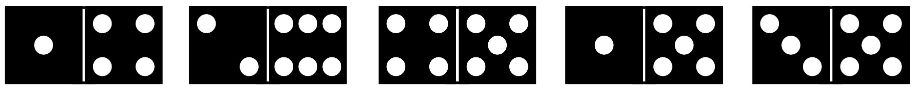
CS106B Handout 19
Autumn 2012 October 15th, 2012

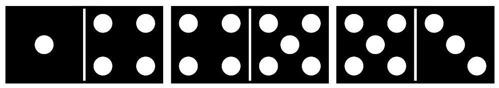
Section Handout

Discussion 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:

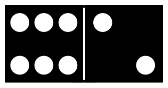


Dominoes are connected end-to-end to form chains, subject to the condition that two dominoes can be linked together only if the numbers match, although it is legal to rotate dominoes 180˚ so that the numbers are reversed. For example, you could connect the first, third, and fifth dominoes in the above 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 **`finish`** 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 **`finish`**, 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 not 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
```

Discussion Problem 2: Revisiting SuDoKu

A variation on the following code block was presented in class on Wednesday (and in Handout 20) as a way of populating a SuDoKu board with an assignment that solves the puzzle. The details of the helper functions aren't important here; you can intuit what they do based on how and where they're called.

```
static const int kEmpty = 0;
static bool solve(Grid<int>& board) {
    int row, col;
    if (!findEmptyLocation(board, row, col)) return true;

    for (int digit = 1; digit <= 9; digit++) {
        if (isFreeOfConflict(board, row, col, digit)) {
            board[row][col] = digit;
            if (solve(board)) return true;
            board[row][col] = kEmpty;
        }
    }

    return false;
}
```

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

Leveraging the above implementation, write a function called **`hasUniqueSolution`**, which calls a **modified** version of **`solve`**, that returns **`true`** if and only if there's exactly one legitimate way to assign numbers to the open squares. Here's what you need to do:

- Implement **`hasUniqueSolution`** as a single call to a new version of **`solve`**, which in addition to the **`Grid<int>&`** will need to take one or more additional parameters. You will need to rewrite **`solve`** yourself to be a slight variation on what's presented above.

- Update **`solve`** to keep running even after a solution has been found. It needs to continue searching in order to confirm that the first solution is really the only one.
- If a second solution is found, then at that point you should stop searching, short-circuit, and have **`hasUniqueSolution`** return **`false`**.
- You may leave the SuDoKu board empty, partially solved, or fully solved when **`hasUniqueSolution`** returns.

Lab Problem 1: Boggle Lite

You're going to work through a supervised lab problem that's requires you write code similar to that you'll need for your Boggle assignment. You're to implement a recursive backtracking function **`canSpell`** that determines whether or not a word can be spelled by placing letter cubes side by side. The letter cubes are expressed as **`string`**s of length 6, and the collection of cubes is stored in a **`Vector<string>`**. Each cube can be used at most once, but you're free to subset, reorder, and rotate the cubes as needed if it helps to spell.

The problem requires you to implement:

```
   static bool canSpell(const string& word, const Vector<string>& cubes,
                        Stack<string>& selections);
```

to contribute to a larger application that asks if words can be spelled. Your implementation should also accumulate copies of the selected cubes into the referenced **`Stack<string>`**. My own solution places **square brackets** around the character within each selected letter cube to accentuate which side would need to be face up.

Here's a short sample run of my own solution:

```
   Enter a word you're trying to spell [or hit <enter> to quit]: boggle
   I can spell that this way:
      a[b]bjoo ach[o]ps aaee[g]n ee[g]hnw dei[l]rx d[e]lvry
   Enter a word you're trying to spell [or hit <enter> to quit]: recurse
   I can spell that this way:
      deil[r]x aa[e]egn a[c]hops cimot[u] delv[r]y affkp[s] [e]eghnw
   Enter a word you're trying to spell [or hit <enter> to quit]: question
   I can spell that this way:
      himn[q]u cimot[u] aa[e]egn achop[s] aoo[t]tw de[i]lrx abbj[o]o eegh[n]w
   Enter a word you're trying to spell [or hit <enter> to quit]: zzyzzva
   No, I can't spell that.  Sorry.
   Enter a word you're trying to spell [or hit <enter> to quit]:
```