Assignment 3: Recursion!

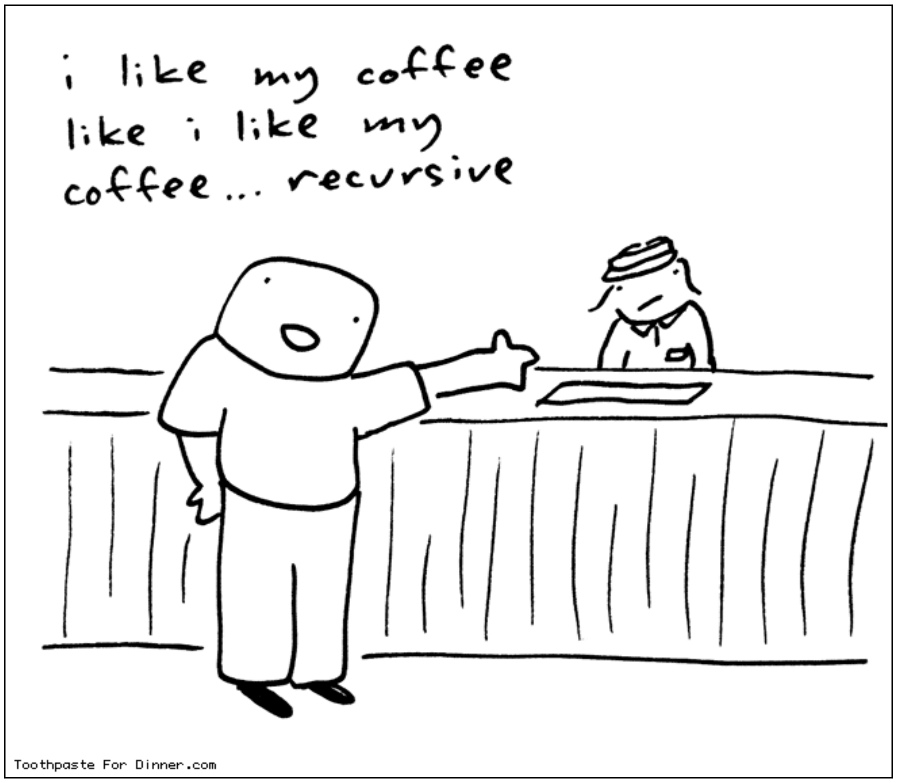
Parts of this handout were written by Julie Zelenski, Jerry Cain, and Eric Roberts.

This 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 around 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.

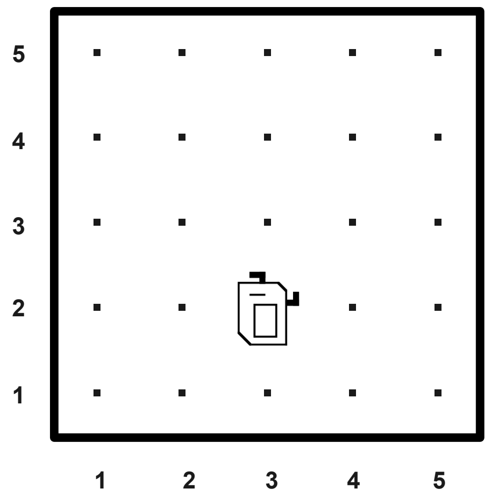
Due Thursday, July 18 at 11:00AM


i like my coffee like i like my coffee... recursive
Toothpaste For Dinner.com

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: Random Subsets

As we saw in lecture, there are 2^n possible subsets of a set of n elements. That's a *lot* of subsets, and it can take a very long time to list them all off. What if instead of generating all possible subsets of a master set, we just generated a few random subsets? That way, we can get a rough sense for what the subsets look like.

We can generate a random subset from a master set by visiting every element of the set, one after the other, and flipping a fair coin to decide whether or not to include that element in the resulting subset. As long as the coin tosses are fair, this produces a uniformly-random subset of the master set.

Write a function

```
Set<int> randomSubsetOf(Set<int>& s);
```

that accepts as input a **`Set<int>`** and returns a random subset of that master set. 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: Subsequences

If *S* and *T* are strings, we say that *S* is a *subsequence* of *T* if all of the letters of *S* appear in *T* in the same relative order that they appear in *S*. For example, the string **`pin`** is a subsequence of the string **`programming`**, and the string **`singe`** is a subsequence of **`springtime`**. However, **`steal`** is not a subsequence of **`least`**, since the letters are in the wrong order, and **`i`** is not a subsequence of **`team`** because there is no **`i`** in **`team`**. The empty string is a subsequence of every string, since all 0 characters of the empty string appear in the same relative order in any arbitrary string.

Write a function

```
bool isSubsequence(string text, string subseq)
```

that accepts as input two strings and returns whether the second string is a subsequence of the first. Your solution should be recursive and must not use any loops (e.g. **`while`**, **`for`**).

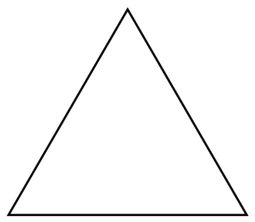
This problem has a beautiful recursive decomposition. As a hint, try thinking about the following:

- What strings are subsequences of the empty string?
- What happens if the first character of the subsequence matches the first character of the text? What happens if it doesn't?

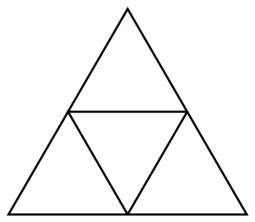
You can assume that the strings are case-sensitive, so **`AGREE`** is not a subsequence of **`agreeable`**.

Problem Two: The Sierpinski Triangle

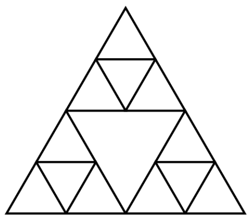
If you search the web for fractal designs, you will find many intricate wonders beyond fractal tree we coded up in lecture. One of these is the ***Sierpinski Triangle,*** named after its inventor, the Polish mathematician Wacław Sierpiński (1882 – 1969). The order-0 Sierpinski Triangle is an equilateral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



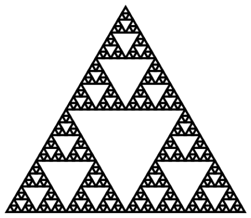
To create an order *N* Sierpinski Triangle, you draw three Sierpinski Triangles of order $N-1$, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



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`** member function of **`GWindow`** 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 proteins. If you'll recall, RNA strands are sequences of the four *nucleotides*, which we represent with the letters `A`, `C`, `U`, and `G`. Three consecutive nucleotides form 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 multiple codons might represent the same amino acid, there might be many different RNA strands that all encode the same protein (that is, the same sequence of amino acids). For example, the protein with amino acid sequence `KWS` could be represented by any of the following RNA strands:

```
AAAUGGAGU      AAAUGGUCC      AAGUGGAGC      AAGUGGUCC
AAAUGGAGC      AAAUGGUCG      AAGUGGAGU      AAGUGGUCG
AAAUGGUCA      AAAUGGUCU      AAGUGGUCA      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, prints out all possible RNA strands that could encode that particular protein. Specifically, your function should have this signature:

```
   void listAllRNAStrandsFor(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`. 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. Your function can list the RNA strings in any order, as long as every strand is listed exactly once.

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.

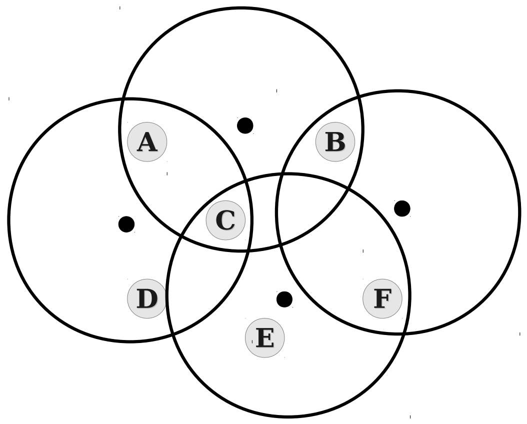
Although you are interested in providing excellent health care to all, you do not have the funds to build an unlimited number of hospitals. Using your newfound skills with recursion, you are interested in seeing whether or not it is possible to provide quality health care to every city given a limit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

Suppose that you are given a `Set<string>` representing the names of all of the cities in 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.

As an example, consider the country below, where each city is represented by a letter and each potential hospital location is represented with a black dot:



Here, each hospital is can cover all the cities within their circle of coverage. This would represented as follows:

- **`cities = { "A", "B", "C", "D", "E", "F" }`**
- **`locations = 〈 {"A", "B", "C"}, {"A", "C", "D"}, {"B", "F"}, {"C", "E", "F"} 〉`**

The topmost hospital would serve cities A, B, and C. The hospital on the left would cover A, C, and D. The hospital on the right covers just B and F, and the hospital on the bottom covers C, E, and F. If you can only purchase two hospitals, then there is no way to guarantee coverage to everyone. However, if you can purchase three hospitals, then you can cover everyone – purchase the top hospital to cover A, B, and C, the bottom hospital to cover C, E, and F, and the leftmost hospital to cover A, C, and D.

General notes

- 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, 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 program to call your **`numPathsHome`** function on inputs where you know the outputs in advance, 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; there's 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'll implement some other fancy recursive algorithms.

Possible Extensions

Want more to explore? Here are a few suggestions to help you get started:

- **Subsequences**: An interesting variation on this problem is the following: given a collection of strings, what is the smallest string *S* such that every string in the collection is a subsequence of *S*? Try seeing how you might solve this problem.
- **Sierpinski Triangle**: Although this assignment is all about recursion, if you're up for a challenge, try replacing the recursive version of your code to draw an order-*n* Sierpinski triangle with an *iterative* function that draws an order-*n* Sierpinski triangle.
- **Inverse Genetics**: Suppose you have an RNA strand that encodes a particular protein. A *point mutation* is a chance to an RNA strand where one letter is replaced with a different letter. Write a program that computes the probability that a single point mutation to that strand will cause the strand to encode a different protein.
- **Universal Health Coverage**: Suppose your objective now is to provide coverage to the maximum number of people. Update your function so that each city is annotated with a total population, then have your function find the maximum number of people that can be covered.