Eric Roberts
CS106B

Handout #20
April 17, 2009

Assignment #3—Recursion

Parts of this handout were written by Julie Zelenski and Jerry Cain.

Due: Friday, April 24

This week's assignment consists of four recursive functions to write at varying levels of difficulty. Learning to solve problems recursively can be challenging, especially at first. We think it's best to practice in isolation before adding the complexity of integrating recursion into a larger program. The recursive solutions to most of these problems are quite short—typically less than a dozen lines each. That doesn't mean you should put this assignment off until the last minute though—recursive solutions can often be formulated in a few concise, elegant lines but the density and complexity that can be packed into such a small amount of code may surprise you.

The assignment begins with two warm-up problems, for which we provide hints and solutions. You don't need to hand in solutions to the warm-ups. We recommend you first try to work through them by yourself. If you get stuck, ask for help and/or take a look at our solutions posted on the web site. You can also freely discuss the details of the warm-up problems (including sharing code) with other students. We want everyone to start the problem set with a good grasp on the recursion fundamentals and the warm-ups are designed to help. Once you're working on the assignment problems, we expect you to do your own original, independent work (but as always, you can ask the course staff if you need a little help).

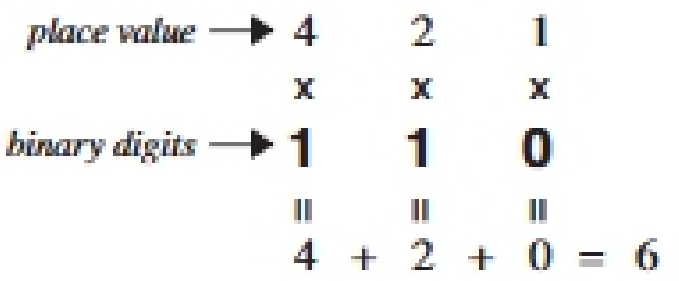
The first few problems after the warm-up exercises include some hints about how to get started, the later ones you will need to work out the recursive decomposition for yourself. It will take some time and practice to wrap your head around this new way of solving problems, but once you "grok" it, you'll be amazed at how delightful and powerful it can be.

Warm-up problem 0a. Binary encoding (Chapter 6, exercise 9, page 230)

Inside a computer system, integers are represented as a sequence of bits, each of which is a single digit in the binary number system and can therefore have only the value 0 or 1. With N bits, you can represent 2^N distinct integers. For example, three bits are sufficient to represent the eight (2^3) integers between 0 and 7, as follows:

0 0 0	→	0
0 0 1	→	1
0 1 0	→	2
0 1 1	→	3
1 0 0	→	4
1 0 1	→	5
1 1 0	→	6
1 1 1	→	7

Each entry in the left side of the table is written in its standard binary representation, in which each bit position counts for twice as much as the position to its right. For instance, you can demonstrate that the binary value **110** represents the decimal number 6 by following the logic shown in the following diagram:



Write a recursive function **`GenerateBinaryCode(nBits)`** that generates the bit patterns for the standard binary representation of all integers that can be represented using the specified number of bits. For example, calling **`GenerateBinaryCode(3)`** should produce the following output:

```
000
001
010
011
100
101
110
111
```

Warm-up problem 0b. Subset sum

The subset sum problem is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to that target number. For example, given the numbers {3, 7, 1, 8, -3} and the target sum 4, the subset {3, 1} sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this function is

```
bool SubsetSumExists(Vector<int> & nums, int targetSum);
```

Remember that many recursive problems are variations on the same age-old themes. Consider how this problem is related to the **`GenerateSubsets`** function from lecture. Take a look at that code first. You should be able to fairly easily adapt it to operate on a vector of numbers instead of a string. Note that you are not asked to print the numbers in the sum, just return a Boolean result. You will likely need a wrapper function to pass additional state through the recursive calls, which means that you need to think carefully about what information you need to track as you try various combinations.

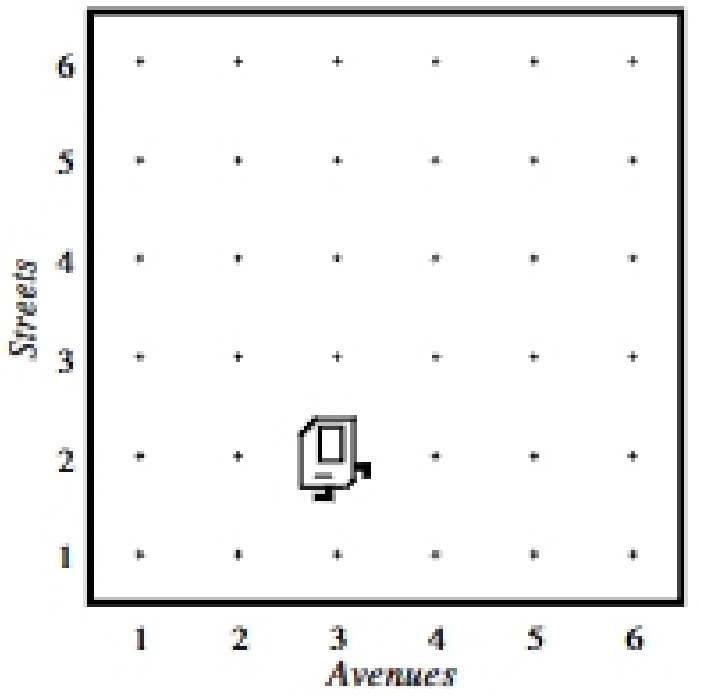
Once you have a basic version of the function working, here are some other variations to give you even more practice.

- The recursive decomposition of **`GenerateSubsets`** considers each element in turn and generates the complete list of subsets by generating all subsets that *include* that element along with all subsets that *exclude* it. This pattern comes up often in recursive programming and is called the **inclusion/exclusion pattern**. That strategy, however, is not the only one you could use to solve this problem. An alternative recursive decomposition repeatedly chooses one of the remaining elements to add to the subset then calls itself recursively on the remaining set. Rewrite the **`SubsetSumExists`** function to use this alternative strategy. One special issue with this version is that you don't want to try the same subset more than once, so be careful to be sure each possible subset is examined at most once (*i.e.*, after trying **`ABC`** there is no reason to try **`CAB`** and **`BCA`**).

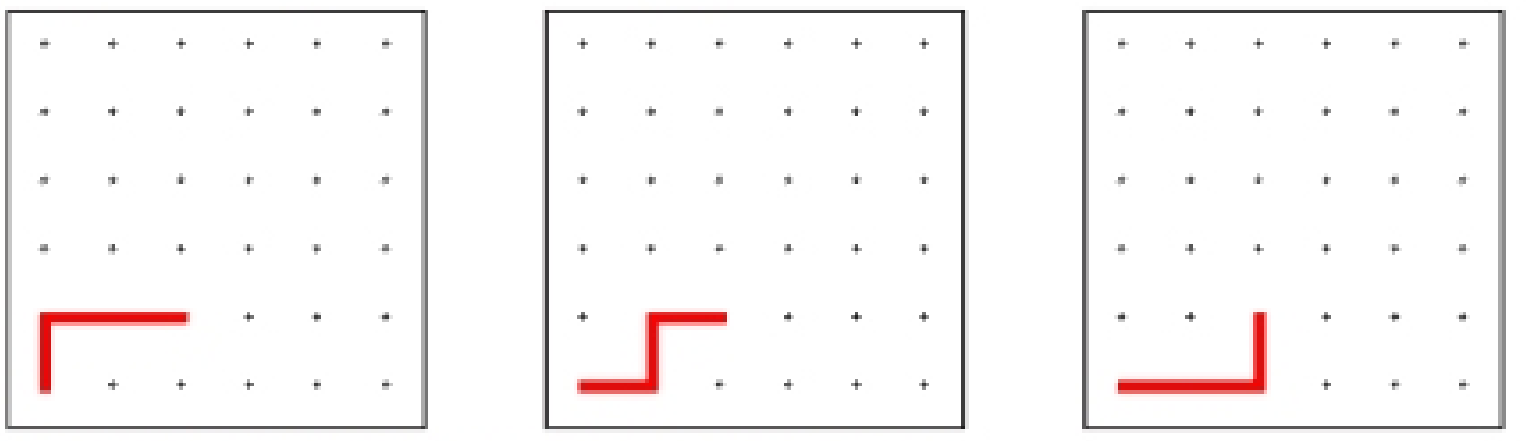
- How could you change the function to print the subset members that sum to the target when successful? One method is to store the numbers into another vector that is updating during the recursive calls. Another approach doesn't add any new data structures or store the numbers, it just prints the chosen members when unwinding from the recursive calls.
- How could you change the function to report not just whether any such subset exists, but the count of all such possible subsets? For example, in the set shown earlier, the subset {7, -3} also sums to 4, so there are two possible subsets for target 4. This somewhat more difficult problem is included in the text as exercise 11 on page 232.

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 always wants to take one of the shortest possible routes and can therefore only move west or south (left or down in the diagram).