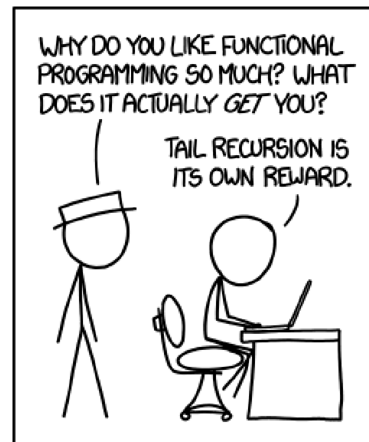
Assignment 3: Recursion!

This assignment consists of four recursive functions of varying flavor and difficulty. Learning to solve problems recursively can be challenging, especially at first. It's best to practice recursion 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, and then there's the added complexity of visualizing output using the SFML library.

The assignment begins with two warm-up problems, for which hints and solutions are provided. 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! You can also freely discuss the details of the warm-up problems (including sharing code) with other students.

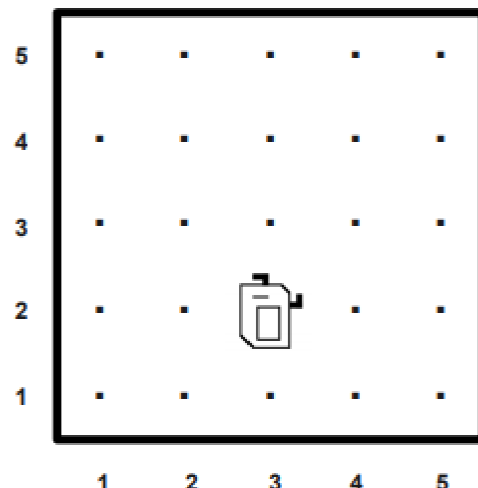
The warm-ups are designed to help you start the problem set with a good grasp on the recursion fundamentals. When you get to working on the assignment problems, do your own original, independent work (but as always, you can ask me or Jonas for a little help).

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 !



<u>Warm-up Problem 0A: Geoff Goes To Work</u>

As you may know, Geoff the Robot lives in a world composed of streets and avenues laid out in a regular rectangular grid as shown in the figure. Suppose that Geoff 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 Geoff wants to avoid going out of the way, there are still several equally short paths:



- 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 Geoff could take back to the origin from the specified starting position, subject to the condition that Geoff 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

There are 2^n possible subsets of a set of n elements (Two possibilities for each of the n members: in or out). That's growing exponentially, so it can take a very long time print them to a list. What if instead of generating all possible subsets, we just generate a few random subsets? That way, we can get a rough sense for what the subsets look like.
To generate a random subset from a set we visit every element of the set, one after the other, and flip 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 given set.

Write a function

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

that accepts as input a `set<int>` and returns a random subset of that set. Your algorithm should be recursive and not use any loops (`for, while,` etc.)

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` (groan). 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.

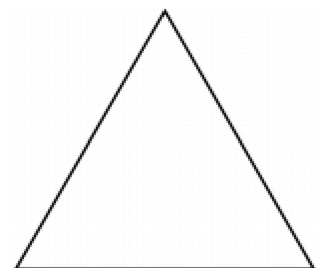
Your solution should be recursive and must not use any loops (e.g. `while, for`). This problem has a beautiful recursive decomposition. As a hint, try thinking about the following:

- 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?

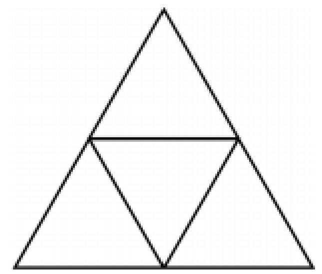
You can assume that the strings are case-sensitive, so `AGREE` is not a subsequence of `agreeable`.

Problem Two: The Sierpinski Triangle

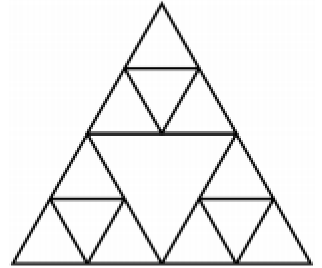
If you search the web for fractal designs, you will find many intricate wonders beyond fractal tree we will develope in lecture. One of these is the Sierpinski Triangle, named after its inventor, the Polish mathematician Wacław Sierpiński (1882 – 1969). The order-0 Sierpinski Triangle is an equilateral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



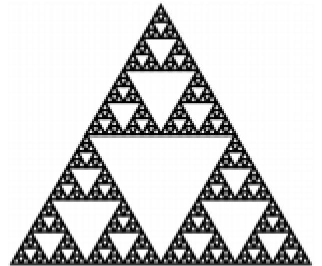
To create an order `N` Sierpinski Triangle, you draw three Sierpinski Triangles of order `N-1`, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



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. For this purpose it will be helpful to develop a `drawPolarLine()` function that meets the following specifications:

```
//drawPolarLine()
Point drawPolarLine(double x0, double y0, double r, double theta)
    //virtualinherited? (or not)
    /*Draws a line using polar coordinates that begins at the given (x,y) point and extends from there by the given angle and radius. Returns the end point where the line ends. */
```

You may want to take advantage of the fact that `drawPolarLine()`returns a `Point` indicating where the line ends.

Problem Three: Inverse Genetics

RNA strands are biological 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 ex-

ample, 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  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, 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. 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: A recursive puzzle

You have been given a puzzle consisting of a row of squares each containing an integer, like this:

The circle on the initial square is a marker that can move to other squares along the row. At each step in the puzzle, you may move the marker the number of squares indicated by the integer in the square it currently occupies. The marker may move either left or right along the row but may not move past either end. For example, the only legal first move is to move the marker three squares to the right because there is no room to move three spaces to the left.

The goal of the puzzle is to move the marker to the 0 at the far end of the row. In this configuration, you can solve the puzzle by making the following set of moves:

Starting position	**3**	6	4	1	3	4	2	5	3	0
Step 1: Move right	3	6	4	**1**	3	4	2	5	3	0
Step 2: Move left	3	6	**4**	1	3	4	2	5	3	0
Step 3: Move right	3	6	4	1	3	4	**2**	5	3	0
Step 4: Move right	3	6	4	1	3	4	2	5	**3**	0
Step 5: Move left	3	6	4	1	3	**4**	2	5	3	0
Step 6: Move right	3	6	4	1	3	4	2	5	3	**0**

Even though this puzzle is solvable—and indeed has more than one solution—some puzzles of this form may be impossible, such as the following one:

3	1	2	3	0

In this puzzle, you can bounce between the two 3's, but you cannot reach any other squares.

Write a function `bool Solvable(int start, Vector<int>& squares)` that takes a starting position of the marker along with the vector of squares. The function should return true if it is possible to solve the puzzle from the starting configuration and false if it is impossible.

You may assume all the integers in the vector are positive except for the last entry, the goal square, which is always zero. The values of the elements in the vector must be the same after calling your function as they are beforehand, (which is to say if you change them during processing, you need to change them back!)

Thoughts on recursion 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 sub-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.

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 If you fully understand and have elegant solutions to all of these problems, you're well on your way to recursion nirvana. 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's a few to consider:

- Find the longest word hidden within a string (i.e. word formed by permuted subset of the letters)
- Spell-checking suggestions – finding legal words that are at most N ”edits” from a misspelled word, where an edit is an insertion, deletion, or exchange of a letter
- Breaking simple substitution ciphers by guessing, using the lexicon to prune bad choices, and backtracking when needed
- Regular expression pattern matching that uses wildcards such as * to match any sequence of letters. A pattern such as a*a matches all words that start and end with a. • Many simple puzzles lend themselves to solvers that use recursive backtracking: jumble, sudoku, word search, and so on.
- Solve a maze problem via recursive backtracking.
- There are many cool varieties of recursive drawing to explore, the Koch snowflake, Sierpinski gaskets, fractal ferns, and more. These kind of recursive fractals can be described using a nifty general-purpose facility called a Lindenmayer System. There are some beautiful and intriguing pictures at http://mathworld.wolfram.com/LindenmayerSystem.html and a good tutorial at http://spanky.triumf.ca/www/fractint/lsys/tutor.html.

To turn in: Turn in one recursion.cpp source file that contains all assigned functions with exactly the prototypes given. Your source file can also contain testing code you used when developing your solutions. Keep a copy of your assignment to ensure that there is a backup in case your submission is lost or the electronic submission is corrupted. Computers often fail at the worst time!