Backtracking

Jie He

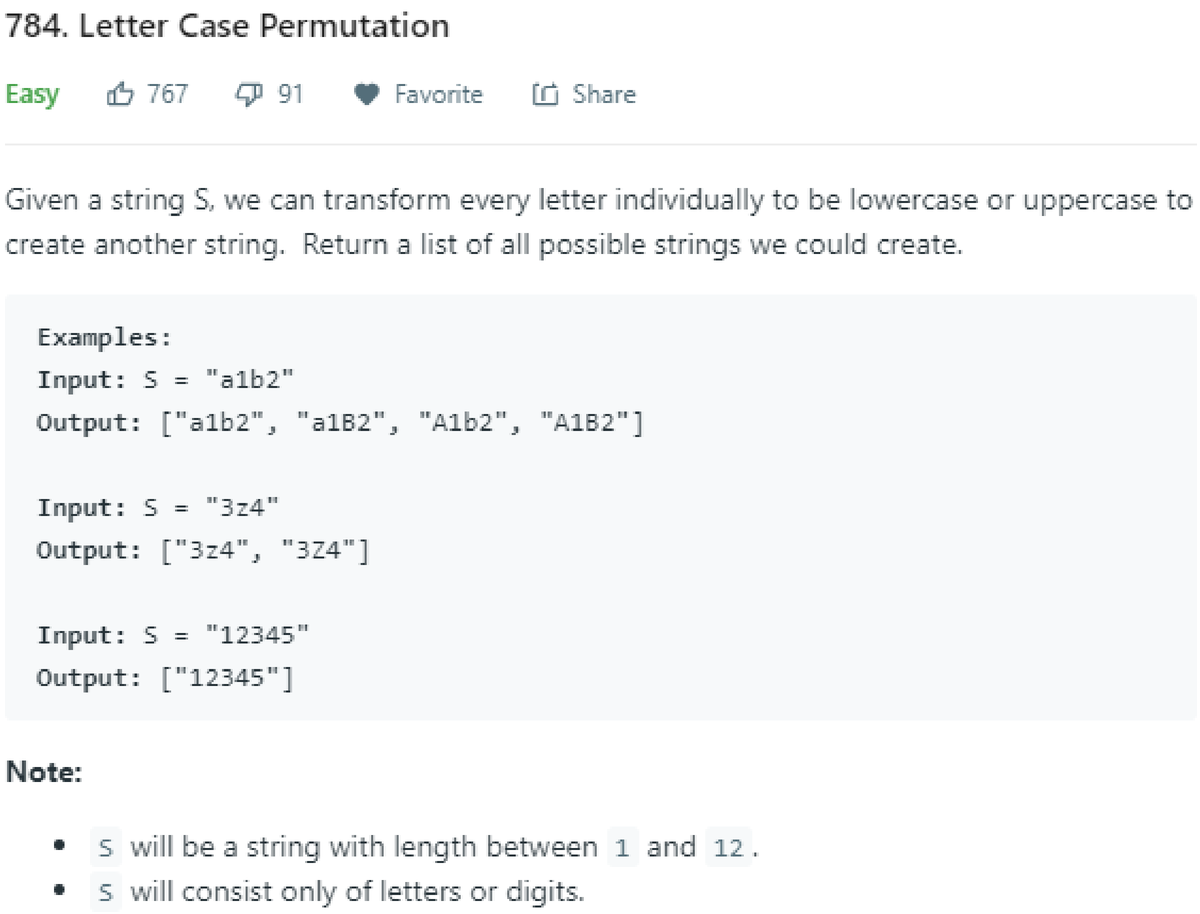
From GeeksforGeeks.org

- Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time
    - Recursion/Iteration of N-step problem
    - DFS of solution tree graph
    - Trim by constraints

Basic Steps

- (Preprocessing)
- Start at Step 0
- At Step N
  - Output the constructed solution
- At Step *i* < *N*:
  - Iterate through each available choice
    - Add the choice to the solution under construction
    - (Label the choice as used/visited)
    - **Go to Step *i*+1**
    - Remove the added choice
    - (Label the choice as available)

- Check/Trim/Skip by
  - Order
  - Repetition
  - Constraints



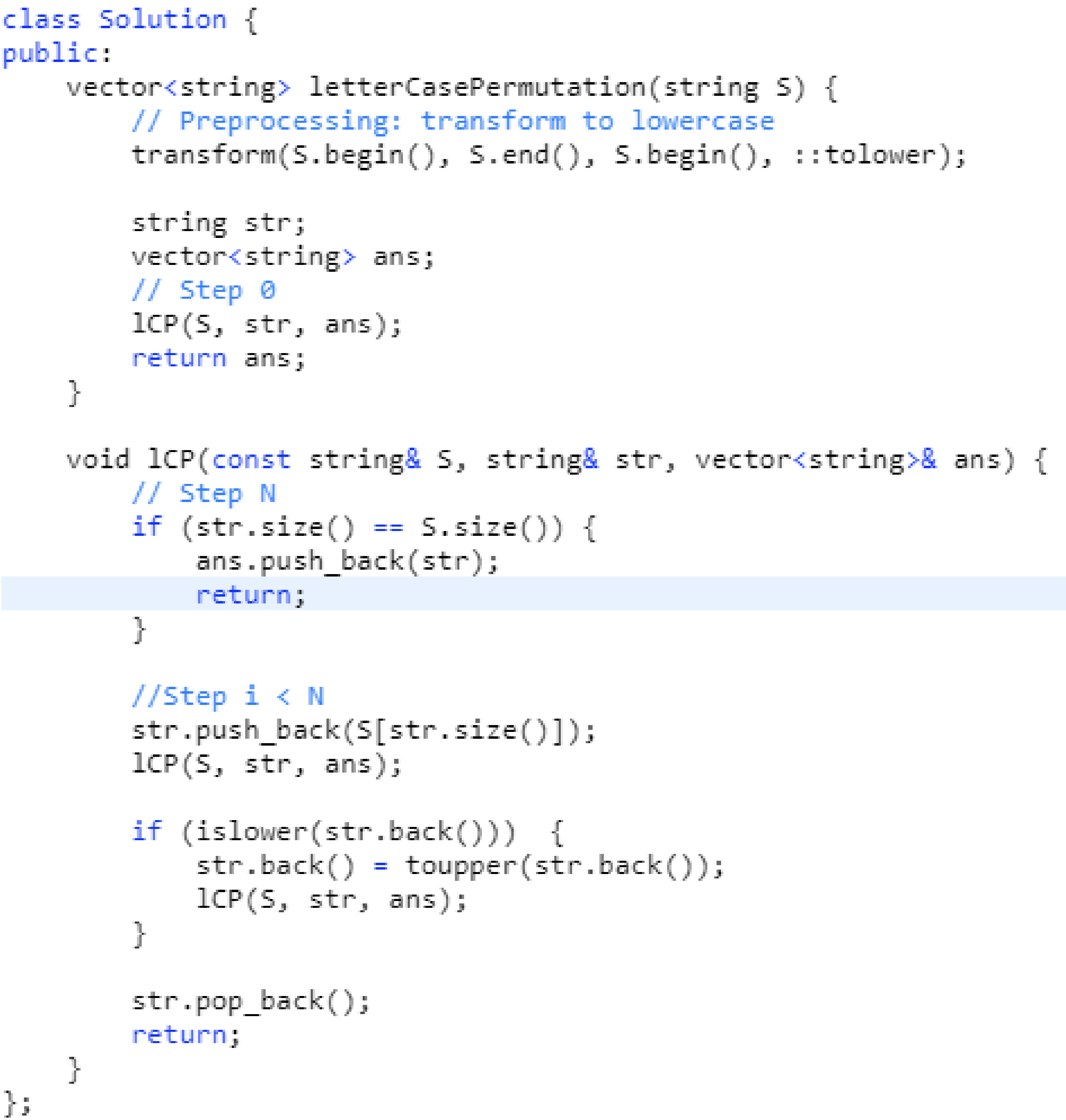
784. Letter Case Permutation

Given a string S, we can transform every letter individually to be lowercase or uppercase to create another string. Return a list of all possible strings we could create.

```
Examples:
Input: S = "a1b2"
Output: ["a1b2", "a1B2", "A1b2", "A1B2"]

Input: S = "3z4"
Output: ["3z4", "3Z4"]

Input: S = "12345"
Output: ["12345"]
```

Note:

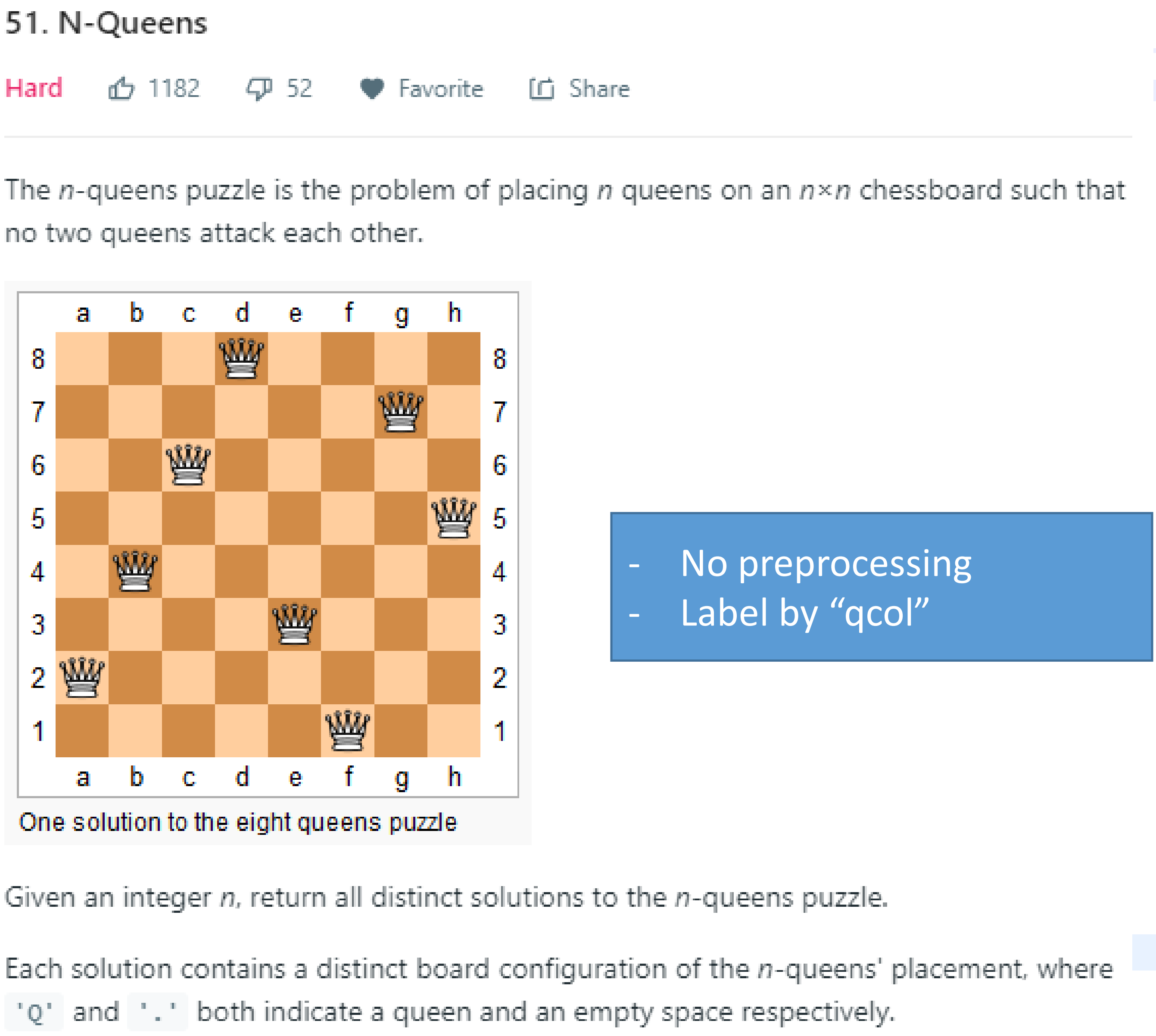
- `S` will be a string with length between `1` and `12`.
- `S` will consist only of letters or digits.



```cpp
class Solution {
public:
    vector<string> letterCasePermutation(string S) {
        // Preprocessing: transform to lowercase
        transform(S.begin(), S.end(), S.begin(), ::tolower);

        string str;
        vector<string> ans;
        // Step 0
        lCP(S, str, ans);
        return ans;
    }

    void lCP(const string& S, string& str, vector<string>& ans) {
        // Step N
        if (str.size() == S.size()) {
            ans.push_back(str);
            return;
        }

        //Step i < N
        str.push_back(S[str.size()]);
        lCP(S, str, ans);

        if (islower(str.back()))  {
            str.back() = toupper(str.back());
            lCP(S, str, ans);
        }

        str.pop_back();
        return;
    }
};
```

- Explicit iteration of choices
- No labels

Can you solve it iteratively?



51. N-Queens

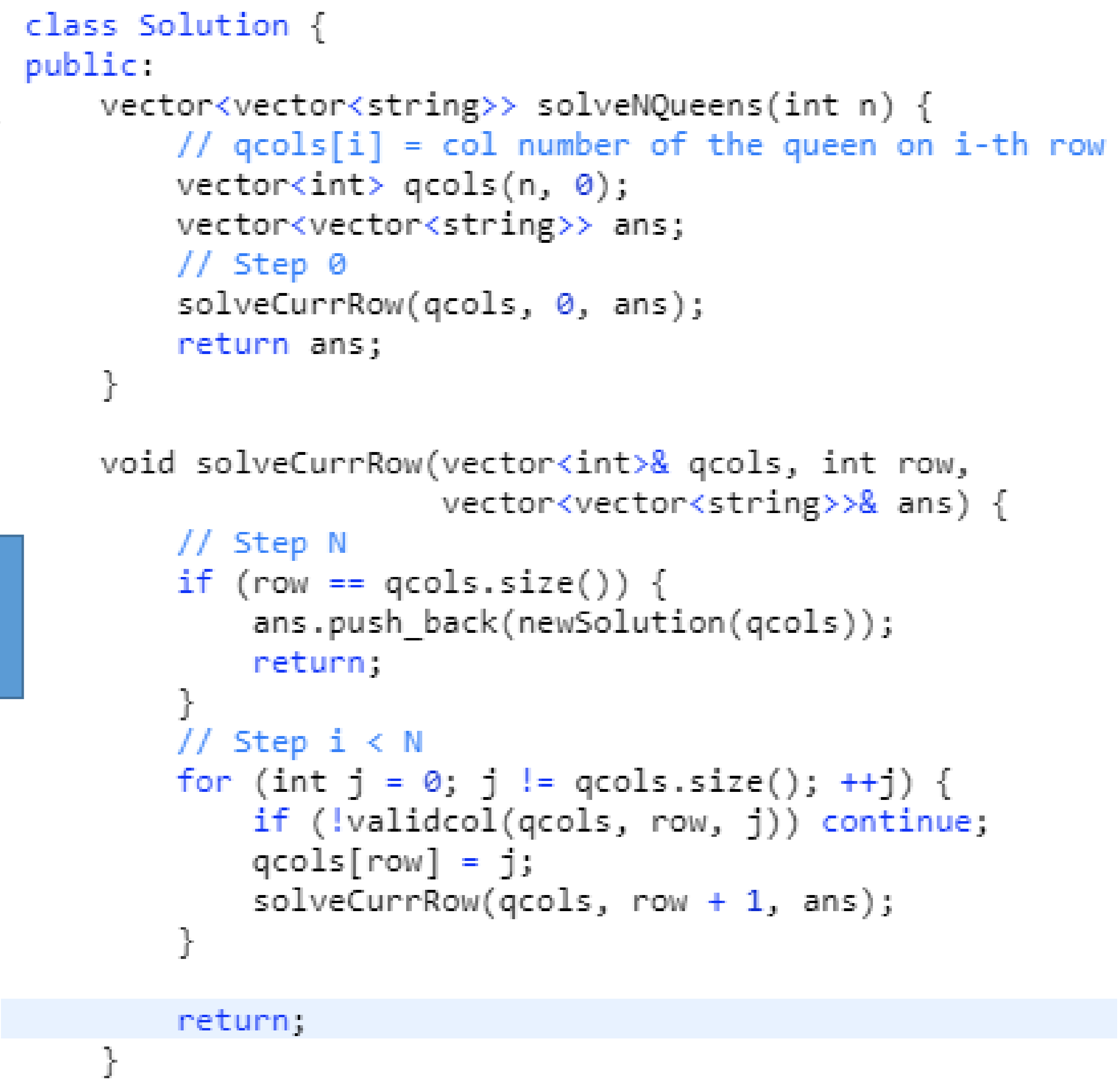
The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.

One solution to the eight queens puzzle

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

Each solution contains a distinct board configuration of the *n*-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

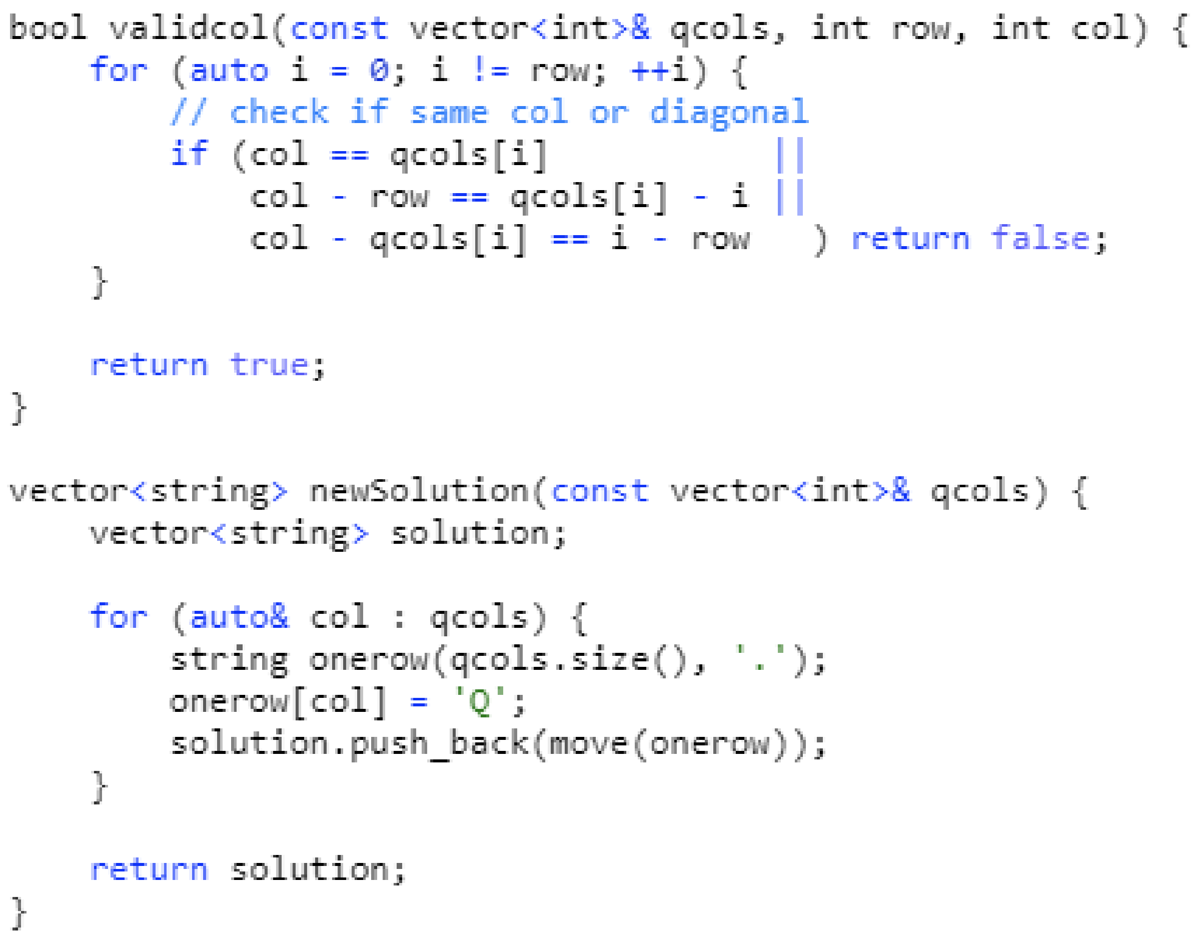
- No preprocessing
- Label by “qcol”



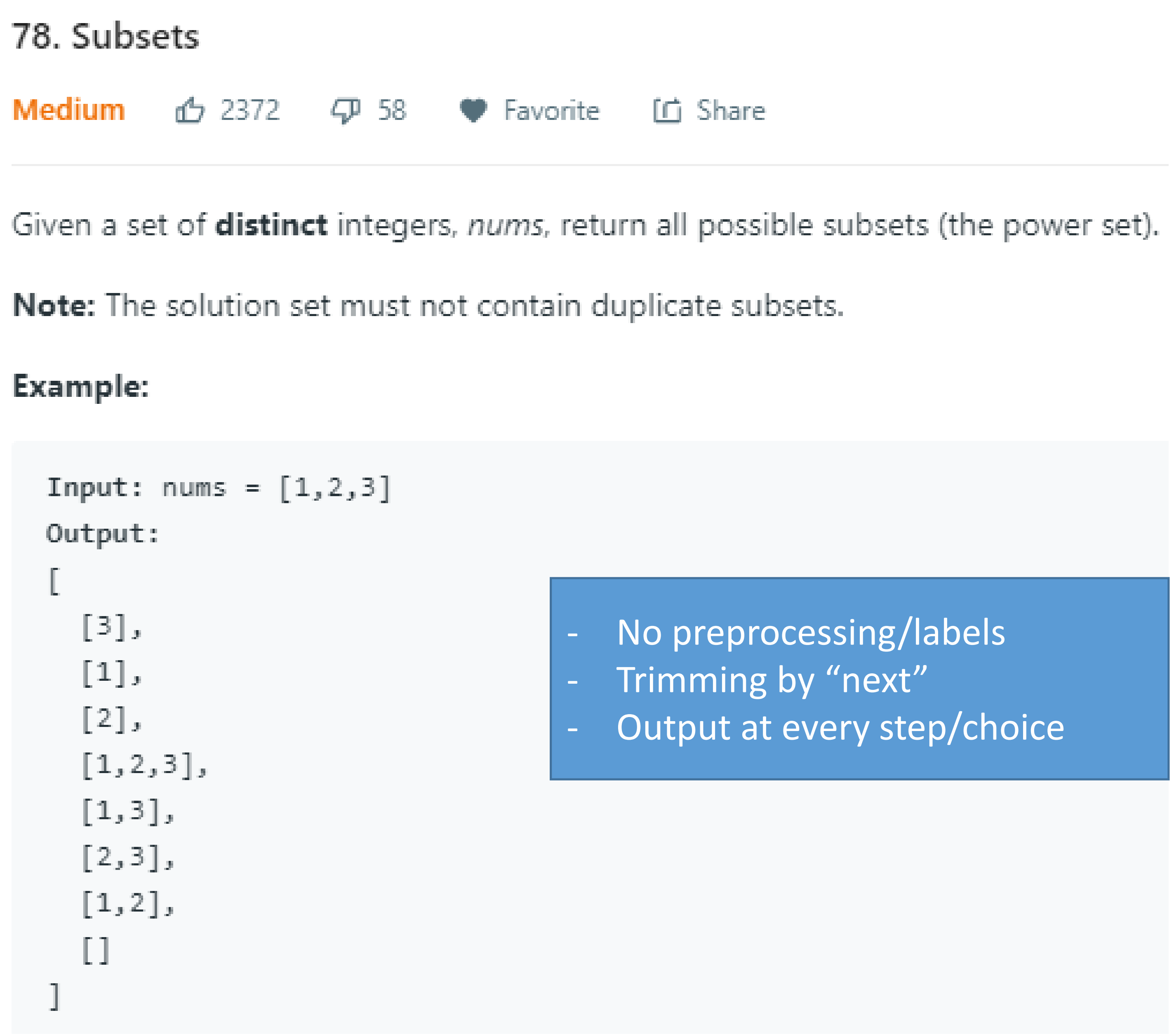
```
class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        // qcols[i] = col number of the queen on i-th row
        vector<int> qcols(n, 0);
        vector<vector<string>> ans;
        // Step 0
        solveCurrRow(qcols, 0, ans);
        return ans;
    }

    void solveCurrRow(vector<int>& qcols, int row,
                      vector<vector<string>>& ans) {
        // Step N
        if (row == qcols.size()) {
            ans.push_back(newSolution(qcols));
            return;
        }
        // Step i < N
        for (int j = 0; j != qcols.size(); ++j) {
            if (!validcol(qcols, row, j)) continue;
            qcols[row] = j;
            solveCurrRow(qcols, row + 1, ans);
        }

        return;
    }
```

HW: Can you solve 52. N-Queens II?

LC51. N-Queens: Code for checking validity & construct new Solution



```cpp
bool validcol(const vector<int>& qcols, int row, int col) {
    for (auto i = 0; i != row; ++i) {
        // check if same col or diagonal
        if (col == qcols[i]           ||
            col - row == qcols[i] - i ||
            col - qcols[i] == i - row   ) return false;
    }

    return true;
}

vector<string> newSolution(const vector<int>& qcols) {
    vector<string> solution;

    for (auto& col : qcols) {
        string onerow(qcols.size(), '.');
        onerow[col] = 'Q';
        solution.push_back(move(onerow));
    }

    return solution;
}
```



78. Subsets

Medium 👍 2372 👎 58 ♥ Favorite Share

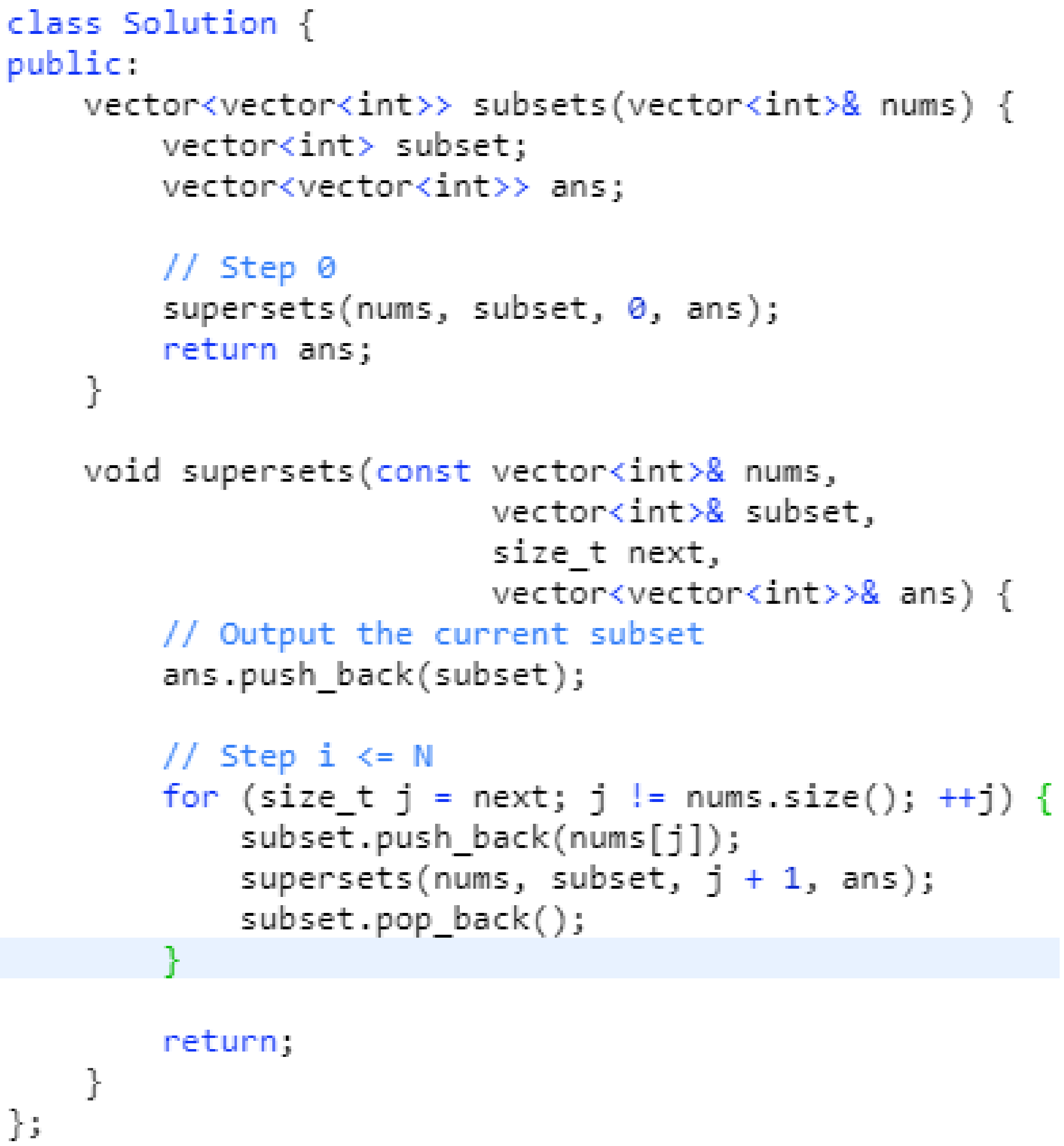
Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

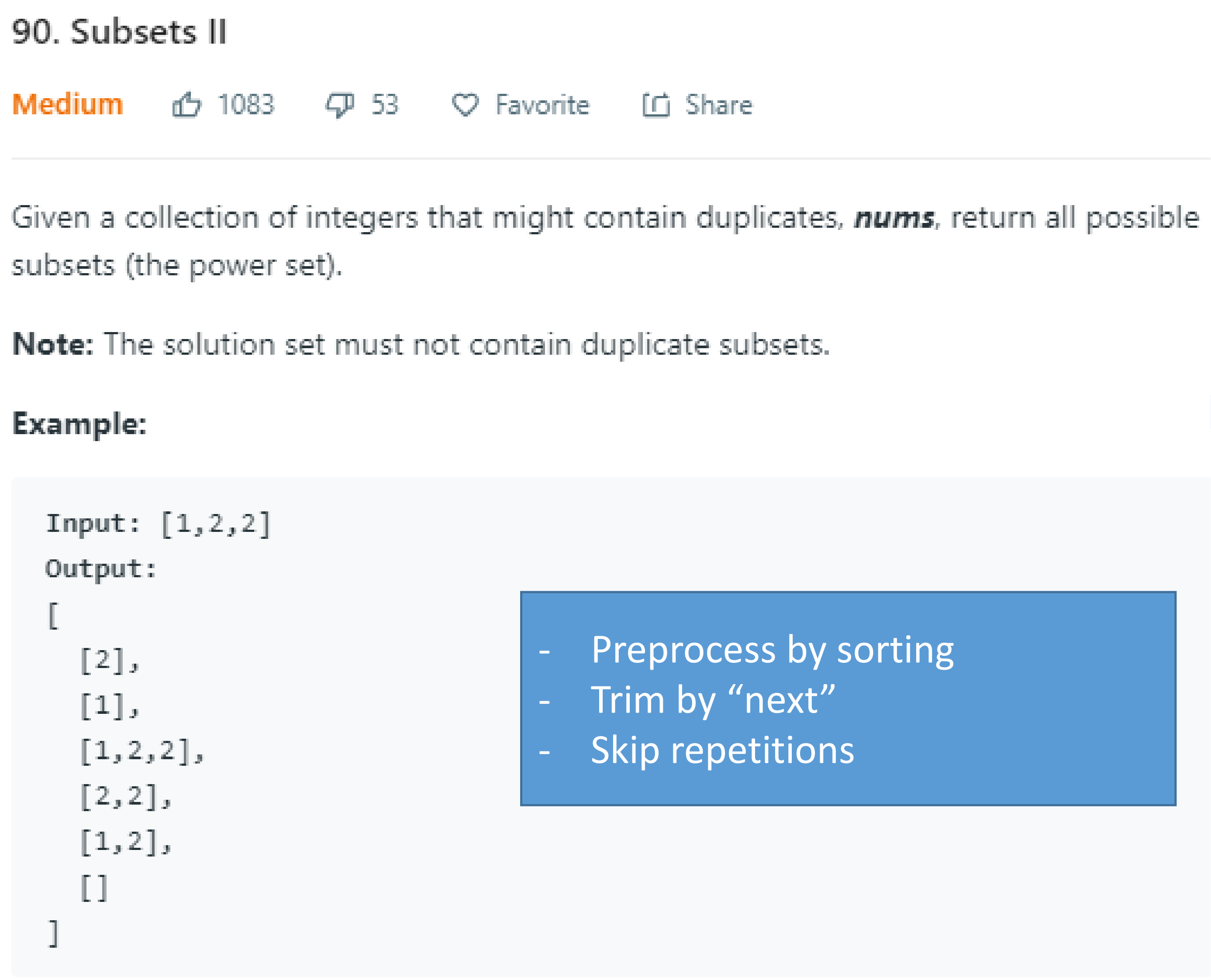
```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

- No preprocessing/labels
- Trimming by “next”
- Output at every step/choice



```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<int> subset;
        vector<vector<int>> ans;

        // Step 0
        supersets(nums, subset, 0, ans);
        return ans;
    }

    void supersets(const vector<int>& nums,
                   vector<int>& subset,
                   size_t next,
                   vector<vector<int>>& ans) {
        // Output the current subset
        ans.push_back(subset);

        // Step i <= N
        for (size_t j = next; j != nums.size(); ++j) {
            subset.push_back(nums[j]);
            supersets(nums, subset, j + 1, ans);
            subset.pop_back();
        }

        return;
    }
};
```

Can you solve it iteratively?



90. Subsets II

Medium 1083 53 Favorite Share

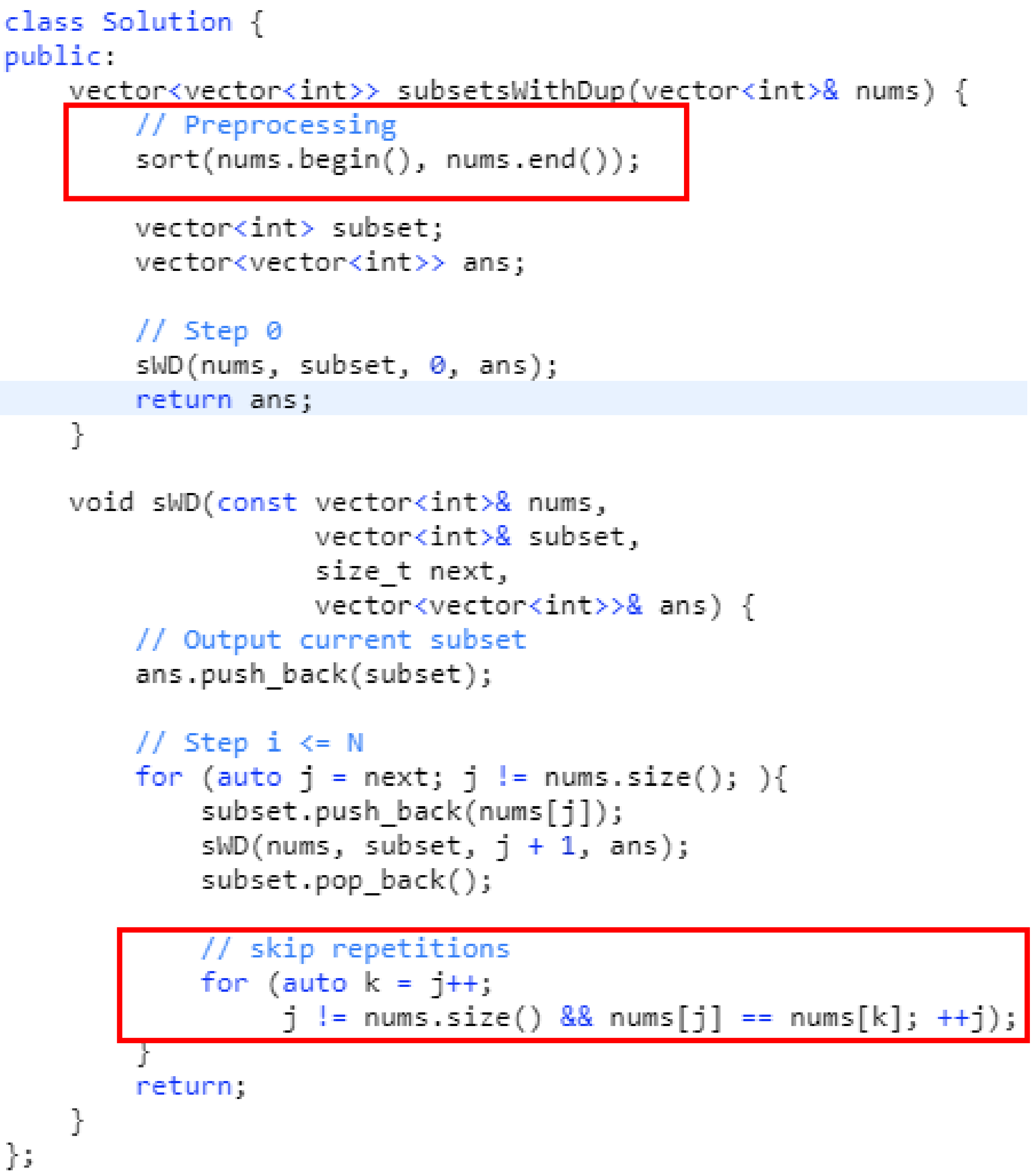
Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

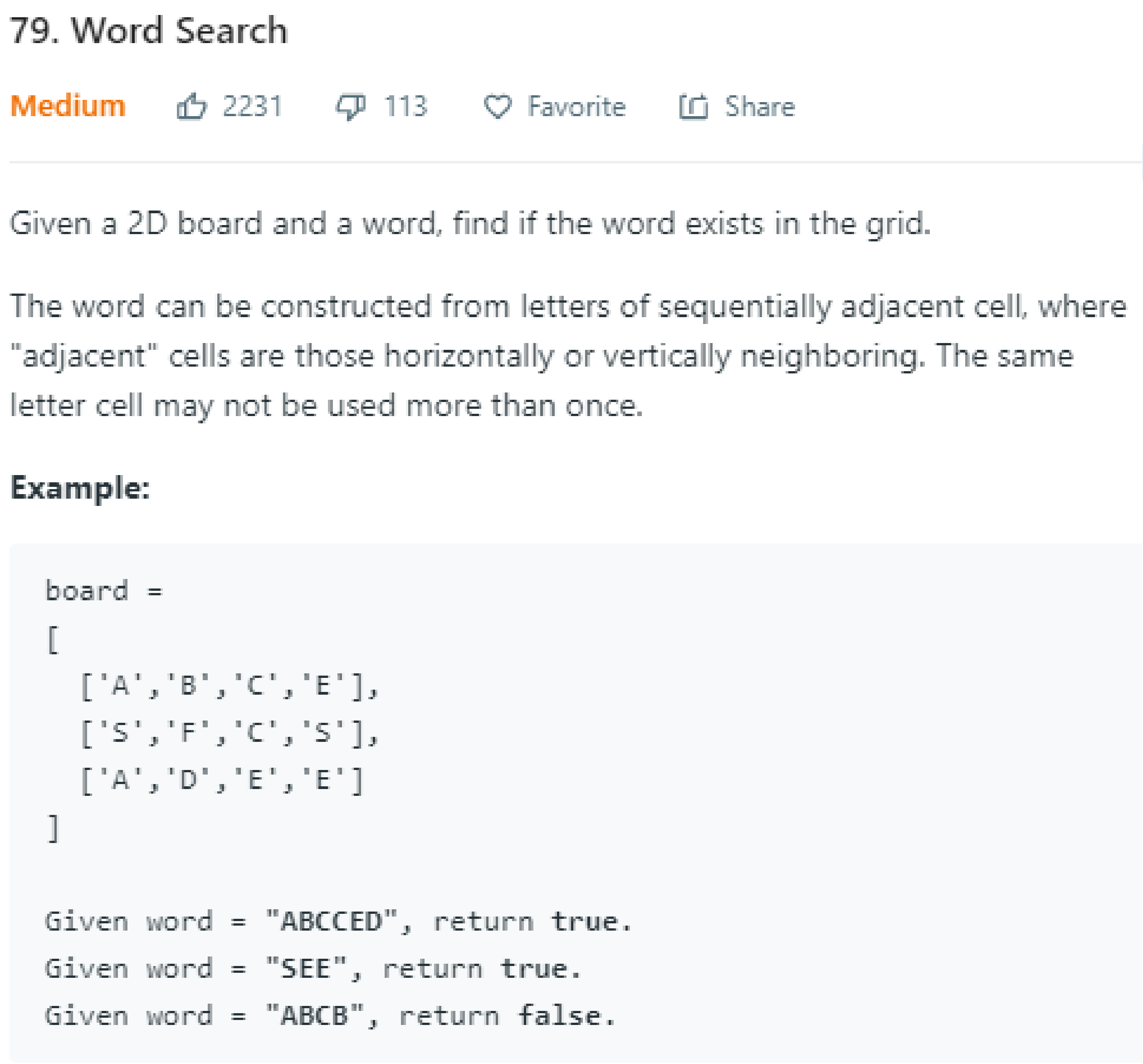
Example:

```
Input: [1,2,2]
Output:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

- Preprocess by sorting
- Trim by “next”
- Skip repetitions



```
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        // Preprocessing
        sort(nums.begin(), nums.end());

        vector<int> subset;
        vector<vector<int>> ans;

        // Step 0
        sWD(nums, subset, 0, ans);
        return ans;
    }

    void sWD(const vector<int>& nums,
                   vector<int>& subset,
                   size_t next,
                   vector<vector<int>>& ans) {
        // Output current subset
        ans.push_back(subset);

        // Step i <= N
        for (auto j = next; j != nums.size(); ){
            subset.push_back(nums[j]);
            sWD(nums, subset, j + 1, ans);
            subset.pop_back();

            // skip repetitions
            for (auto k = j++;
                 j != nums.size() && nums[j] == nums[k]; ++j);
        }
        return;
    }
};
```



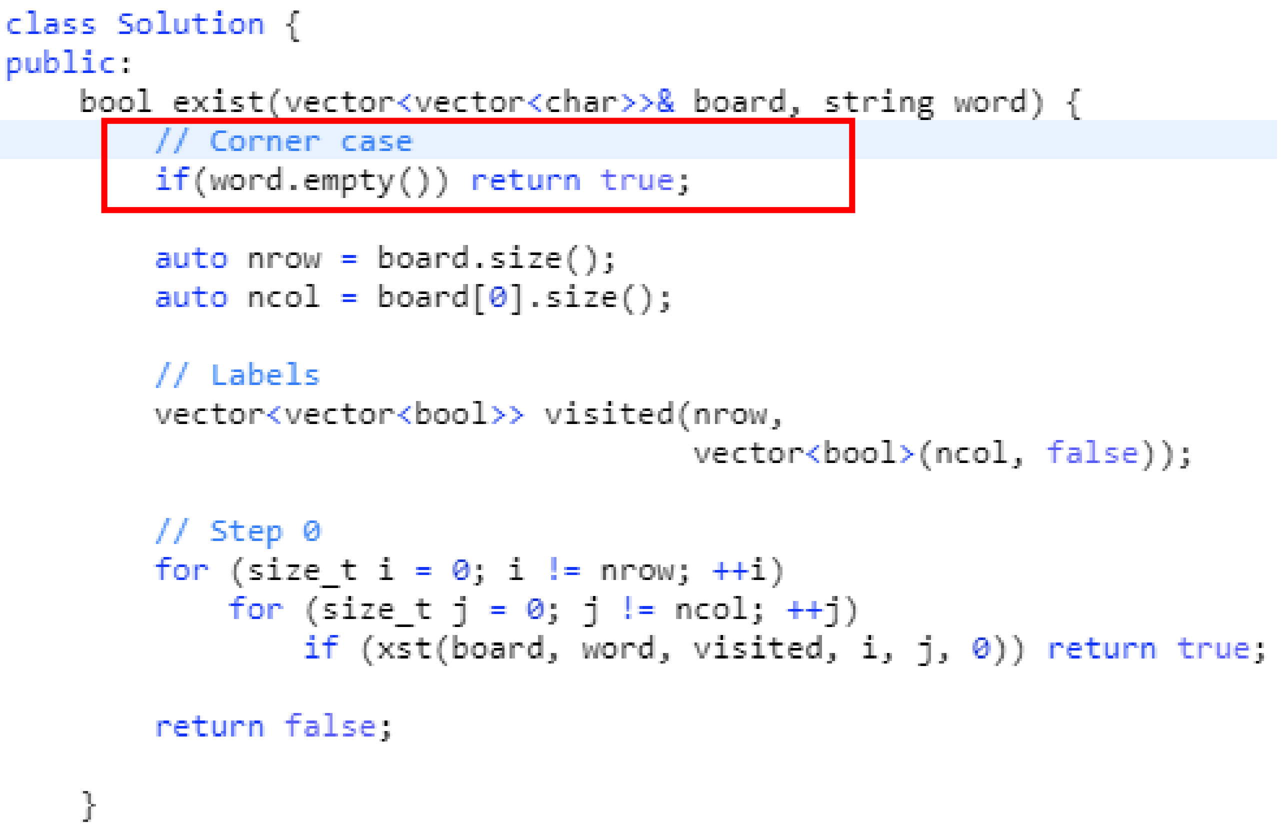
79. Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:

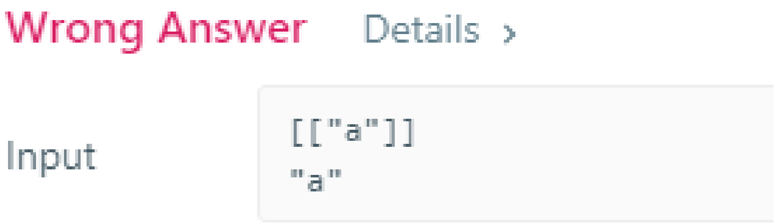
```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```



```cpp
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        // Corner case
        if(word.empty()) return true;

        auto nrow = board.size();
        auto ncol = board[0].size();

        // Labels
        vector<vector<bool>> visited(nrow,
                                     vector<bool>(ncol, false));

        // Step 0
        for (size_t i = 0; i != nrow; ++i)
            for (size_t j = 0; j != ncol; ++j)
                if (xst(board, word, visited, i, j, 0)) return true;

        return false;

    }
```

- Check corner case
- See next page for xst()

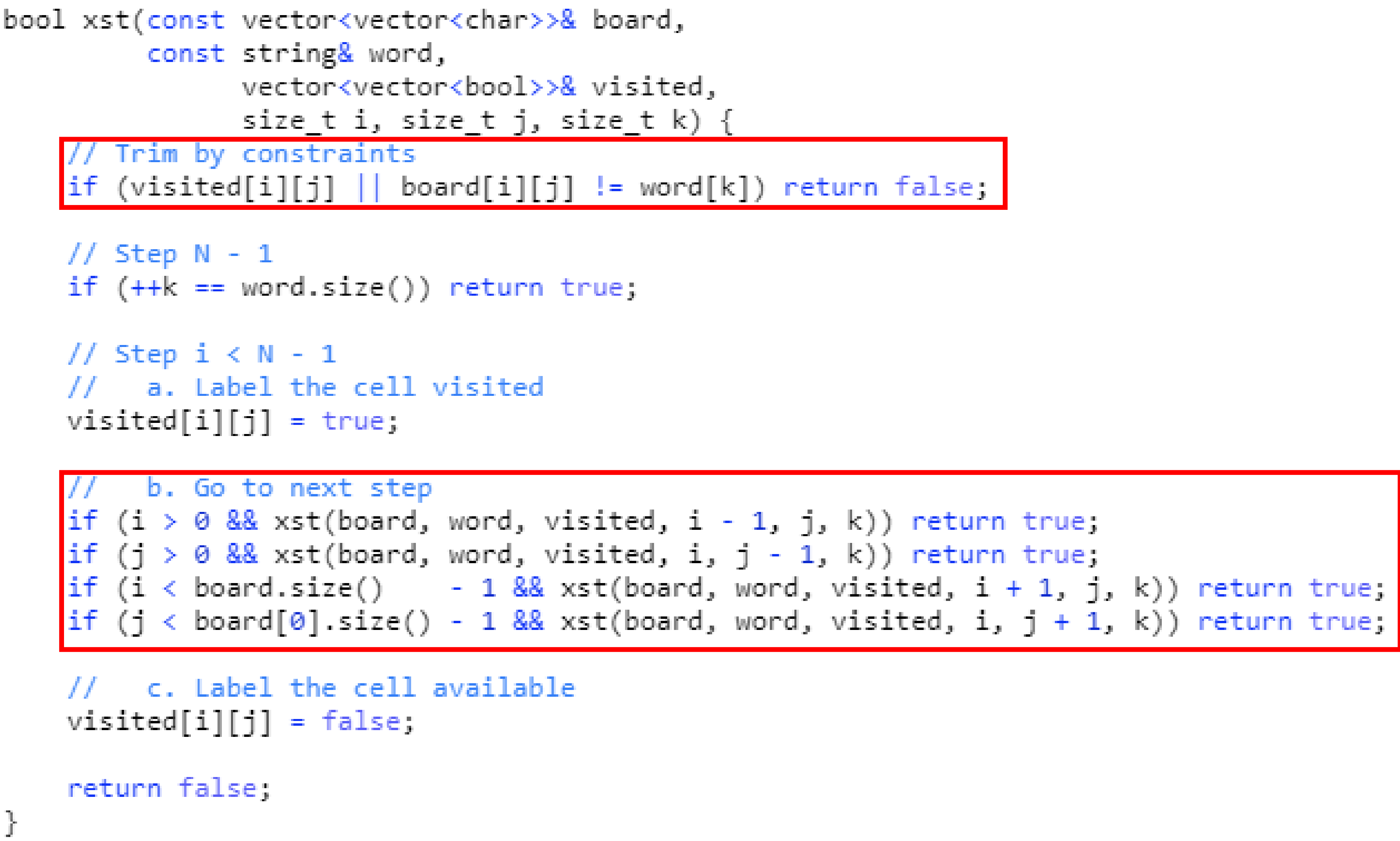
- Check corner case to avoid out of bound error
- Explicit iterations
- Stop at Step N - 1 or the test case below will fail



Wrong Answer

Input

```
[["a"]]
"a"
```



```
bool xst(const vector<vector<char>>& board,
         const string& word,
               vector<vector<bool>>& visited,
               size_t i, size_t j, size_t k) {
    // Trim by constraints
    if (visited[i][j] || board[i][j] != word[k]) return false;

    // Step N - 1
    if (++k == word.size()) return true;

    // Step i < N - 1
    //   a. Label the cell visited
    visited[i][j] = true;

    //   b. Go to next step
    if (i > 0 && xst(board, word, visited, i - 1, j, k)) return true;
    if (j > 0 && xst(board, word, visited, i, j - 1, k)) return true;
    if (i < board.size()    - 1 && xst(board, word, visited, i + 1, j, k)) return true;
    if (j < board[0].size() - 1 && xst(board, word, visited, i, j + 1, k)) return true;

    //   c. Label the cell available
    visited[i][j] = false;

    return false;
}
```