Discussion 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.

```cpp
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 copy so we can modify it
    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\textsuperscript{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\textsuperscript{th} domino set up a sub-problem that couldn’t be solved recursively, we know the i\textsuperscript{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 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.
Discussion Problem 2 Solution: Revisiting SuDoKu

The new version is structurally similar to the old one. After all, we need to find solutions, and the recursive substructure is the same whether we’re trying to find just one solution or many of them. However, traditional bool return value isn’t needed, because this isn’t standard backtracking. The procedural recursion operates like it needs to find all solutions, not just one. We thread a reference to a master count variable throughout, and every time we discover a solution, we increment that count by one. And because we don’t always need an exact count on the number of solutions—for our purposes, 2 is the same as 2000—we implant a short-circuiting check just after the recursive call to check to see whether we’d be wasting time trying to find out more.

```cpp
static void solve(Grid<int>& board, int& numSolutions) {
    int row, col;
    if (!findEmptyLocation(board, row, col)) {
        numSolutions++;
        return;
    }

    for (int digit = 1; digit <= 9; digit++) {
        if (isFreeOfConflict(board, row, col, digit)) {
            board[row][col] = digit;
            solve(board, numSolutions);
            if (numSolutions > 1) return; // if solution isn’t unique, we’re done
            board[row][col] = kEmpty;
        }
    }
}

static bool hasUniqueSolution(Grid<int>& board) {
    int numSolutions = 0; // set up the master count variable
    solve(board, numSolutions);
    return numSolutions == 1; // numSolutions will only ever be 0, 1, or 2
}
Lab Problem 1 Solution: Boggle Lite

The first of these two functions does the backtracking, and the second just wraps a call to the first so it can create a copy of the client’s read-only `cubes` variable. The recursive approach is to find a cube that offers up the leading letter of `str`, and to see whether the other cubes can be used to construct the rest of it. If so, we record which cube we’re using to cover `str[0]` by pushing a copy of it (modified with [ and ] around the matching letter) onto a stack and return `true` to signal victory. If not, we search for some other cube, and possibly another, and so on until we either recursively succeed, or we’re forced to admit defeat because we ran out of options.

```cpp
static bool canSpellRec(const string& str, Vector<string>& cubes,
                         Stack<string>& selections) {
    if (str.empty()) return true;
    if (cubes.isEmpty()) return false; // trigger backtracking

    for (int i = 0; i < cubes.size(); i++) {
        int found = cubes[i].find(str[0]);
        if (found != string::npos) {
            string cube = cubes[i];
            cubes.remove(i);
            if (canSpellRec(str.substr(1), cubes, selections)) {
                cube.replace(found, 1,
                              string("[") + cube[found] + string("]");
                selections.push(cube);
                return true;
            }
            cubes.insert(i, cube);
            // recursive call failed: undo, try something else
        }
    }

    return false;
}
```

```cpp
static bool canSpell(const string& str, const Vector<string>& cubes,
                      Stack<string>& selections) {
    Vector<string> copy = cubes;
    return canSpellRec(str, copy, selections);
}
```