 'O') {
                uf._union(dummy, i * n + n - 1);
            }
        }

        // use array d to search
        int d[4][2] = {{1, 0}, {0, 1}, {0, -1}, {-1, 0}};
        for (int i = 1; i < m - 1; i++) {
            for (int j = 1; j < n - 1; j++) {
                if (board[i][j] == 'O') {
                    for (int k = 0; k < 4; k++) {
                        int x = i + d[k][0];
                        int y = j + d[k][1];
                        if (board[x][y] == 'O') {
                            uf._union(i * n + j, x * n + y);
                        }
                    }
                }
            }
        }

        // replace O not union with df
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 'O' && !uf.connected(dummy, i * n + j)) {
                    board[i][j] = 'X';
                }
            }
        }
    }
};

```