

## Section Solution

---

### Solution 1: Creating Word Wreck Tangles

```
bool IsReasonableToContinue(Grid<char>& board, Lexicon& lex, int row, int col)
{
    string rowText;
    for (int c = 0; c <= col; c++)
        rowText += board[row][c];

    if ((col == board.numCols() - 1) && !lex.containsWord(rowText)) return false;
    if ((col < board.numCols() - 1) && !lex.containsPrefix(rowText)) return false;

    string columnText;
    for (int r = 0; r <= row; r++)
        columnText += board.getAt(r, col);

    if ((row == board.numRows() - 1) && !lex.containsWord(columnText)) return false;
    if ((row < board.numRows() - 1) && !lex.containsPrefix(columnText)) return false;

    return true;
}

bool FindUnassignedLocation(Grid<char>& board, int& row, int& col)
{
    for (row = 0; row < board.numRows(); row++) {
        for (col = 0; col < board.numCols(); col++) {
            if (board[row][col] == ' ') return true;
        }
    }

    return false;
}

bool CreateWreckTangle(Grid<char>& board, Lexicon& lex)
{
    int row, col;
    if (!FindUnassignedLocation(board, row, col)) return true;

    Vector<char> letters;
    AddPermutationOfAlphabet(letters);

    for (int i = 0; i < letters.size(); i++) {
        board[row][col] = letters[i];
        if (IsReasonableToContinue(board, lex, row, col) &&
            CreateWreckTangle(board, lex)) return true;
    }

    board[row][col] = ' '; // pretend this never worked out..
    return false;
}
```

**Solution 2: Beehives**

```

bool ArrangementExists(Vector<string>& pieces)
{
    string anchor = pieces[0];
    pieces.removeAt(0);
    bool success = false;
    for (int k = 0; !success && k < 6; k++) {
        success = ArrangementExists(pieces, anchor[k], anchor[(k + 4) % 6]);
    }
    pieces.insertAt(anchor, 0); // restoration wasn't required...
    return success;
}

bool ArrangementExists(Vector<string>& pieces,
                       char startChar, char bridgingChar)
{
    if (pieces.size() == 0) return startChar == bridgingChar;
    bool success = false;
    for (int j = 0; !success && j < pieces.size(); j++) {
        string bridgingPiece = pieces[j];
        pieces.removeAt(j);
        for (int k = 0; !success && k < 6; k++) {
            if (bridgingPiece[k] == bridgingChar) {
                char newBridgingChar = bridgingPiece[(k + 4) % 6];
                success = ArrangementExists(pieces, startChar, newBridgingChar);
            }
        }
        pieces.insertAt(bridgingPiece, j);
    }
    return success;
}

```

**Solution 3: Revisiting SuDoKu**

```

bool HasUniqueSolution(Grid<int>& board)
{
    int numSolutions = 0;
    SolveSudoku(board, numSolutions);
    return numSolutions == 1;
}

static const int kUnassigned = 0;
void SolveSudoku(Grid<int>& grid, int& numSolutions)
{
    int row, col;
    if (!FindUnassignedLocation(grid, row, col)) {
        numSolutions++;
        return;
    }

    for (int num = 1; num <= 9; num++) {
        if (NoConflicts(grid, row, col, num)) {
            grid[row][col] = num;
            SolveSudoku(grid, numSolutions);
            if (numSolutions > 1) return; // if beyond a unique solution, we're done
            grid[row][col] = kUnassigned;
        }
    }
}

```

## Solution 4: Minimizing Quicksort Recursion

```
void Quicksort(Vector<int>& vec, int start, int finish)
{
    while (start < finish) { // while unsorted range is of size two or more
        int boundary = Partition(vec, start, finish);
        int mid = (start + finish)/2;
        if (boundary <= mid) {
            // subvector left of partition is smaller, so
            // recur on that, then redefine start
            Quicksort(vec, start, boundary - 1);
            start = boundary + 1;
        } else {
            // subvector right of partition is smaller, so
            // recur on that, then redefine finish
            Quicksort(vec, boundary + 1, finish);
            finish = boundary - 1;
        }
    }
}
```