Assignment 3: Recursion!

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

This week's assignment consists of four recursive functions of varying flavor and 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 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).

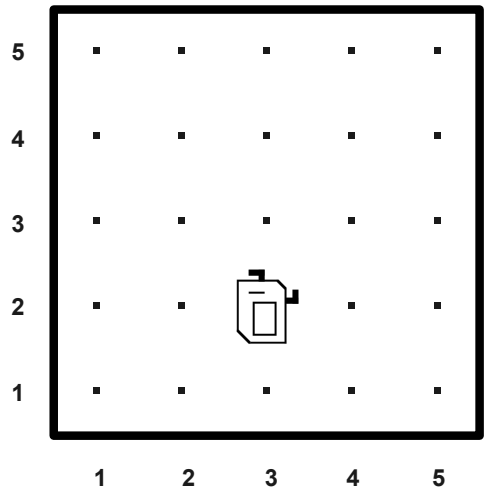
The first few problems after the warm-up exercises include some hints about how to get started. For the later ones, you will need to work out the recursive decomposition yourself. It will take some time and practice to wrap your head around this new way of solving problems, but once you gain an intuition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

Due Monday, April 30 at 10:00AM

http://www.toothpastefordinner.com/index.php?date=022410

Warm-up Problem 0A: 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:

- Move left, then left, then down.
- Move left, then down, then left.
- Move down, then left, then left.

Your job in this problem is to write a recursive function

```
int numPathsHome(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).

Warm-up Problem 0B: Shuffling a Deck

How might the computer shuffle a deck of cards? It turns out that this problem is a bit more complex than it might seem, and while it's easy to come up with algorithms that randomize the order of the cards, only a few algorithms will do so in a way that ends up generating a uniformly-random reordering of the cards.

One simple algorithm for shuffling a deck of cards is based on the following idea – we can choose a card at random from the deck, shuffle the remaining cards, and then put the card that we picked on top of the deck. Assuming that we choose the card that we put on top randomly, this ends up producing a random shuffle of the deck.

Write a function

```
string randomShuffle(string input)
```

that accepts as input a string, then returns a random permutation of the elements of the string using the above algorithm. Your algorithm should be recursive and not use any loops (**`for`**, **`while`**, etc.)

The remaining problems in this assignment should be submitted for credit.

Problem One: Even Words

An *even word* is a word that contains an even number of copies of every letter. For example, the word "tattletale" is an even word, since there are four copies of 't' and two copies of 'a,' 'e,' and 'l.' Similarly, "appeases" and "arraigning" are even words. However, "banana" is not an even word, because there is just one 'b' and three copies of 'a.'

Write a function

```
bool isEvenWord(string word)
```

that accepts as input a string representing a single word and returns whether or not that word is an even word. Your solution should be recursive and must not use any loops (e.g. **`while`**, **`for`**). As a hint, this problem has a beautiful recursive decomposition:

- The empty string is an even word, since it has 0 copies of every letter.
- Otherwise, a word is an even word if there are at least two copies of the first letter and the word formed by removing two copies of the first letter is itself an even word.

For example, we can see that the word "appeases" is an even word using the following logic:

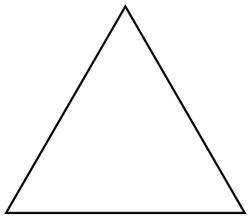
"**a**ppe**a**ses" is an even word, because
"**pp**eses" is an even word, because
"**e**s**e**s" is an even word, because
"**ss**" is an even word, because
"" is an even word.

We also know that "tattletale" is even via the following reasoning:

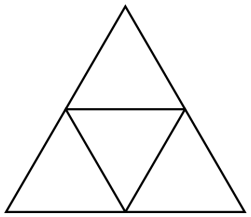
"**t**a**t**tletale" is an even word, because
"**a**tlet**a**le" is an even word, because
"**t**le**t**le" is an even word, because
"**l**e**l**e" is an even word, because
"**ee**" is an even word, because
"" is an even word.

Problem Two: The Sierpinski Triangle (Chapter 8, exercise 18, page 388)

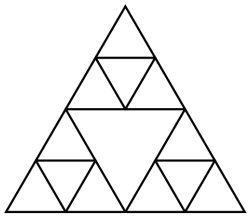
If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the ***Sierpinski Triangle,*** named after its inventor, the Polish mathematician Wacław Sierpiński (1882 – 1969). The order-0 Sierpinski Triangle is an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



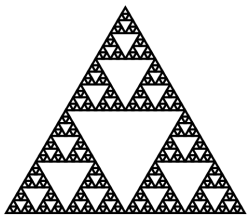
To create an order N Sierpinski Triangle, you draw three Sierpinski Triangles of order $N-1$, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



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. You may find the **`drawPolarLine`** function from the graphics package useful here, and may want to take advantage of the fact that **`drawPolarLine`** returns a **`GPoint`** indicating where the line ends.

Problem Three: Inverse Genetics

In Section Handout 2, we explored how RNA encodes for proteins. If you'll recall, RNA strands are sequences of the four *base pairs*, which we represent with the letters A, C, U, and G. Three consecutive base pairs is called a *codon*, which encodes a specific *amino acid*. A protein is then a sequence of amino acids. Just as the letters A, C, U, and G encode the base pairs in an RNA strand, there are a set of letters used to encode specific amino acids. For example, N represents the amino acid asparagine, Q represents glutamine, I represents isoleucine, etc. Thus if we had the protein consisting of glutamine, then glutamine, then isoleucine, then asparagine, we would write it as QQIN.

Each codon encodes an amino acid, but some codons encode the same amino acid. For example, the codons UUA, UUG, CUU, CUC, CUA, and CUG all encode for leucine. We can represent the mapping from amino acids to the set of codons that represent that amino acid as a **`Map<char, Set<string> >`**, where the keys are the one-letter codes for the amino acids and the **`Set<string>`** represents the set of codons that encode the particular amino acid. As a sampling, this **`Map`** would contain

L { UUA, UUG, CUU, CUC, CUA, CUG }

S { AGC, AGU, UCA, UCC, UCG, UCU }

K { AAA, AAG }

W { UGG }

...

Because there are multiple codons that represent a single amino acid, there might be many different RNA strands that all encode the same protein. For example, the protein with amino acid sequence KWS could be represented by any of the following RNA strands:

AAAUGGAGU
AAAUGGAGC
AAAUGGUCA
AAAUGGUCC
AAAUGGUCG
AAAUGGUCU
AAGUGGAGC
AAGUGGAGU
AAGUGGUCA
AAGUGGUCC
AAGUGGUCG
AAGUGGUCU

Your task is to write a function that, given a protein (represented as a string of letters for its amino acids) and the mapping from amino acids to codons, returns the set of all possible RNA strands that could encode that particular protein. Specifically, your function should have this signature:

```
Set<string> allRNAStrandsFor(string protein, Map<char, Set<string> >& codons)
```

You can assume that all the letters in the protein are valid amino acids, so you will never find a letter in the protein that isn't in the **`Map`**. Also, don't worry about handling start and stop codons as you did in section. Instead, just construct all sequences of codons that directly translate to the given amino acids.

When testing, be aware that if you run this function on a long string, you might get back a **huge** number of strings!

Problem Four: Universal Health Coverage

Suppose that you are the Minister of Health for a small country. You are interested in building a set of state-of-the-art hospitals to ensure that every city in your country has access to top-quality care. Your Hospital Task Force has come to you with a set of proposed locations for these hospitals, along with the cities that each one would service.

Although you are interested in providing excellent health care to all, you do not have the funds to build an unlimited number of hospitals. Using your newfound skills with recursion, you are interested in seeing whether or not it is possible to provide quality health care to every city given a limit on the number of hospitals you can construct.

Suppose that you are given a `Set<string>` representing the names of all of the cities your country. You are also provided with a list of all the proposed hospital locations, each of which is represented by the set of cities that the hospital could service. Given your funding restrictions, you can purchase at most `numHospitals` total hospitals. Write a function

```
bool canOfferUniversalCoverage(Set<string>& cities,
                               Vector< Set<string> >& locations,
                               int numHospitals,
                               Vector< Set<string> >& result)
```

that accepts as input the set of all cities and a list of the cities that would be covered by each hospital, along with the maximum number of hospitals that can be constructed, and returns whether or not it is possible to provide coverage to all cities using the limited number of hospitals. If so, your function should update the `result` parameter to contain one such choice of hospitals. If not, you do not need to use the `result` parameter.

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 Karel warm-up, you might write a main program to call your function for selected values of `numPathsHome` 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 available on the web; check out `http://mathworld.wolfram.com/LindenmayerSystem.html` for one possible option.