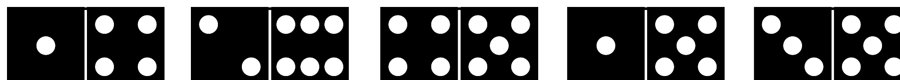
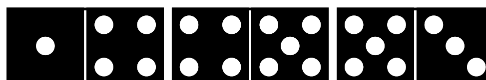
Section Handout

Problem 1: Domino Chaining [courtesy of Eric Roberts]

The game of dominoes is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following five rectangles represents a domino:

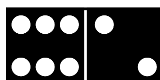


Dominoes are connected end-to-end to form chains, subject to the condition that two dominoes can be linked together only if the numbers match (although it is legal to rotate dominoes 180˚ so that the numbers are reversed). For example, you could connect the first, third, and fifth dominoes in the above collection to form the following chain:



Note that the 3-5 domino had to be rotated so that it matched up correctly with the 4-5.

Given a set of dominoes, an interesting question to ask is whether it is possible to form a chain starting at one number and ending with another. For example, the example chain shown earlier makes it clear that you can use the original set of five dominoes to build a chain starting with a 1 and ending with a 3. Similarly, if you wanted to build a chain starting with a 6 and ending with a 2, you could do so using only one domino:



On the other hand, there is no way—using just these five dominoes—to build a chain starting with a 1 and ending with a 6.

Dominoes can, of course, be represented in C++ very easily as a pair of integers. Assuming the type **`domino`** is defined as

```
struct domino {
   int first;
   int second;
};
```

write a predicate function:

```
static bool chainExists(const Vector<domino>& dominoes, int start, int end);
```

that returns **`true`** if it is possible to build a chain from **`start`** to **`end`** using any subset of the dominoes in the **`dominoes`** vector. To simplify the problem, assume that **`chainExists`** always returns **`true`** if **`start`** is equal to **`end`**, because you can trivially connect any number to itself with a chain of zero dominoes. Don't worry about what the chain is—worry only about the yes or no that comes back in the form of a **`bool`**.

For example, if **`dominoes`** is the domino set illustrated above, **`chainExists`** should produce the following results:

```
chainExists(dominoes, 1, 3) → true
chainExists(dominoes, 5, 5) → true
chainExists(dominoes, 1, 6) → false
```

Problem 2: Finding Anagrams

An anagram is a word or phrase formed by the rearrangement of the letters of another word or phrase. Here are a few of the funnier ones I found a long time ago while surfing **`http://wordsmith.org/anagram/hof.html`**.

partial men is an anagram of **parliament**.
Old West Action is an anagram of **Clint Eastwood**
The United States of America is an anagram of **Attaineth its cause: freedom**
Firefox browser is an anagram of **fix web of errors**

We're interested in writing code that, given a string of lowercase letters (i.e. **`"oldwestaction"`**), manages to find any one of its anagrams. Here's the prototype I want you to work with:

```
static bool findAnagram(const string& letters, const Lexicon& english,
                        Vector<string>& words);
```

Implement the **`findAnagram`** function as described above, which returns **`true`** if and only if the supplied string of letters can be permuted into a word or concatenation of words. When **`true`** is returned, the **`Vector`** addressed by **`words`** should be populated with the sequence of words making up the anagram. Further constrain that all words in the anagram be at least four letters long.

Problem 3: Revisiting SuDoKu

The following backtracking function was presented in class as a way of solving a SuDoKu puzzle. The details of the helper functions aren't important here, as you can intuit what they do based on how and where they're called.

```
static const int kUnassigned = 0;
static bool solve(Grid<int>& board) {

   int row, col;
   if (!findLocation(board, row, col)) return true;

   for (int num = 1; num <= 9; num++) {
      if (noConflicts(board, row, col, num)) {
         board[row][col] = num;
         if (solve(board)) return true;
         board[row][col] = kUnassigned;
      }
   }

   return false;
}
```

One problem of interest that the above version doesn't quite solve is whether the solution is unique. Properly constructed SuDoKu boards are such that there's only one way to solve the puzzle. Not zero. Not two or more. Just one. ☺

Cannibalize the above implementation to author **`hasUniqueSolution`**, which calls a modified version of **`solve`** and returns **`true`** if and only if there's exactly one way to assign numbers to the open squares to solve the puzzle.

```
static bool hasUniqueSolution(Grid<int>& board);
```

Problem 4: The XL Puzzle

The XL puzzle is a single-player game where one is challenged to navigate through a 7 x 7 grid of randomly placed Roman numeral digits (I, V, X, and L) and, beginning from the center square, stamp through the Roman numerals I through XL (1 through 40), each separated by a space. One can step to any neighboring square (up, down, left, or right, but not diagonally), and the same location can be used multiple times, even within the same numeral.

I	I	X	V		L	X
X	L		I	X	V	I
	I	V	I	X		X
L	I	X		V	I	X
X		X	I	X	I	
L	I	V	L		X	X
V	I		X	I	X	L

The problem can be framed as a variation of Boggle, except that the word of interest is always taken to be **"I II III IV V VI VII VIII IX X XI XII XIII XIV XV XVI XVII XVIII XIX XX XXI XXII XXIII XXIV XXV XXVI XXVII XXVIII XXIX XXX XXXI XXXII XXXIII XXXIV XXXV XXXVI XXXVII XXXVIII XXXIX XL XLI XLII XLIII XLIV XLV XLVI XLVII XLVIII XLIX L"**.

The puzzle itself is modeled as a **Grid<char>**, where each character in the grid is understood to be either an **'I'**, a **'V'**, an **'X'**, an **'L'**, or a space character, and the center coordinate is constrained to be a space.

Couple the above with the definition of a **coord**, the **Direction** type as defined in the CS106 library **"direction.h"**, and a helper function around coordinates and directions, as provided below.

```
struct coord {
   int row;
   int col;
};
```

```
static coord neighboringCoord(const coord& c, Direction dir) {
   coord next = c;
   switch (dir) {
      case NORTH: next.row--; break;
      case EAST: next.col++; break;
      case SOUTH: next.row++; break;
      case WEST: next.col--; break;
   }

   return next;
}
```

You're then equipped to implement a recursive backtracking routine which decides whether it's possible to roll over the numbers I through XL, in succession, starting from the center coordinate, while respecting all of the rules.

Present your implementation of **`solutionExists`**, which returns **`true`** if and only if there's some way to roll through the numbers 1 through 40, Roman numeral style, and if **`true`**, populates the path stack with the sequence of coordinates you should step through to prove the puzzle is solvable.

Three things:

- As it turns out, one can roll over the numbers 1 through 51—that is, I through LI—but there's no way to reach LII/52. Use this information to confirm that your solution starts to return **`false`** when asked to go that high.
- A particularly ambitious solution will use caching and memoization to ensure that you don't waste time searching for a particular suffix from a particular coordinate a second time if it didn't work out the first time. If you have the time and the ambition, install a memoization-caching layer into your solution.
- If you're really feeling ambitious and you lack interest in anything other than CS106X, you might spend some time generating a small, square puzzle—perhaps 9 x 9, 11 x 11, 13 x 13, or whatever it takes—where one can start from the center and roll through the numbers 1 through 100.