

## Section Solution

---

### Problem 1: Generating Anagrams

```
bool FindAnagramWithFixedPrefix(string prefix, string rest,
                                Lexicon& lex, Vector<string>& words)
{
    if (!lex.containsPrefix(prefix)) return false;
    if (lex.containsWord(prefix) && prefix.length() >= 4) {
        if (rest == "" || FindAnagram(rest, lex, words)) {
            words.add(prefix);
            return true;
        }
    }

    for (int i = 0; i < rest.length(); i++) {
        if (FindAnagramWithFixedPrefix(prefix + rest[i],
                                        rest.substr(0, i) + rest.substr(i + 1),
                                        lex, words)) return true;
    }

    return false;
}

bool FindAnagram(string letters, Lexicon& lex, Vector<string>& words)
{
    return FindAnagramWithFixedPrefix("", letters, lex, words);
}
```

### Problem 2: Creating Word Wreck Tangles

```
/**
 * Function: IsReasonableToContinue
 * -----
 * Contemplates the given board, which is assumed to be populated
 * from left to right, top to bottom with visible characters, all
 * the way through the (row, col) entry. We assume that this (row, col)
 * character is the most recently placed character and is the only
 * character that could have invalidated the partial board. We
 * check the string in the rowth row and the colth column to see if this
 * most recently added character is extending previously approved
 * of prefixes to be longer prefixes (or words, if the (row, col)
 * position is along the left and/or bottom boundary.)
 */

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

/**
 * Predicate: FindUnassignedLocation
 * -----
 * Reads the Grid as we'd read a book and returns true iff and only if
 * it finds a position within the Grid that houses a ' '. If true
 * is returned, then the ints referenced by row and col are updated
 * to store the row and column of that space character. If false is
 * returned, then the ints referenced by row and col are invalidated.
 */

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.setAt(row, col, letters[i]);
        if (IsReasonableToContinue(board, lex, row, col) &&
            CreateWreckTangle(board, lex)) return true;
    }

    board.setAt(row, col, ' '); // pretend this never worked out..
    return false;
}

```

**Problem 3: 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;
}

```

**Problem 4: Variation On Boggle**

```

bool CanSpell(string word, Vector<string>& cubes)
{
    if (word.empty()) return true;

    for (int i = 0; i < cubes.size(); i++) {
        string curCube = cubes[i];
        if (curCube.find(word[0]) != string::npos) {
            cubes.removeAt(i); // remove cube so not used again
            if (CanSpell(word.substr(1), cubes)) {
                cubes.insertAt(i, curCube);
                return true;
            }
            cubes.insertAt(i, curCube); // backtrack, replace cube
        }
    }

    return false; // trigger backtracking
}

```