CS106X
Autumn 2015 Cynthia Lee

Section 3 (Week 4) – SOLUTION

Problem and solution authors include Marty Stepp, Jerry Cain,
Eric Roberts, Ilan Goodman, and Cynthia Lee.

Problem 1 Solution: isSubsequence [courtesy of Marty Stepp]

```
bool isSubsequence(string big, string small) {
	if (small == "") {
		return true;
	} else if (big == "") {
		return false;
	} else {
		if (big[0] == small[0]) {
			return isSubsequence(big.substr(1), small.substr(1));
		} else {
			return isSubsequence(big.substr(1), small);
		}
	}
}
```

Problem 2 Solution: Domino Chaining [courtesy of Jerry Cain and Eric Roberts]

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 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 ith 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 ith domino set up a sub-problem that couldn't be solved recursively, we know the ith domino will **never** be part of any solution. That means we don't need to re-`insert` it.

```
static bool chainExistsRecOpt(const 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.

Problem 3: Big O [Cynthia Lee and Ilan Goodman]

For each pair, say whether $f(n)$ is $O(g(n))$, or $g(n)$ is $O(f(n))$, or both.

a) $f(n) = n \log n + 5n$	$g(n) = 3n \log n$	both
b) $f(n) = 10 \log_{10} n$	$g(n) = 2 \log_2 n$	both
c) $f(n) = 1.01^n$	$g(n) = 1000n^3$	g(n) is O(f(n))
d) $f(n) = 1.6^n$	$g(n) = 2^n$	f(n) is O(g(n))
e) $f(n) = (\log n)^3$	$g(n) = n^{1/5}$	f(n) is O(g(n))
f) $f(n) = 7n \log n^2$	$g(n) = \begin{cases} 10n^2, & n < 100 \\ 100n, & n \geq 100 \end{cases}$	g(n) is O(f(n))