Eric Roberts
CS106B

Handout #21
January 28, 2013

Assignment #3—Recursion

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

Due: Monday, February 4

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).

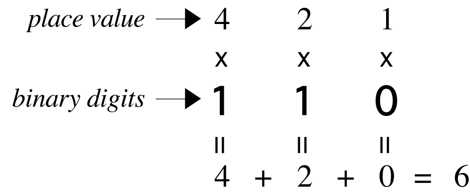
The first few problems after the warm-up exercises include some hints about how to get started, the later ones you will need to work out the recursive decomposition for yourself. It will take some time and practice to wrap your head around this new way of solving problems, but once you "grok" it, you'll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

Warm-up problem 0a. Binary encoding

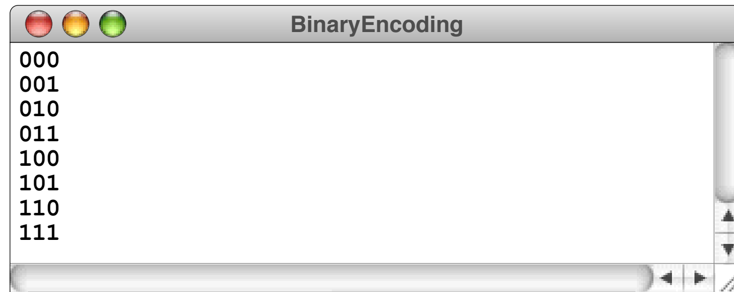
Inside a computer system, integers are represented as a sequence of bits, each of which is a single digit in the binary number system and can therefore have only the value 0 or 1. With N bits, you can represent 2^N distinct integers. For example, three bits are sufficient to represent the eight (2^3)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

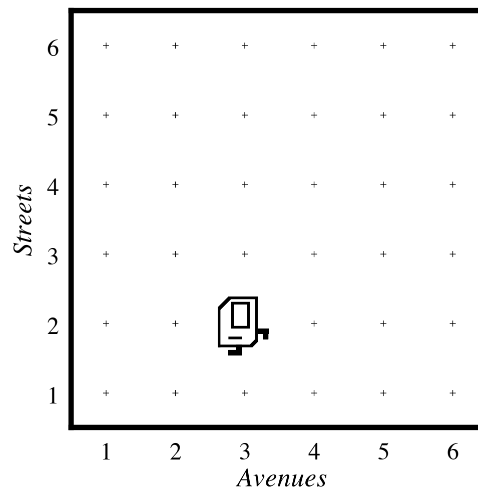


Write a recursive function **`generateBinaryCode(nBits)`** that prints out the bit patterns for the standard binary representation of all integers that can be represented using the specified number of bits. For example, calling **`generateBinaryCode(3)`** should produce the following output:

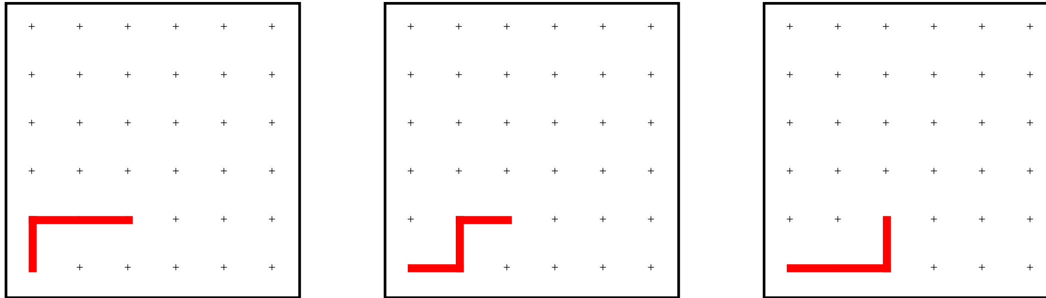


Warm-up problem 0b. Karel Goes Home

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).

Problem 1. Balancing parentheses

In the class discussion of stacks, I went through an example (which comes from Chapter 5, exercise 14, page 255, if you want to see it in the text) that uses stacks to check whether the bracketing operators in an expression are properly nested. You can do the same thing recursively.

Write a function

```
bool isBalanced(string exp);
```

that takes a string consisting only of parentheses, brackets, and curly braces and returns `true` or `false` according to whether that expression has properly balanced operators. Your function should operate recursively by making use of the recursive insight that such a string is balanced if and only one of the following conditions holds:

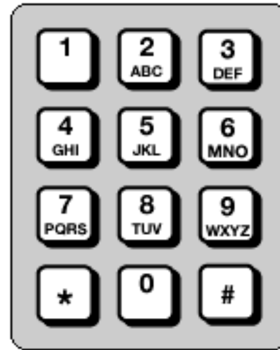
- The string is empty
- The string contains `"()"`, `"[]"`, or `"{}"` as a substring and is balanced if you remove that substring.

For example, you can show that the string `"[(){}]"` is balanced by the following recursive chain of reasoning:

```
isBalanced("[(){}]") is true because
  isBalanced("[{}]") is true because
    isBalanced("[]") is true because
      isBalanced("") is true because the argument is empty.
```

Problem 2. Cell phone mind-reading (Chapter 8, exercise 12, page 383)

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:



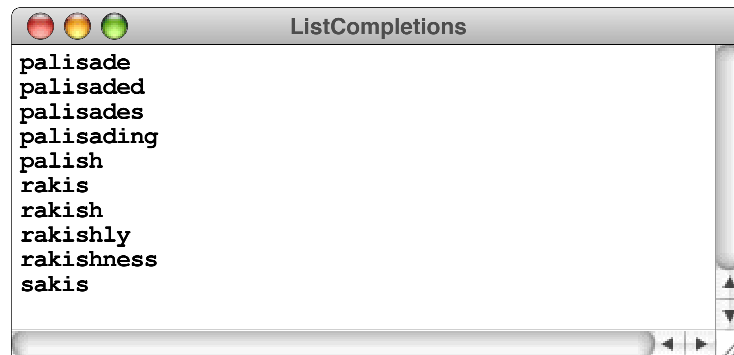
Some cell phones use a "multi-tap" user interface, in which you tap the 2 key once for **a**, twice for **b**, and three times for **c**, which can get tedious. A streamlined alternative is to use a predictive strategy in which the cell phone guesses which of the possible letter you intended, based on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

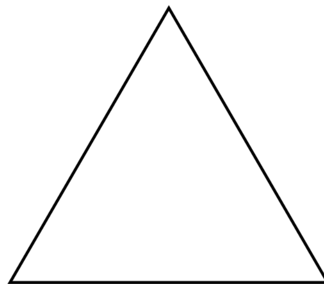


If your only concern were getting the answer, the easiest way to solve this problem would be to iterate through the words in the lexicon and print out every one that matches the specified digit string. That solution requires no recursion and very little thinking. Your managers, however, believe that 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.

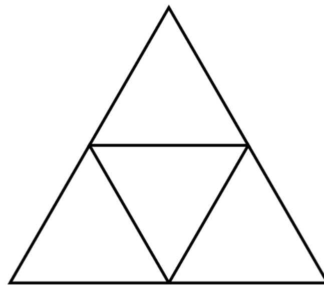
Note that this problem has two recursive pieces that require similar, but not identical, code. First you must explore converting the digit sequence into letters. Then, you need to extend that prefix recursively in an attempt to build words (since you're not allowed to go through the lexicon word by word 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. Sierpinski Triangle (Chapter 8, exercise 18, page 388)

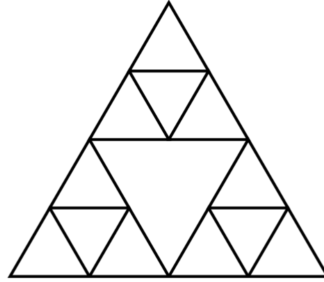
If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the ***Sierpinski Triangle,*** named after its inventor, the Polish mathematician Wacław Sierpiński (1882–1969). The order-0 Sierpinski Triangle is an equilateral triangle:



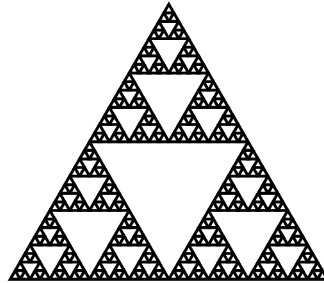
To create an order-N Sierpinski Triangle, you draw three Sierpinski Triangles of order $N-1$, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



The downward-pointing triangle in the middle of this figure is not drawn explicitly, but is instead formed by the sides of the other three triangles. That area, moreover, is not recursively subdivided and will remain unchanged at every level of the fractal decomposition. Thus, the order-2 Sierpinski Triangle has the same open area in the middle:



If you continue this process through three more recursive levels, you get the order-5 Sierpinski Triangle, which looks like this:



Write a program that asks the user for an edge length and a fractal order and draws the resulting Sierpinski Triangle in the center of the graphics window.

Problem 4. Stock cutting (Chapter 9, exercise 10, page 426)

Suppose that you have been assigned the job of buying the plumbing pipes for a construction project. Your foreman gives you a list of the varying lengths of pipe needed, but the distributor sells stock pipe only in one fixed size. You can, however, cut each stock pipe in any way needed. Your job is to figure out the minimum number of stock pipes required to satisfy the list of requests, thereby saving money and minimizing waste.

Your job is to write a recursive function

```
int cutStock(Vector<int> & requests, int stockLength);
```

that takes two arguments—a vector of the lengths needed and the length of stock pipe that the distributor sells—and returns the minimum number of stock pipes needed to service all requests in the vector. For example, if the vector contains [4, 3, 4, 1, 7, 8] and the stock pipe length is 10, you can purchase three stock pipes and divide them as follows:

Pipe 1: 4, 4, 1
Pipe 2: 3, 7
Pipe 3: 8

Doing so leaves you with two small remnants left over. There are other possible arrangements that also fit into three stock pipes, but it cannot be done with fewer.

Cutting stock is representative of the general problem of minimizing the consumption of a scarce resource. Variants come up in many situations: optimizing file storage on

removable media, assigning commercials to station breaks, allocating blocks of computer memory, or minimizing the number of processors for a set of tasks. Cloth, paper, and sheet metal manufacturers use a two-dimensional version of the problem to cut pieces of material from standard-sized sheets. Shipping companies use a three-dimensional version to pack containers or trucks.

This one is tricky, and we expect you will mull over it for a while before you have a solution. Once you have conquered this problem, however, you can proudly say you have earned your CS106 Recursion Merit Badge!

Here are a few hints and specifications:

- You may assume that all the requests are positive and do not exceed the stock length. In the worst case, you will need *N* stock pipes, where *N* is the size of the vector.
- You do not need to report how to cut the pipes, just the number of stock pipes needed.
- It almost certainly makes sense to write **`cutStock`** as a wrapper function that takes additional arguments. Think about what information you need in the general case.
- There are several different approaches for decomposing this problem and for managing the state. As a general hint, given the current state and a request, what options do you have for satisfying that request and how does choosing that option change the state? Making a choice should result in a smaller, similar subproblem that can be recursively solved. How can you that solution to solve the larger problem?
- You cannot solve this problem using a "greedy" algorithm that fills each new request by finding the first remnant in which it fits, even if you sort the requests first. To convince yourself of this fact, try to come up with examples in which the greedy algorithm fails.

General notes

- Please note that we have given you function prototypes for each of the four problems. The functions you write must match those prototypes *exactly*—the same names, the same arguments, the same return types. Your functions must also use recursion; even if you can come up with an iterative alternative, we insist on a recursive formulation!
- For each problem on this assignment, you will want to test your function thoroughly to verify that it correctly handles all the necessary cases. For example, for the binary warm-up, you might write a main program to call your function for selected values of **`nBits`** or one that lets the user enter those values. Such testing code is encouraged and in fact, necessary, if you want to be sure you have handled all the cases. You can leave your testing code in the file you submit, no need to remove it.
- Recursion is a tricky topic, so don't be dismayed if you can't immediately sit down and solve these problems. Take time to figure out how each problem is recursive in nature and how you could formulate a solution given the solution to a smaller, simpler subproblem. You will need to depend on the "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 cases lest you end up in infinite recursion.
- Once you learn to think recursively, recursive solutions to problems seem very intuitive and direct. Spend some time on these problems and you'll be much better

prepared for the next assignment where you implement fancy recursive algorithms at the heart of a sophisticated program.

Extensions to the assignment

For most of you, the four problems that form the body of this assignment will be more than enough to keep you busy. Those of you who manage to find the correct recursive insights quickly may find yourself with extra time because the code required to implement these problems is not that long. But if you're really jazzed on recursion or just want some more practice, feel free to play with some other recursive code. There are tons of neat problems out there that lend themselves to a recursive solution. Here are a few that we've suggested over the years, and we're sure you will think of others!

- Find the longest hidden word within a string (*i.e.*, a word that can be made from a permuted subset of the letters).
- Write a simple spell-checker that finds legal words that are at most *N* "edits" from a misspelled word, where an edit is an insertion, deletion, or exchange of a letter.
- Write a program that solves simple substitution ciphers by guessing, using the lexicon to prune bad choices, and backtracking when needed.
- Write a function that matches filename patterns involving wildcards. The most common wildcard patterns are `*`, which matches any sequence of characters, and `?`, which matches any single character. For example, the pattern `*.tmp` matches all filenames that end with the extension `.tmp` and the pattern `test??` matches any filename that consists of `test` followed by two arbitrary characters.
- Write a program that use recursive backtracking to solve any of the classic puzzles that appear in newspapers, such as jumbles, sudoku, word search, and so on.
- Explore the many wonderful possibilities available for graphical recursion. The exercises in Chapter 7 offer a few ideas, but there are many wonderful examples to explore in the world of fractal mathematics. Many fractals can be described using a nifty general-purpose facility called a Lindenmayer System. There are some intriguing pictures at `http://mathworld.wolfram.com/LindenmayerSystem.html` and a good tutorial at `http://spanky.triumf.ca/www/fractint/lsys/tutor.html`.