Assignment #3: Recursion

Due: Mon, Jan 31st 2:15pm

Most of the assignments in this course are single programs of a substantial size. However, this week your task is a set of small problems to solve in isolation. Recursion is a difficult concept to master and one that is worth concentrating on separately before you try to integrate it into a larger program. For each problem, you are to write a short recursive program to solve it (even if you could take an iterative approach, we insist on a recursive formulation!). By short, we mean that none of them will require more than a page or so to complete. That doesn't mean you should put this assignment off until the last minute though—recursive solutions can often be formulated in just a few concise, elegant lines but if you don't yet comprehend recursion, they can be extremely dense and complex.

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 one simple recursive problem and a second more involved one, for which we provide hints and solutions. You don’t need to hand in your solutions to the warm-up problems. 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 the staff and 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. Note that for the assignment problems, you must hand in your own original, independent work.

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  |
| 10 → 2 |
| 11 → 3 |
| 100 → 4 |
| 101 → 5 |
| 110 → 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:

place value → 4 2 1
              x x x

binary digits → 1 1 0
               || ||
               4 + 2 + 0 = 6

Basically, this is a base-2 number system instead of the decimal (base-10) system we are familiar with. Write a recursive function \texttt{PrintInBinary(num)} that prints the binary representation for a given integer. For example, calling \texttt{PrintInBinary(3)} would print 11. Your function may assume the integer parameter is always non-negative.
The recursive insight to solve this problem is to use the fact that 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 is the simplest possible number(s) to print in binary that identifies the base case that stops the recursion?

One slightly tricky part about the recursive decomposition here is that you have to think about it a bit 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. (Think back to our discussion of reversing a file in lecture and the order of the get/put activity and the recursive call…)

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 set \{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(int setOfNums[], int nElements, int targetSum)

Remember that many recursive problems are variations on the same common 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 an array of numbers instead of a string. Note that you are not asked to print the subset members, just return true/false. 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 subset combinations?

Once you have a basic version of the function working, here are some other variations you may want to play with to get even more practice.

- How could you change the function to print the members in the successful subset if one is found? Do this without adding any new data structures (i.e. don't build a second array to hold the subset), just use the unwind of the recursive calls.

- The recursive decomposition from ListSubsets considers the next member and tries it both in and out of the current subset being assembled. This is a classic "in/out" exhaustive recursion pattern. Try instead implementing the function to use this alternative recursive decomposition: at each step, choose one of the remaining members to add to the subset and recur from there. This is another classic, more of a "choose one from remaining" pattern. This alternative strategy works equally well (although the two may find different subsets when more than one works). One special issue to consider 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 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 warmup, you might use test code to call your function in a loop on the first 50 integers or use a loop to allow the user to repeatedly enter numbers that are fed to your function until you are 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 the file you submit, no need to remove it.

1. Balancing parentheses

In the syntax of most programming languages, there are characters that occur only in nested pairs, which are called **bracketing operators**. C++, for example, has these bracketing operators:

```
( . . . )
[ . . . ]
{ . . . }
```

In a properly formed program, these characters will be properly nested and matched. To determine whether this condition holds for a particular program, you can ignore all the other characters and look simply at the pattern formed by the parentheses, brackets, and braces. In a legal configuration, all the operators match up correctly, as shown in the following example:

```
{ ( [ 1 ) ( [ ( ) ] ) } }
```

The following configurations, however, are illegal for the reasons stated:

```
( [ ] ) The line is missing a close parenthesis.
) ( The close parenthesis comes before the open parenthesis.
{ ( } ) The parentheses and braces are improperly nested.
```

For this problem, your task is to write a recursive function `bool IsBalanced(string str)` that takes a string `str` from which all characters except the bracketing operators have been removed. The function should return `true` if the bracketing operators in `str` are balanced, which means that they are correctly nested and aligned. If the string is not balanced, the function returns `false`.

Although there are many other ways to implement this operation, you should code your solution so that it embodies the recursive insight that a string consisting only of bracketing characters is balanced if and only if one of the following conditions holds:

- The string is empty.
- The string contains "()", "[]", or "{}" as a substring and is balanced if you remove that substring.

For example, the string "[({}]") is shown to be balanced by the following chain of calls:

```
IsBalanced("[({}]") →
IsBalanced("[{}]") →
IsBalanced("[") →
IsBalanced(""") → true
```

`bool IsBalanced(string str)`
2. Arithmetic combinations

Write the `ArithmeticCombinations` function which given an array of integers and a desired result, recursively determines how many different ways you can combine those integers to form that result. Combining the numbers means choosing a sequence of arithmetic operations to apply to the numbers to produce an expression that evaluates to the result. The sum starts as the first number in the array and you process the remaining array entries from left to right. For each number, you can either add it to the current sum, subtract it from the current sum or multiply the current sum by the number (i.e. valid operators are +, -, and *). All numbers in the array must be used in the expression and must appear in the same order in the expression as in the array.

For example, consider the array `{7, 0, -3, 4, 10}` and desired result 6. The function should return 3 since there are three possible combinations:

\[
\begin{align*}
7 & \times 0 \times -3 - 4 + 10 = 6 \\
7 & + 0 + -3 \times 4 - 10 = 6 \\
7 & - 0 + -3 \times 4 - 10 = 6
\end{align*}
\]

Note the expression is not evaluated using C++ precedence rules, it is simply evaluated left to right.

The first two parameters to the function are the array of numbers and its effective size. You may assume the array always has at least one entry. The third parameter is the desired result. Your function should return the count of the different ways you can combine the numbers to get that result (you do not have to keep track of the combinations, just report the number of possible combinations). The array should not be modified. You will need a wrapper function.

```c
int ArithmeticCombinations(int arr[], int n, int result)
```

3. Longest common subsequence

A `subsequence` of a string is a sequence of characters extracted from the string that preserves the original order of the characters. A subsequence can be thought of as a generalized substring; one that can skip over chars. For example, "apple" contains the subsequences "ppl" and "ale", among others. Rearrangements or duplications such as "pa" or "aal" are not subsequences of "apple". A `common` subsequence of two strings is a subsequence that appears in both. One common subsequence of "apples" and "pears" is "pes". Another is "as".

For this problem, you are to write the recursive function `LongestCommonSubsequence` that will determine the longest common subsequence between two strings. The two parameters to `LCS` are the two strings, the return value is the longest common subsequence. If there is a tie for the longest subsequence, your function can return any one of them. If there is no common subsequence at all (i.e. the strings have no characters in common), return the empty string.

Here are a few examples:

```c
LCS("cs106b", "rocks") = "cs" 
LCS("Recursion", "c*u*r*s*e!") = "curs" 
LCS("abcd", "xBy") = ""
```

There are several different approaches you could use to recursively decompose this problem. Try to think through what smaller, similar subproblems exist within the problem and how their solution could be useful in solving the entire problem.

```c
string LCS(string s1, string s2)
```
4. Making change

Everyone who's had an exciting summer job at Jack-in-the-Box (that is to say, me, at least) has had to deal with the intellectually stimulating task of making change. The customer is owned $.90, do I give them 9 dimes? 3 quarters, a dime, and a nickel? 90 pennies? What if I can't count that high?

There are clearly many different configurations of coins that will work; let's say that we are interested in the combination that uses the fewest coins. One approach would be to use as many high value coins (i.e. quarters) as possible, then move on to use dimes for what is leftover, then nickels and finally pennies if needed. This type of algorithm is known as a greedy algorithm since at any given moment it makes the choice that looks best at the moment, the hope being that the locally optimal choice will lead to a globally optimal solution. However, what if I had no nickels in my change drawer and was trying to make $.31? The greedy solution chooses 1 quarter and 6 pennies, which is worse than the optimal solution of 3 dimes and 1 penny. Clearly we need something cleverer to find the truly optimal arrangement, thus, recursion to the rescue!

Write a function `int MakeChange(int amount, int coinValues[], int n)` that takes an amount along with the array of available coin values and its effective size. You can assume you have use many of each coin as you want (i.e. if your cash drawer has pennies, it has an infinite supply of them). The function should return the minimum number of coins required that sum to the given amount. If the amount cannot be made (for example if you try to make $0.31 and have no pennies), the function should return -1. You do not have to print or return the coin combination, just return the minimum number of coins in the optimal configuration.

Like the previous problem, there are several different ways to recursively decompose this problem. Let your creativity lead you toward the one that makes most sense to you.

```c
/* Sample usage:
 * int coins[] = {10, 25,  1};    dimes, quarters, & pennies
 * min = MakeChange(31, coins, 3);  returns 4
 * min = MakeChange(31, coins, 2);  returns -1
 */
int MakeChange(int amount, int coinValues[], int nValues)
```

5. A recursive puzzle

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

```
3 6 4 1 3 4 2 5 3 0
```

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:

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

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, int squares[], int nSquares)` that takes a starting position of the marker along with the array of squares and its effective size. 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 array are non-negative. The values of the elements in the array 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 code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same 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.

The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you'll be much better prepared for the next assignment where you will have to implement some fairly sophisticated recursive algorithms.

Getting started
There are no starter projects to download, since there are no special modules or starting code we need to give you.

Deliverables
The deliverables for this assignment is one recursion.cpp source file that contains all assigned functions with exactly the prototypes given above. Your source file can also contain testing code you used when developing your solutions.

You are required to submit both a printed version in lecture, as well as an electronic version via ftp. Both are due before the beginning of lecture on the due date. The printed version should be handed in before lecture starts. Be sure the pages are firmly stapled together and that the printout is clearly marked with your name, CS106B and your section leader's name! SITN students need to submit only electronically, no need for paper copies through courier.

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!

Little-known fact: If you have 3 quarters, 4 dimes, and 4 pennies, you have $1.19, which is the largest amount of money in coins without being able to make change for a dollar.