CS106B Handout 14
Summer 2011 July 5th, 2011

# Boggle Part I: Short Recursion Problems

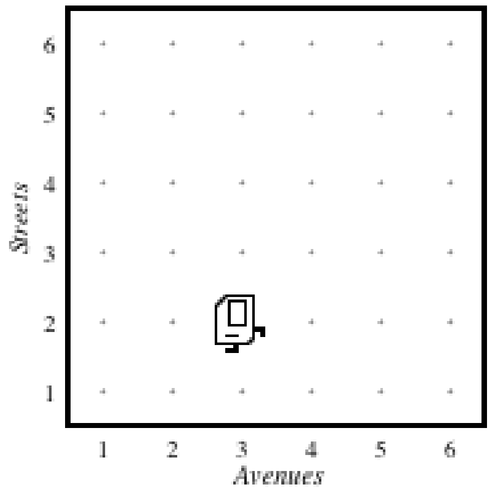
Assignment 3 is going out in two parts: this one, which has you implement a few short recursion problems and submit them for feedback, and a larger one, which has you implement the game of Boggle. Both parts are required, but you're to complete and submit solutions for the problems described in this handout first, and then move on to the larger assignment—one that has you implement the game of Boggle—afterwards.

**Solutions to Boggle Part 1: Monday, July 11th at 4:00 p.m.**
**Solution to Boggle Part 2: Thursday, July 14th at 4:00 p.m.**

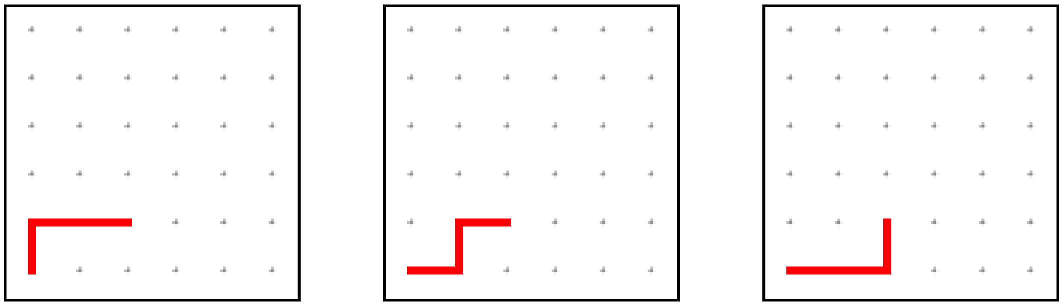
For the three warm-up exercises, we specify the function prototype. **Your function must exactly match that prototype** (same name, same arguments, same return type). Your function must use recursion; even if you can come up with an iterative alternative, we insist on a recursive formulation! Also, note that the Boggle assignment is much more involved than these warm-up problems, so don't be left with the impression that you somehow need six calendar days to complete these two and just four for Boggle. In practice, you'll want to press through these problems fairly soon and move on to Boggle as soon as possible. Thinks of these three problems and the Boggle portion of the assignment as one big assignment, and consider the completion of these three problems to be a milestone that needs to be completed by next Monday.

## Problem 1. Karel goes home

As most of you know, Karel the Robot lives in a world composed of streets and avenues laid out in a regular rectangular grid that looks like this:



Suppose that Karel is sitting on the intersection of 2nd Street and 3rd Avenue as shown in the diagram and wants to get back to the origin at 1st Street and 1st Avenue. Even if Karel wants to avoid going out of the way, there are still several equally short paths. For example, in this diagram there are three possible routes, as follows:



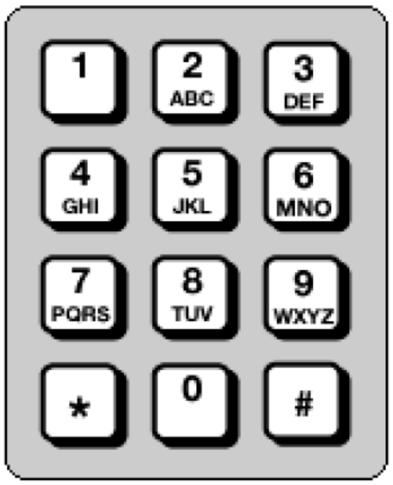
Your job in this problem is to write a recursive function

```
int CountPaths(int street, int avenue)
```

that returns the number of paths Karel could take back to the origin from the specified starting position, subject to the condition that Karel doesn't want to take any unnecessary steps and can therefore only move west or south (left or down in the diagram).

**2. Cell phone mind reading**

Entering text using a phone keypad is problematic, because there are only 10 digits for 26 letters. As a result, each of the digit keys (other than 0 and 1, which could not be part of telephone exchange prefixes until about thirty years ago) is mapped to several letters, as shown in the following diagram:



Some cell phones use a "multi-tap" user interface, in which you tap the 2 key once for a, twice for b, and three times for c, which can get tedious. A streamlined alternative is to use a predictive strategy in which the cell phone guesses which of the possible letter you intended, based on the sequence so far and its possible completions.

For example, if you type the digit sequence 72, there are 12 possibilities: pa, pb, pc, qa, qb, qc, ra, rb, rc, sa, sb, and sc. Only four of these—pa, ra, sa, and sc—seem promising because they are prefixes of common English words like party, radio, sandwich, and scanner. The others can be ignored because there are no common words that begin with those sequences of letters. If the user enters 9956, there are 144 (4 x 4 x 3 x 3) possible letter sequences, but you can be assured the user meant xylo since that is the only sequence that is a prefix of any English words.

Write a function **`void ListCompletions(string digits, Lexicon & lex);`** that prints all words from the lexicon that can be formed by extending the given digit sequence. For example, calling **`ListCompletions("72547", english)`** should generate the following sample run:

```
palisade
palisaded
palisades
palisading
palish
rakis
rakish
rakishly
rakishness
sakis
```

If your only concern was getting the answer, the easiest way to solve this problem would be to iterate through the words in the lexicon and print out every one that matches the specified digit string. That solution requires no recursion and very little thinking. Your managers, however, recognize that looking through every word in the dictionary is slow, and they insist that your code use the lexicon structure only to test whether a given string is a word or a prefix of an English word. With that restriction, you need to figure out how to generate all possible letter sequences from the string of digits. That task is easiest to solve recursively.

Note that this problem has two recursive pieces that require similar, but not identical, code. First you must explore converting the digit sequence into letters. Then, you need to extend that prefix recursively in an attempt to build words (since you're not allowed to go through the lexicon word by word to find all words with a given prefix). In the first case, the choices for the letters are constrained by the digit-to-letter mapping. In the second, what are the possible choices for letters that could be used to extend the sequence to build a completion? How can you use recursion to explore those choices?

**Problem 3: Making Change**
Everyone who's worked a cash register (that is to say, me, at least) has had to deal with the intellectually stimulating task of making change. The customer is owed $.90, so do I give her 9 dimes? 3 quarters, a dime, and a nickel? 90 pennies? We're used to 25, 10, 5, and 1 cent denominations, but how might I give change if coins are worth 20, 13, 4, and 1 cents instead?

There are clearly many different configurations of coins that will work, but let's say that we're interested in the combination that uses the fewest coins. One approach would be to use as many high value coins (i.e. quarters) as possible, then move on to use dimes for what is leftover, then nickels and finally pennies if needed. This type of algorithm is known as a greedy algorithm, since at any given moment it makes the choice that looks best at the moment, the hope being that the locally optimal choice will lead to a globally optimal solution. However, what if I had no nickels in my change drawer and was trying to make $.31? The greedy solution chooses 1 quarter and 6 pennies, which is worse than the optimal solution of 3 dimes and 1 penny. Clearly we need something even smarter to find the truly optimal arrangement. Recursion to the rescue!

Write a function **`MakeChange`** that takes an amount along with the **`Vector`** of available coin values. You can assume you have as many of each coin as you want (i.e. if your cash drawer has pennies, it has an infinite supply of them). The function should return the minimum number of coins required that sum to the given amount. If the amount cannot be made (for example, if you try to make $.31 and have no pennies), the function should return -1. You do not have to print or return the coin combination, just return the minimum number of coins in the optimal configuration.
There are several different ways to recursively decompose this problem. Let your creativity lead you toward the one that makes the most sense to you.

```
int MakeChange(int amount, Vector<int>& denominations)
```

**Thoughts on recursion**

Recursion is a tricky topic, so don't be dismayed if you can't immediately sit down and code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. You will need to depend on a recursive "leap of faith" to write the solution in terms of a problem you haven't solved yet. Be sure to take care of your base case(s) lest you end up in infinite recursion.

The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you'll be in much better shape for the second half of the assignment.