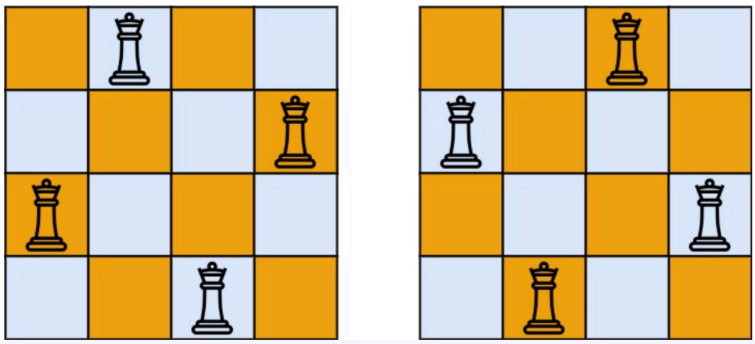
The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

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.

Example 1:



```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

Constraints:

- `1 <= n <= 9`

```
Def back_track (...) :
    stop check
    for choice in choice_list:
        do choice
        back_track (...)
        redo choice
```

Time Complexity : $O(n^2)$
Space Complexity : $O(n)$

```cpp
class Solution {
public:
    vector<vector<string>> res;

    vector<vector<string>> solveNQueens(int n) {
        vector<string> board;

        back_track(n, board);

        return res;
    }

private:
    bool valid(vector<string> board, string new_col) {
        int new_x = (int)new_col.find("Q");
        int new_y = (int)board.size();
        for (int y = 0; y < board.size(); y++) {
            string col = board[y];
            int x = (int)col.find("Q");
            if (new_x == x) {
                return false;
            }
            if (new_x == x + (new_y - y) || new_x == x - (new_y - y)) {
                return false;
            }
        }
        return true;
    }

    void back_track(int num, vector<string> board) {
        // stop check
        if (board.size() == num) {
            res.push_back(board);
            return;
        }

        // for choice in choice_list
        for (int i = 0; i < num; i++) {
            // construct choice
            string col;
            col += string(i, '.');
            col += string(1, 'Q');
            col += string(num - i - 1, '.');

            // validation check
            if (!valid(board, col)) {
                continue;
            }

            // do choice
            board.push_back(col);

            // recursive
            back_track(num, board);

            // redo choice
            board.pop_back();
        }
    }
};
```