/**
 * @brief Attempt to solve the N queens from given configuration
 * Parameters are the partially-filled board and the row index we
 * are trying to place a queen in. It will return a boolean which
 * indicates whether we succeeded in placing the rest of the queens
 * starting from this configuration.
 *
 * Base case: if there are no more queens to place,
 * then we have successfully solved the problem!
 *
 * Otherwise, we find a safe column in this row, place a queen at
 * (row,col) of the board and recursively call solveQueens starting
 * at the next row using the updated board configuration.
 * If that solveQueens call fails, remove queen from (row,col)
 * and try again with the next safe column within the row.
 * If have tried all the columns in row and did not reach a
 * solution, then we return false, which then backtracks
 * out of this unsolvable partial configuration.
 *
 * The initial call to solveQueens should be an empty board and
 * placing a queen in row 0.
 */

bool solveQueens(Board& board, int row)
{
    if (row == board.size()) { // if all rows filled
        return true; // base case: we have a solution!
    } else {
        // options to consider are columns in this row
        for (int col = 0; col < board.size(); col++) {
            if (board.isSafe(row, col)) {
                board.place(row, col); // choose
                if (solveQueens(board, row + 1)) { // explore
                    return true;
                }
                board.remove(row, col); // unchoose
            }
        }
        return false; // no col works for this row, backtrack
    }
}
/**
 * @brief Attempt to solve the Sudoku puzzle starting from given configuration
 * Parameter is the partially-filled board. It will return a boolean
 * value which indicates whether or not we successfully completed the
 * board starting from this configuration.
 *
 * Base case: if there are no more empty cells,
 * then we have successfully solved the problem!
 *
 * Otherwise, find an empty cell at (row,col), assign a digit
 * that satisfies the uniqueness property, and recursively call
 * solveSudoku on the updated board configuration.
 * If that solveSudoku call fails, remove assignment and try with
 * different digit.
 * If have tried all possible digits and did not reach a
 * solution, the function returns false. This causes the
 * algorithm to backtrack out of this unsolvable configuration.
 */

bool solveSudoku(Board& board)
{
    int row, col;

    if (!board.findEmpty(row, col)) { // if all assigned
        return true; // base case: we have a solution!
    } else {
        // options to consider are digits from 1 to 9
        for (int digit = 1; digit <= 9; digit++) {
            if (board.isUnique(row, col, digit)) {
                board.place(row, col, digit); // choose
                if (solveSudoku(board)) { // explore
                    return true;
                }
            }
        }
        board.remove(row, col); // unchoose
    }

    return false; // no digit works for this cell, backtrack
}