Assignment 5: Thinking Recursively

The Word begets one,
one begets two,
two beget three,
and three beget everything.

— Laozi, Charter 42, Tao Te Ching

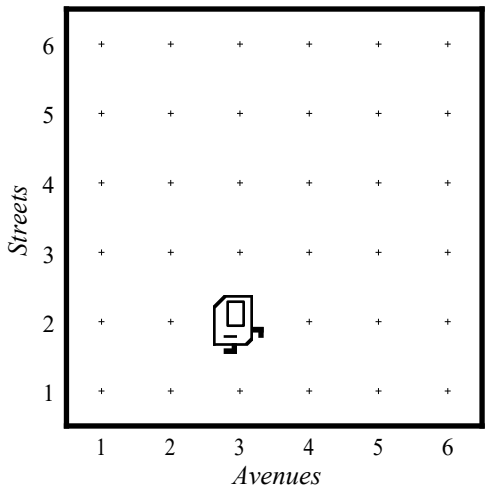
This assignment consists of several recursive functions to write at varying levels of difficulty. Learning to solve problems recursively can be challenging, especially at first. We think it's best to practice in isolation before adding the complexity of integrating recursion into a larger program. The 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. You can 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).

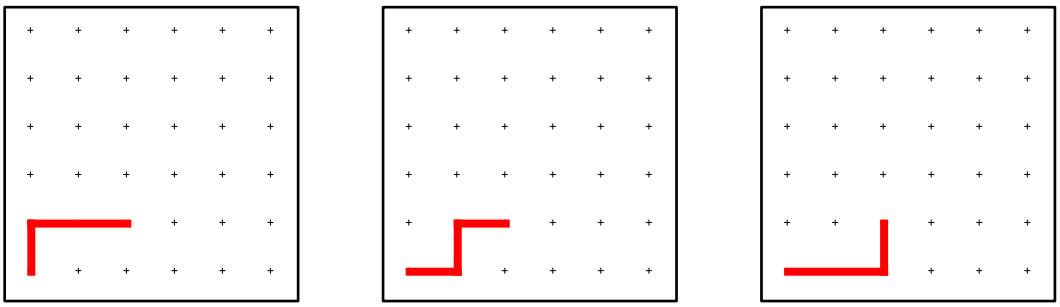
The first few problems after the warm-up exercises include some hints about how to get started, the later ones you will need to work out the recursive decomposition for yourself. It will take some time and practice to wrap your head around this new way of solving problems, but once you "grok" it, you'll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

Warm-up problem 0a 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).

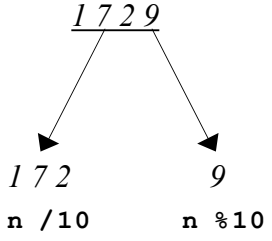
Warm-up problem 0b. Numeric Conversion

The **strlib.h** interface exports the following methods for converting between integers and strings:

```
string integerToString(int n);
int stringToInteger(string str);
```

The first function converts an integer into its representation as a string of decimal digits, so that, for example, **integerToString(1729)** should return the string **"1729"**. The second converts in the opposite direction so that calling **stringToInteger("-42")** should return the integer –42.

Your job in this problem is to write the functions **intToString** and **stringToInt** (the names have been shortened to avoid having your implementation conflict with the library version) that do the same thing as their **strlib.h** counterparts but use a recursive implementation. Fortunately, these functions have a natural recursive structure because it is easy to break an integer down into two components using division by 10. This decomposition is discussed on page 42 in the discussion of the **digitSum** function. The integer 1729, for example, breaks down into two pieces, as follows:



If you use recursion to convert the first part to a `string` and then append the character value corresponding to the final digit, you will get the `string` representing the integer as a whole.

As you work through this problem, you should keep the following points in mind:

- Your solution should operate recursively and should use no iterative constructs such as `for` or `while`. It is also inappropriate to call the provided `integerToString` function or any other library function that does numeric conversion.
- The value that you get when you compute `n % 10` is an integer, and not a character. To convert this integer to its character equivalent you have to add the ASCII code for the character `'0'` and then cast that value to a `char`. If you then need to convert that character to a one-character string, you can concatenate it with `string()`. (The Java trick of concatenating with `""` doesn't work because string constants are C-style strings.)
- You should think carefully about what the simple cases need to be. In particular, you should make sure that calling `intToString(0)` returns `"0"` and not the empty string. This fact may require you to add special code to handle this case.

Your implementation should allow `n` to be negative, as illustrated by the earlier example in which `stringToInt("-42")` returns –42. Again, implementing these functions for negative numbers will probably require adding special-case code.

Problem 1 List All Fifteens

In the card game called **Cribbage**, part of the game consists of adding up the score from a set of five playing cards. One of the components of the score is the number of distinct card combinations whose values add up to 15, with ACEs counting as 1 and face cards Jacks, Queens, and Kings counting as 11, 12 and13, respectively. Consider, for example, the following hand of cards: AD, 5C, 10S, 4H, 9C. Here, card suits are abbreviated as: D - Diamonds, S - Spade, C - Club, H – Heart.

There are three different combinations that sum to 15, as follows:

AD + 10S + 4H AD + 5C + 9C 5C + 10S

As a second example, the cards

contain the following eight different combinations that add up to 15:

5C + 5D + 5H	5C + 5D + 5S	5C + 5H + 5S	5D + 5H + 5S
5C + 10S	5D + 10S	5H + 10S	5S + 10S

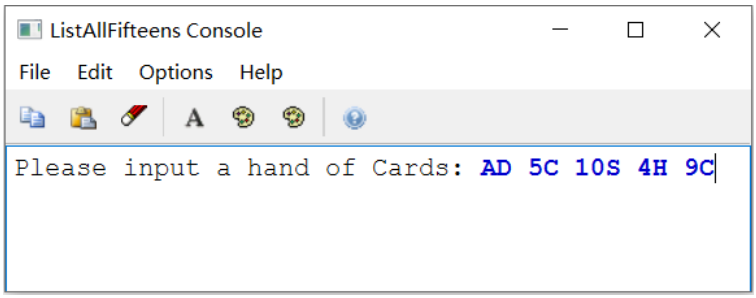
Write a function

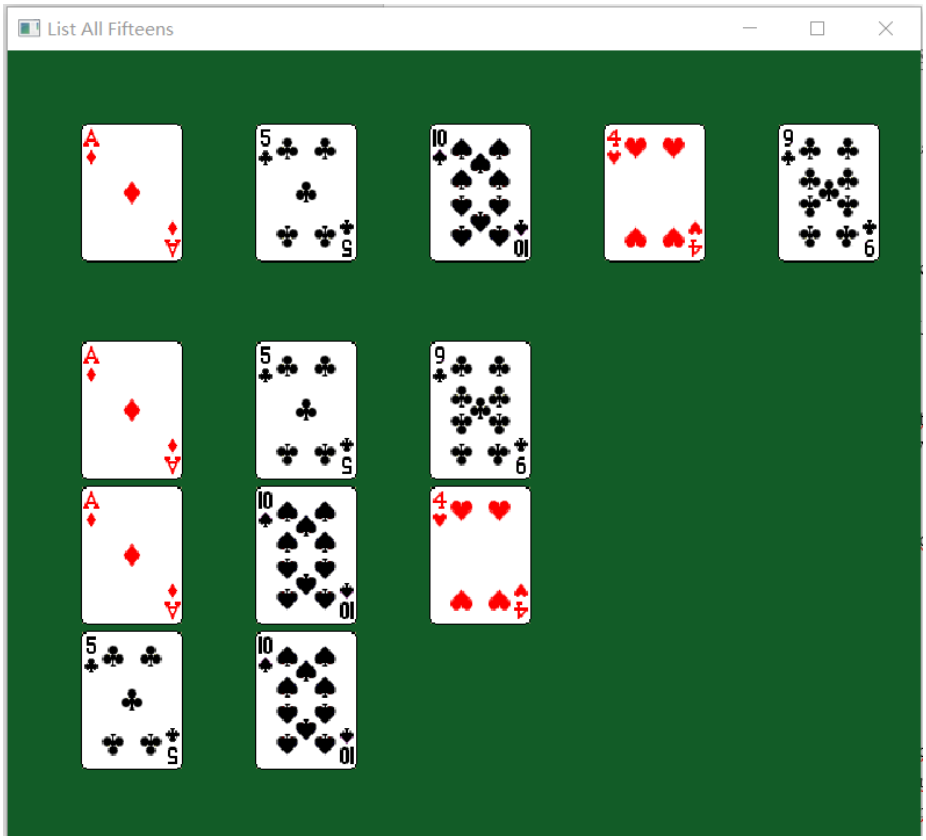
```
int listAllFifteens(Vector<Card> & cards);
```

that takes a vector of **`Card`** objects and returns the number of ways you can make 15 from that set of cards. You don't need to know much about the **`Card`** class to solve this problem. The only thing you need is the **`getRank`** method, which returns the rank of the card as an integer. In addition to the **`Card`** class, the **`card.h`** and **`card.cpp`** files included with the starter code export the constant names **`ACE`**, **`JACK`**, **`QUEEN`**, and **`KING`** with the values 1, 11, 12, and 13, respectively.

Extension

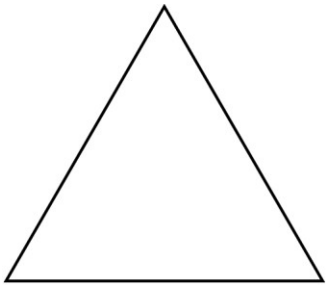
If you have solved the above problem, you can consider improving the code with the goal to output ***all cards combinations*** that add up to 15. Please apply **`Vector<Card>`** to record the information of a cards combination, and declare **`Vector<Vector<Domino>>`** to store all card combinations. What you need to do for this assignment is to implement the recursive ***listAllFifteensRec*** function in ***ListAllFifteensExtension.cpp***. You can work with a new starter folder (named as ***1b-ListAllFifteensExtension***) which contains all required program frameworks. Sample run as follows:



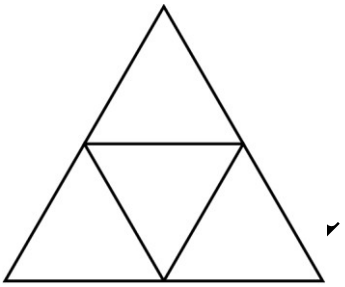


Problem 2 Sierpinski Triangle

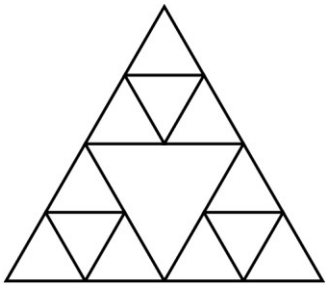
If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the ***Sierpinski Triangle,*** named after its inventor, the Polish mathematician Wacław Sierpiński (1882–1969). The order-0 Sierpinski Triangle is an equilateral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



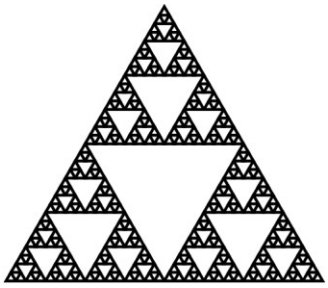
To create an order-*N* Sierpinski Triangle, you draw three Sierpinski Triangles of order $N-1$, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



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:



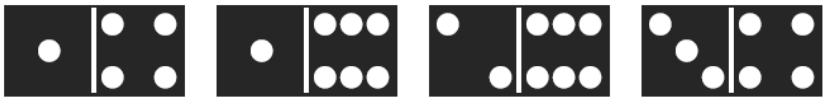
If you continue this process through three more recursive levels, you get the order-5 Sierpinski Triangle, which looks like this:



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 3 Domino Chain

The game of dominos is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following four rectangles represents a domino:



Dominos can be connected end-to-end to form chains, subject to the condition that two dominos can be linked together only if the numbers match. For example, you can form a chain consisting of these four dominos by connecting them in the following order:

As in the traditional game, dominos can be rotated by 180˚ so that their numbers are reversed. In this chain, for example, the 1–6 and 3–4 dominos have been "turned over" so that they fit into the chain.

Suppose that you have access to a `Domino` class (see ***domino.cpp*** and ***domino.h***) that exports the methods `getLeftDots` and `getRightDots`. Given this class, write a wrapper function

```
bool canFormDominoChain(Vector<Domino> & dominos);
```

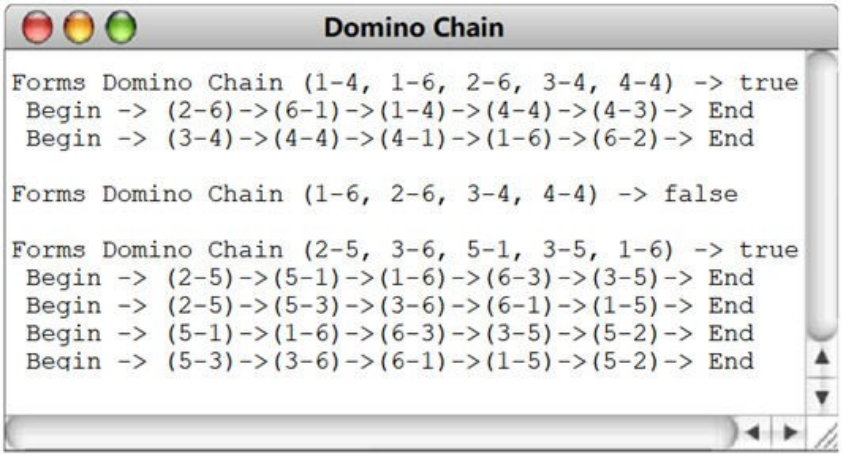
that returns true if it possible to build a chain consisting of every domino in the vector as illustrated by the following sample run:

As usually, the wrapper function supplies additional arguments to the recursive function

that checks whether a domino chain is formed. Please complete the codes in the wrapper function, as well as design and implement the recursive function carefully.

Extension

On the basis of the previous exercise, you can improve the codes to output ***all possible domino chains***. A good suggestion is to apply `Vector<Domino>` to record the information of domino chain, and declare a set of domino chains as a `Set<Vector<Domino>>` to collect all domino chains recursively. Moreover, rewrite the base cases and the recursive steps. Remember to try all options in the recursive steps. You can work with a new starter folder (named as ***3b-DominoChainExtension***) which contains all required program frameworks. Sample run as follows:



Here are a few hints**:**

- The exported method `turnOver` of `Domino` Class provides a reversed-numbers domino (see ***domino.cpp*** and ***domino.h***).

  For example:

```
Domino myDomino(1,6);
Domino yourDomino = myDomino.turnOver();
cout << myDomino << "<--->" << yourDomino << endl;
```

  Output:

```
(1,6) <---> (6,1)
```

Wrapper functions are extremely common in recursive programming. The function `canFormDominoChain(Vector<Domino>& dominos)` can check to see if the domino vector forms a chain which begins with any dominos in the vector. The recursive insight is that the entire set forms a chain if and only if there is some domino whose left side matches the designated starting number and the remaining dominos form a chain starting with the count from that domino's right side or, conversely, there is a domino that fits if you reverse its left and right sides.

Problem 4 Stock Cutting

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 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. 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 several 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 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`.