棋盘替换

```
给定一个二维数组，代表一个棋盘将四周被`X`包围的`O`替换成`X`。
```

思路：

1. 将所有违背`X`包围的`O`放入一个`union find`数组中，相连；
2. 此时所有被包围的`O`，必不在`union find`数组中。

二维数组转一维数组的方法：对坐标`(x,y)`转换到一维数组为`x*n+y`,其中二维数组的行数为`m`,列数为`n`.

```
class UF {
public:
  UF(int n) {
    _weight = std::vector<int>(n, 1);
    _parent = std::vector<int>(n, 0);
    _count  = n;

    for (int i = 0; i < n; i++) {
      _parent[i] = i;
    }
  }

  void connect(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);

    if (rootP == rootQ) {
      return;
    }

    if (_weight[rootP] >= _weight[rootQ]) {
      _parent[rootQ] = rootP;
      _weight[rootP] += _weight[rootQ];
    } else {
      _parent[rootP] = rootQ;
      _weight[rootQ] += _weight[rootP];
    }

    _count--;
  }

  bool isConnect(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);

    return rootP == rootQ;
  }
  int find(int x) {
    while (x != _parent[x]) {
      // 路径压缩
      _parent[x] = _parent[_parent[x]];
      x          = _parent[x];
    }

    return x;
  }

  int count() const {
    return _count;
  }

private:
  int              _count;
  std::vector<int> _parent;
  std::vector<int> _weight;
};

class Solution {
public:
  void solve(std::vector<std::vector<char>>& board) {
    int m = board.size(), n = 0;
    if (m > 0) {
      n = board[0].size();
    }
    UF  uf(m * n + 1);
    int dummy = m * n;

    // 连接首列与尾列的`O`
    for (int i = 0; i < m; i++) {
      if ('O' == board[i][0]) {
        uf.connect(i * n, dummy);
      }

      if ('O' == board[i][n - 1]) {
        uf.connect(i * n + n - 1, dummy);
      }
    }

    // 连接首行与尾行的'O'
    for (int i = 0; i < n; i++) {
      if ('O' == board[0][i]) {
        uf.connect(i, dummy);
      }

      if ('O' == board[m - 1][i]) {
        uf.connect((m - 1) * n + i, dummy);
      }
    }

    // 方向数组上下左右
    std::vector<std::vector<int>> dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
    for (int i = 1; i < m - 1; i++) {
      for (int j = 1; j < n - 1; j++) {
        if ('O' == board[i][j]) {
          for (int k = 0; k < 4; k++) {
            int dx = i + dir[k][0];
            int dy = j + dir[k][1];

            if ('O' == board[dx][dy]) {
              uf.connect(i * n + j, dx * n + dy);
            }
          }
        }
      }
    }

    // 所有与dummy不相连的'O'变为'X'
    for (int i = 1; i < m - 1; i++) {
      for (int j = 1; j < n - 1; j++) {
        if (!uf.isConnect(dummy, i * n + j)) {
          board[i][j] = 'X';
        }
      }
    }
  }
};
```

`fill flood`解法：

```
class Solution {
public:
  void solve(std::vector<std::vector<char>>& board) {
    int m = board.size(), n = 0;
    if (m > 0) {
      n = board[0].size();
    }
    // 替换首尾行
    for (int i = 0; i < n; i++) {
      if ('O' == board[0][i]) {
        fill(board, m, n, 0, i);
      }

      if ('O' == board[m - 1][i]) {
        fill(board, m, n, m - 1, i);
      }
    }

    // 替换首尾列
    for (int i = 0; i < m; i++) {
      if ('O' == board[i][0]) {
        fill(board, m, n, i, 0);
      }

      if ('O' == board[i][n - 1]) {
        fill(board, m, n, i, n - 1);
      }
    }

    // 余下的'O'变为'X'
    for (int i = 1; i < m - 1; i++) {
      for (int j = 1; j < n - 1; j++) {
        if ('O' == board[i][j]) {
          board[i][j] = 'X';
        }
      }
    }

    // 余下的'#'变为'O'
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        if ('#' == board[i][j]) {
          board[i][j] = 'O';
        }
      }
    }
  }

private:
  void fill(std::vector<std::vector<char>>& board,
            int                             row,
            int                             col,
            int                             x,
            int                             y) {
    board[x][y] = '#';
    for (int i = 0; i < 4; i++) {
      int dx = x + dir[i][0];
      int dy = y + dir[i][1];
      if (isInArea(row, col, dx, dy) && 'O' == board[dx][dy]) {
        fill(board, row, col, dx, dy);
      }
    }
  }

  bool isInArea(int row, int col, int x, int y) {
    return (x < row) && (x >= 0) && (y < col) && (y >= 0);
  }

  std::vector<std::vector<int>> dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
};
```

判断合法算式

> 给你一个数组equations，装着若干字符串表示的算式。每个算式equations[i]长度都是 4，而且只有这两种情况：a==b或者a!=b，其中a,b可以是任意小写字母。你写一个算法，如果equations中所有算式都不会互相冲突，返回 true，否则返回 false。
> 比如说，输入["a==b","b!=c","c==a"]，算法返回 false，因为这三个算式不可能同时正确。
> 再比如，输入["c==c","b==d","x!=z"]，算法返回 true，因为这三个算式并不会造成逻辑冲突。

```
class Solution {
public:
  bool equationsPossible(std::vector<std::string>& equations) {
    UF  uf(26);
    int row = equations.size();

    for (int i = 0; i < row; i++) {
      char ch = equations[i][1];
      if (ch == '=') {
        uf.connect(equations[i][0] - 'a', equations[i][3] - 'a');
      }
    }

    for (int i = 0; i < row; i++) {
      char ch = equations[i][1];
      if (ch == '!') {
        if (uf.isConnect(equations[i][0] - 'a', equations[i][3] - 'a')) {
          return false;
        }
      }
    }

    return true;
  }

private:
  class UF {
  public:
    UF(int n) {
      _weight = std::vector<int>(n, 1);
      _parent = std::vector<int>(n, 0);
      for (int i = 0; i < n; i++) {
        _parent[i] = i;
      }

      _count = n;
    }

    int find(int p) {
      while (p != _parent[p]) {
        _parent[p] = _parent[_parent[p]];
        p          = _parent[p];
      }
      return p;
    }

    void connect(int p, int q) {
      int rootP = find(p);
      int rootQ = find(q);

      if (rootQ == rootP) {
        return;
      }

      if (_weight[rootP] >= _weight[rootQ]) {
        _parent[rootQ] = rootP;
        _weight[rootP] += _weight[rootQ];
      } else {
        _parent[rootP] = rootQ;
        _weight[rootQ] += _weight[rootP];
      }
      _count--;
    }

    int count() const {
      return _count;
    }

    bool isConnect(int p, int q) {
      int rootP = find(p);
      int rootQ = find(q);

      return rootP == rootQ;
    }

  private:
    std::vector<int> _weight;
    std::vector<int> _parent;
    int              _count;
  };
};
```