# Section Solution

## Problem 1 Solution: Domino Chaining

The solution looks like typical recursive backtracking, save for the fact there are two recursive calls per iteration instead of just one. There's some über-clever short-circuit evaluation going on here, where recursive calls are circumvented unless two numbers that need to match actually match. Note that we don't make a second recursive call within any given iteration if the first one works out, or if each half of the chaining domino has the same number.

```
static bool chainExistsRec(Vector<domino>& dominoes, int start, int end) {
   if (start == end) return true;
   if (dominoes.isEmpty()) return false; // technically optional! know why?

   for (int i = 0; i < dominoes.size(); i++) {
      domino d = dominoes[i];
      dominoes.remove(i);
      if ((d.first == start && chainExistsRec(dominoes, d.second, end)) ||
          (d.first != d.second &&
           d.second == start && chainExistsRec(dominoes, d.first, end)))
         return true;
      dominoes.insert(i, d); // pretend we never made this choice by reverting
   }

   return false;
}

static bool chainExists(const Vector<domino>& dominoes, int start, int end) {
   Vector<domino> copy = dominoes; // we need our own, mutable copy
   return chainExistsRec(copy, start, end);
}
```

In this case, I go with a wrapper not because I need to introduce any new parameters, but because I need a deep clone of the supplied **`Vector`** so I can add and remove from it knowing it won't impact the original.

One could also argue that the **`insert`** and **`remove`** calls are time consuming, but the domain is such that we never expect, at least in practice, that the set of dominoes is all that large, and optimizing for speed when it won't buy us very much just makes the recursion harder to follow. If you're really concerned about running time for large domino sets, then you might go with a version that swaps the chaining domino to the end before removing it, eventually re-introducing it at the end and swapping it back to its original position, like this:

```
static bool chainExistsRec(Vector<domino>& dominoes, int start, int end) {
   if (start == end) return true;
   if (dominoes.isEmpty()) return false; // technically optional! know why?

   for (int i = 0; i < dominoes.size(); i++) {
      domino d = dominoes[i];
      swap(dominoes[i], dominoes[dominoes.size() - 1]);
      dominoes.remove(dominoes.size() - 1);
      if ((d.first == start && chainExistsRec(dominoes, d.second, end)) ||
          (d.first != d.second &&
           d.second == start && chainExistsRec(dominoes, d.first, end)))
         return true;
      dominoes += d;
      swap(dominoes[i], dominoes[dominoes.size() - 1]);
   }

   return false;
}
```

The student truly anxious about wasted work will complain that each of the two solutions above **`remove`** and re-**`insert`** the $i^{th}$ domino whether we end up making recursive calls or not. It's reasonable to commit to the swap-and-remove trick only after we decide a recursive call should be made. And as it turns out, if we get information that removing the $i^{th}$ domino set up a sub-problem that couldn't be solved recursively, we know the $i^{th}$ domino will **never** be part of any solution. That means we don't need to re-**`insert`** it.

```
static bool chainExistsRecOpt(Vector<domino>& dominoes, int start, int end) {
   if (start == end) return true;
   if (dominoes.isEmpty()) return false; // technically optional! know why?

   for (int i = 0; i < dominoes.size(); i++) {
      domino d = dominoes[i];
      if (d.first == start || d.second == start) {
         // only delete if we're going to recur
         swap(dominoes[i], dominoes[dominoes.size() - 1]); // send d to back
         dominoes.remove(dominoes.size() - 1);
         if ((d.first == start && chainExistsRecOpt(dominoes, d.second, end)) ||
             (d.first != d.second &&
              d.second == start && chainExistsRecOpt(dominoes, d.first, end)))
            return true;
         // got there and d didn't connect us? It never will, so leave it out!
         i--; // but something else took its place (so don't skip it)
      }
   }

   return false;
}
```

Be clear, however, that the first solution of the three is perfectly acceptable, because I'm more interested in your recursive thinking. Only after you get the recursion working should you analyze your algorithm and/or profile your code to determine where things are unnecessarily slow.

**Problem 2 Solution: Finding Anagrams**

This is the most difficult recursion problem I've ever included in a CS106 lecture, section handout, assignment, or exam. Recursion problems aren't usually this dense, but there's value in examining it anyway, as it pushes the limits of what you think you're able to understand.

```
// forward declare second prototype because of mutual recursion
static bool findAnagram(const string& letters, const Lexicon& english,
                        Vector<string>& words);

static const int kThresholdLength = 4;
static bool findAnagram(const string& prefix, const string& rest,
                        const Lexicon& english, Vector<string>& words) {

   if (!english.containsPrefix(prefix)) return false; // up-communicate failure
   if (english.contains(prefix) &&
       prefix.length() >= kThresholdLength &&
       (rest.empty() || findAnagram(rest, english, words))) {
      words.add(prefix); // add word to the accumulation of other words
      return true;       // up-communicate success!
   }

   for (size_t i = 0; i < rest.length(); i++) {
      string extended = prefix + rest[i];
      string restofrest = rest.substr(0, i) + rest.substr(i + 1);
      if (findAnagram(extended, restofrest, english, words))
         return true;
   }

   return false;
}

static bool findAnagram(const string& letters, const Lexicon& english,
                        Vector<string>& words) {
   // because we're taking the approach where a working prefix is recursively
   // extended, we need to introduce the granddaddy of all prefixes—the empty
   // string—as the initial prefix
   return findAnagram("", letters, english, words);
}
```

The **`for`** loop within the four-argument version is classic recursive backtracking—repeated attempts to return **`true`**, returning **`false`** only after you've considered all possible ways to extend the running prefix with some letter in **`rest`**. The decision to frame the three-argument version in terms of a four-argument version is also something we've seen a lot of.

The second base case is, in my opinion, the hardest part to understand. It reads as follows:

> *If the supplied prefix is an English word (and it's long enough) and we're either out of letters or we hear back that the remaining letters can be anagrammed, then we should return* **`true`**. *Otherwise, we should advance on to the recursion phase to see if the prefix can be extended to make everything recursively work out.*

**Problem 3 Solution: Revisiting SuDoKu**

The overarching change to the program presented in lecture is that I thread a reference to a counter throughout the recursion, and I increment that counter every time I find a solution. Note that **solve** doesn't return a **true** or a **false**; instead, it effectively returns information about success or failure by detecting changes in that counter. The **count > 1** scenario (or equivalently, a **count == 2** scenario) is one we're continually polling to see if we have reason to stop searching, since a **count** of 2 and a **count** of 2000 are equally bad if we're hoping for a puzzle with a unique solution.

```
static void solve(Grid<int>& board, int& count) {
   int row, col;
   if (!findLocation(board, row, col)) {
      count++; // we found a solution!
      return;
   }

   for (int num = 1; num <= 9; num++) {
      if (noConflicts(board, row, col, num)) {
         board[row][col] = num;
         solve(board, count);
         if (count > 1) return; // do we have too many solutions? yes? punt!
         board[row][col] = kUnassigned;
      }
   }
}

static bool hasUniqueSolution(Grid<int>& board) {
   int count = 0;
   solve(board, count);
   return count == 1;
}
```

**Problem 4 Solution: The XL Puzzle**

I elect to implement memoization, because it speeds things up (more so for the 1-to-50 version, but still a little for the 1-to-40 version), and because it's cool. This is also the first time I pass through a **position** index so I know what part of the string I'm trying to find. In other problems, I often create a new, slightly smaller string with something like **substr(1)**, and you could certainly do that here as well. But since I'm dealing with an original string that's hundreds of characters long, it felt a little silly and wasteful to create so many deep copies of almost-as-long strings with the **substr(1)** calls when the approach I take here is almost, if not as, easy.

Inspect the next page for my solution. Note that **NORTH**, **EAST**, **SOUTH**, and **WEST** enumerated type constants are all backed by consecutive integers, so we can for loop over them in that order, as you'll see that I do precisely that.

```
static bool solutionExists(XLUpDisplay& display, const Grid<char>& grid,
                           const string& numbers, int position, const coord& curr,
                           Stack<coord>& path, Map<coord, Set<int>>& cache) {

   if (position == numbers.size()) return true;
   if (!grid.inBounds(curr.row, curr.col)) return false;
   if (grid.get(curr.row, curr.col) != numbers[position]) return false;
   if (cache.containsKey(curr) && cache[curr].contains(position)) return false;

   display.provisonallyMove(curr);

   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (solutionExists(display, grid, numbers, position + 1,
                         neighboringCoord(curr, dir), path, cache)) {
         path.push(curr);
         return true;
      }
   }

   display.vetoProvisionalMove(curr);
   display.eraseProvisionalMove(curr);
   cache[curr] += position; // remember failure of (coord, position) combination
   return false;
}

static bool solutionExists(XLUpDisplay& display, const Grid<char>& grid,
                           const string& numbers, const coord& curr,
                           Stack<coord>& path) {
   Map<coord, Set<int>> cache; // memoization just speeds things up
   return solutionExists(display, grid, numbers, 0, curr, path, cache);
}
```