Assignment #3: Recursion

Due: Wed, Feb 6th 2:15pm

This week's task consists of a problem set of several small recursion functions to write. 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 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 first few problems have 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.

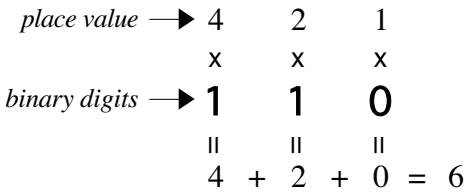
Warm-ups. First, we present two warm-up problems (one simple, one more involved), for which we provide hints and solutions. You don't need to hand in solutions to the warm-ups. We recommend you first try to work through them by yourself. If you get stuck, ask for help and/or take a look at our solutions 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).

Warm-up A: Print in binary

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. The table below shows the first few integers represented in binary:

0	→	0
1	→	1
1 0	→	2
1 1	→	3
1 0 0	→	4
1 0 1	→	5
1 1 0	→	6

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 this logic:



Binary is a base-2 number system instead of the decimal (base-10) system we are familiar with. Write a recursive function `PrintInBinary(int num)` that prints the binary representation for a given integer. For example, calling `PrintInBinary(5)` would print 101. Your function may assume the integer parameter is non-negative.

The recursive insight for this problem is to realize you can identify the least significant binary digit by using the modulus operator with value 2. For example, given the integer 35, mod by 2 tells you that the last binary digit must be 1 (i.e. this number is odd), and division by 2 gives you the remaining portion of the integer (17). What 's the right way to handle the remaining portion? What is the simplest possible number(s) to print in binary that can be used as the base case?

One twist to this problem is that you need to work backwards. There is a straightforward way to easily identify and print the last binary digit, but you need to print that digit only *after* you have printed all the other binary digits. This dictates the placement of the printing relative to the recursion.

```
void PrintInBinary(int number)
```

Warm-up B: Subset Sum
The subset sum problem is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to that target number. For example, given the numbers {3, 7, 1, 8, -3} and the target sum 4, the subset {3, 1} sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this function is

```
bool CanMakeSum(Vector<int> & nums, int targetSum)
```

Remember that many recursive problems are variations on the same age-old themes. Consider how this problem is related to the ListSubsets function from lecture. Take a look at that code first. You should be able to fairly easily adapt it to operate on a vector of numbers instead of a string. Note that you are not asked to print the numbers in the sum, just return a Boolean result. You will likely need a wrapper function to pass additional state through the recursive calls— what is the other information you need to track as you try various combinations?

Once you have a basic version of the function working, here are some other variations to give you even more practice.

- The recursive decomposition from **`ListSubsets`** considers the next element and tries it both in and out of the current subset being assembled. This is a classic "in/out" exhaustive recursion pattern. Consider this alternative recursive decomposition: at each step, choose one of the remaining elements to add to the subset and recur from there. This is another classic pattern of "choose one from those remaining". Rewrite the function to use this alternative strategy. One special issue with this version is that you don't want to wastefully try the same subset more than once, so be careful to be sure each possible subset is examined at most once (i.e. after trying ABC there is no reason to try CAB and BCA).

- How could you change the function to print the subset members that sum to the target when successful? One method is to store the numbers into another vector that is updating during the recursive calls. Another approach doesn't add any new data structures or store the numbers, it just prints the chosen members when unwinding from the recursive calls.

- How could you change the function to report not just whether any such subset exists, but the count of all such possible subsets? For example, in the set shown earlier, the subset {7, -3} also sums to 4, so there are two possible subsets for target 4.

Now onto the assigned problems!

A note on testing: For each problem on this assignment, you will want to thoroughly test your function to verify it correctly handles all the necessary cases. For example, for the binary warm-up, test code might call your function on the first 50 numbers or allow the user to enter numbers to test until satisfied. Testing code like this 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 your submission, no need to remove it.

For each exercise, we specify the function prototype. **Your function must exactly match our given prototype** (same name, same arguments, same return type). This function can be just a wrapper that does some setup and makes a call to the real recursive function.

Also note your function must operate recursively; even if you can come up with an iterative alternative, we insist on a recursive formulation!

1. The 12-step program to recursive enlightenment

You're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: via three small strides or one small stride followed by one large stride or one large followed by one small. A staircase of four steps can be climbed in five different ways (enumerating them is an exercise left to reader :-).

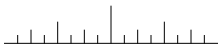
Write the recursive function `int CountWays(int numStairs)` that takes a positive `numStairs` value and returns the number of different ways to climb a staircase of that height taking strides of one or two stairs at a time.

Here's a hint about the recursive structure of the problem: consider the options you have at each stair. You must either take a small stride or a large stride; either will take you closer to the goal and therefore represents a simpler instance of the same problem that can be handled recursively. What is the simplest possible situation and how is it handled?

```
int CountWays(int numStairs)
```

2. Ruler of the world

In countries like the United States that still use the traditional English system of measurement, each inch on a ruler is marked off into fractions using tick marks that look like this:



The tallest tick mark falls at the center dividing into halves, two shorter tick marks indicate the quarter divisions, and even shorter ones are used to mark the eighths and sixteenths and so on. Write the recursive function `DrawRuler(double x, double y, double w, double h)`. The function is given the coordinates for a rectangle in which to draw the ruler. The function draws a line along the rectangle's bottom edge and a sequence of vertical tick marks. The middlemost tick mark is centered and is as tall as the rectangle height. Each smaller tick mark is half the height of the next larger one. Once the tick marks become sufficiently small, the recursion terminates.

The recursive insight to this problem is to recognize that the middlemost tick mark subdivides the rectangle into two smaller rectangles, each of which is a smaller ruler of its own.

```
void DrawRuler(double x, double y, double w, double h)
```

3. Every vote counts

It's said that "every vote counts", but does every vote count equally? There are 538 votes in the US Electoral College and a presidential candidate must earn a majority (270 votes) to be elected. Each state is allocated a number of electoral votes based on population; California has 55, Wyoming just 3. The electoral votes for each state are cast in a block, all for the same candidate. Given that California's block is roughly 10% of all electoral votes, you could conclude it has 10% of the "voting power" and Wyoming a mere half-percent, but the dynamics of a block-voting system are a bit more subtle.

One measure of a vote's importance is the count of situations in which that vote is critical. A *critical* vote is one that changes the election outcome. Consider an election between Alice and Bob. Excluding California, Alice has 250 votes and Bob 233. If California votes for Alice, adding those 55 votes give the win to Alice. If it goes with Bob, Bob wins. Thus California's vote is critical in this situation. If Alice has 300 votes and Bob 183, California's vote is not critical, because Alice wins regardless of how California votes.

For this problem, you will count the number of situations in which a block's vote is critical. The voting system has N blocks. Take the target block and set it aside. Now consider the possible election outcomes using the remaining N-1 blocks. There are 2^{N-1} different possibilities; each corresponds to assembling a subset voting for Alice, the other votes going to Bob. In each of those 2^{N-1} situations, consider adding in the target block's vote— will its choice of candidate force the win for that candidate? If so, then it's a critical vote.

Solving this problem builds on the solution to the subset sum warm-up. You build a coalition and total the votes, i.e. choose a subset and sum it, just as the warm-up does. Carefully consider what conditions must be true about the coalition sum in order for the target block's vote to be both necessary and sufficient to reach a majority. The code from the warm-up is a great start, but you'll need to modify some of the details to add the housekeeping and counting required for this problem. (If you skipped the warm-up, now might be a good time to reconsider that decision…). Adapting an existing recursion solution to solve a similar problem is a great way to build your recursive mastery.

The recursive function `CountCriticalVotes(Vector<int> & blocks, int blockIndex)` is given a vector of block vote counts and an index within that vector. The function counts the number of election outcomes in which the block at the given index has a critical vote. The vector `{4,2,7,4}` would represent a voting system with four blocks of 17 total votes. A majority of 9 votes is required to win the election. If the block index is 3, the function counts the critical votes for the last block. If the first two blocks vote for Alice, and the third for Bob, the last block's vote is critical. If the first block votes for Alice, and the second two for Bob, then the last block's vote is irrelevant. The last block's vote is critical in just two of the eight possible situations. Surprisingly, the smaller 2-vote block at index 1 has the same count of critical votes, giving it equivalent voting power despite its smaller size. The large 7-vote block has six critical votes, three times as many as the other blocks, notwithstanding its modest size relative to the others. Very interesting!

The number of critical votes per block is used in computing the Banzhaf Power Index, a metric designed by John Banzhaf III in a lawsuit he raised challenging the fairness of a block-voting system. His particular objection was to a system with block vote counts of `{9,9,7,3,1,1}` where the three smallest blocks never cast a critical vote!

```
int CountCriticalVotes(Vector<int> & blocks, int blockIndex)
```

4. Cell phone mind reading

Entering text using a phone keypad is problematic; there are only 10 digits for 26 letters and thus each digit key is mapped to several letters. Some cell phones require "multi-tap" —tap the 2 key once for 'a', twice for 'b' and three times for 'c', which can get tedious. A streamlined alternative such as Tegic's T9 predictive text requires just one tap on each digit, it then guesses which letter was intended based on the sequence so far and its possible completions.

For example, if the user types the digit sequence "72", there are nine possibilities (pa, pb, pc, ra, rb, rc, sa, sb, sc)**.** Three of these seem promising (pa, ra, sa) because they prefix words such as "party" and "sandwich", while the others can be ignored since they don't start any words. If the user enters "9956", there are 81 (3^4) possibilities, but you can be assured the user meant "xylo" since that is the only one that is a prefix of any English words.

Write the function **`ListCompletions(string digits, Lexicon & lex)`** that prints all words from the lexicon that can be formed by extending the given digit sequence. For example, the completions for "72547" are:

palisade palisaded palisades palisading palish rakis rakish rakishly rakishness sakis

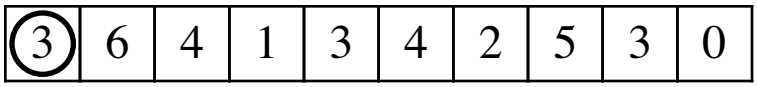
Some hints to get you started:

- Start by working through **`ListMnemonics`** from Section #3 exercises. This is very useful code to study and adapt for use in this problem.
- Note there are two recursive pieces. They require similar, but not identical, code. First you must explore converting the digit sequence into letters. Then, you need to recursively extend that prefix in attempt to build words. 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?
- For this problem, we supply **`Lexicon`**, a special-purpose class for storing a word list, and datafile containing a large English dictionary. Read the interface file **`lexicon.h`** in the starter project for details on how to create a new lexicon, check if a word exists, and so forth. Behind the scenes, the lexicon employs a scarily complex representation that is extremely efficient and compact to give you blindingly fast access to a list of over 100,000 words. However you don't need to know any of that in order to use it!
- Be sure to take advantage of the Lexicon member function **`containsPrefix`**. This allows you check whether a sequence of letters is the prefix of any word contained in the lexicon. Use this to avoid going down dead ends.
- The letters Q and Z have a long history of being ignored on phone keypads, please rectify this injustice by including Q with the 7 group and Z with 9.

```
void ListCompletions(string digitSequence, Lexicon & lex)
```

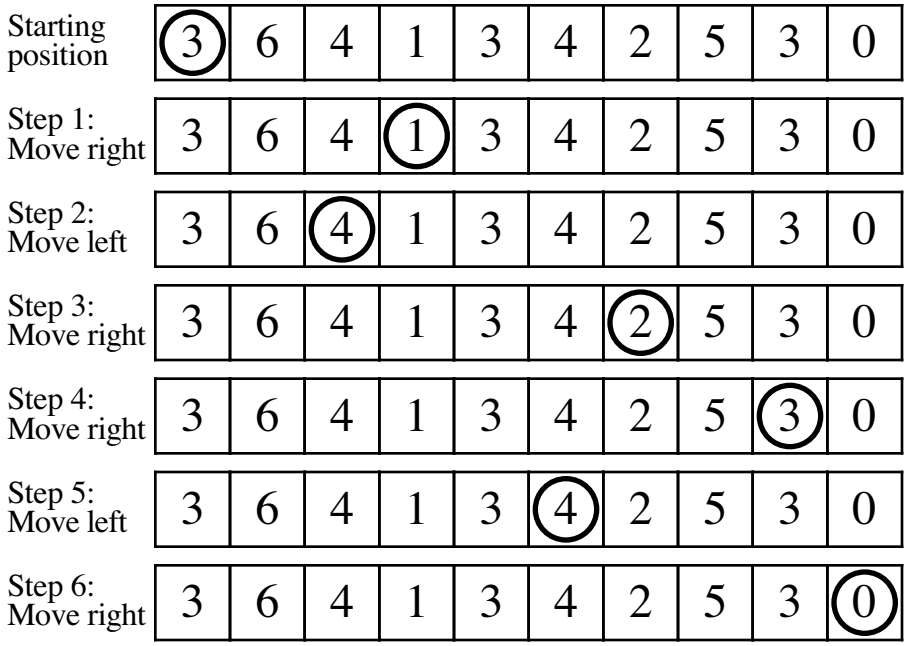
5. A recursive puzzle

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:



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:

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!)

```
bool Solvable(int start, Vector<int> & squares)
```

6. Stock cutting

You are charged with buying the plumbing pipes for a construction project. Your foreman gives you a list of the varying lengths of pipe needed. Home Depot sells stock pipe in one fixed length. You can divide 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.

The recursive function `CutStock(Vector<int> & requests, int stockLength)` is given a vector of the lengths needed and the stock pipe length. It 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: `{4,4,1} {3,7} {8}` and have two small leftover remnants. There are other possible arrangements that also fit into three stock pipes, but it cannot be done with fewer.

Cutting stock is a representative of the fundamental problem for 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 optimally cut pieces of material from standard-sized sheets. Shipping companies use a three-dimensional version to optimally pack containers or trucks.

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

A few hints and specifications:

- You can assume that all elements in the vector are positive numbers that are no longer than the stock pipe length. In the worst case, it will take a number of stock pipes equal to the size of the vector.
- You do not need to report the cutting/division in the optimal solution, just report the number of stock pipes needed.
- You will definitely need a wrapper function as you must maintain additional state as you work through the recursive calls. Consider what information you will need to track as you try out various configurations.
- There are several different approaches for recursively decomposing this problem and how you manage 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 use the solution to the smaller subproblem(s) to solve the larger problem?

```
int CutStock(Vector<int> & requests, int stockLength)
```

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 subproblem. 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 and you may want to take it easy and save yourself up for Boggle next week. 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 off the top of my head, I'm sure you will think of others!

- 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.
- Take your previous maze program and implement a solver that works via recursive backtracking.
- See exercises 12 and 13 in Chapter 5 for ideas on further recursive drawing. There are many cool varieties 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.

Accessing files http://see.stanford.edu/see/materials/icspacs106b/assignments.aspx

On the class web site, you'll find a starter folder containing these files:

lexicon.cpp/h	Interface/implementation for Lexicon (**add** .cpp to project)
lexicon.dat	Large lexicon word file in binary format

To get started, create your own starter project and **add the .cpp file for lexicon** to the list of files compiled for your project.

Deliverables

We expect 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. Hand in a printed version in lecture, as well as an electronic version via ftp. Both are due at the beginning of lecture on the due date. SCPD students need only e-submit. 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 know how to fail at the worst time!

"If you already know what recursion is, just remember the answer. Otherwise, find someone who is standing closer to Douglas Hofstadter than you are; then ask him or her what recursion is." — Andrew Plotkin