Assignment 1: Welcome Back to C++

Isn't it a happy thing if you review what you have learned on time?

— The Analects of Confucius, *Confucius*, 500 BC

Most of the assignments in this course are single programs of a substantial size. To get you started, however, this assignment is a series of four short problems that are designed to give you a review and practice on how to apply C++ function and to reinforce the concept of functional recursion. None of these problems require more than a page of code to complete; most can be solved in just a few lines.

Problem 1 Finding Perfect Numbers (Chapter 2, exercise 5, page 123)

Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of *n* is any divisor less than *n* itself). They called such numbers ***perfect numbers***. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

Write a predicate function ***isPerfect*** that takes an integer ***n*** and returns ***true*** if ***n*** is perfect, and ***false*** otherwise. Test your implementation by writing a main program that uses the ***isPerfect*** function to check for perfect numbers in the range 1 to 9999 by testing each number in turn. When a perfect number is found, your program should display it on the screen. The first two lines of output should be 6 and 28. Your program should find two other perfect numbers in the range as well.

Problem 2 Adding Commas (Chapter 3, exercise 14, page 153)

When large numbers are written out on paper, it is traditional—at least in the United States—to use commas to separate the digits into groups of three. For example, the number one million is usually written in the following form:

1,000,000

To make it easier for programmers to display numbers in this fashion, implement a function

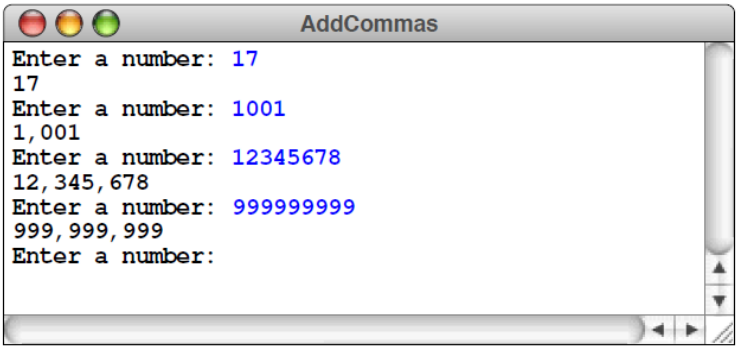
***string addCommas*(*string digits*);**

that takes a string of decimal digits representing a number and returns the string formed by inserting commas at every third position, starting on the right. For example, if you were to execute the main program

```
int main() {
   while (true) {
      string digits = getLine("Enter a number: ");
```

```
            if (digits == "") break;
            cout << addCommas(digits) << endl;
        }
        return 0;
    }
```

your implementation of the addCommas function should be able to produce the following sample run:



Problem 3 Finding DNA Strand Matches (Chapter 3, exercise 20, page 157)

There is no gene for the human spirit.

—Tagline for the 1997 film *GATTACA*

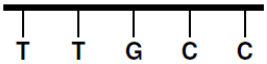
The genetic code for all living organisms is carried in its DNA—a molecule with the remarkable capacity to replicate its own structure. The DNA molecule itself consists of a long strand of chemical bases wound together with a similar strand in a double helix. DNA's ability to replicate comes from the fact that its four constituent bases—adenosine, cytosine, guanine, and thymine—combine with each other only in the following ways:

• Cytosine on one strand links only with guanine on the other, and vice versa.

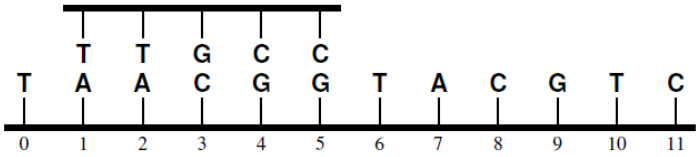
• Adenosine links only with thymine, and vice versa.

Biologists abbreviate the names of the bases by writing only the initial letter: **A**, **C**, **G**, or **T**. Inside the cell, a DNA strand acts as a template to which other DNA strands can attach themselves. As an example, suppose that you have the following DNA strand, in which the position of each base has been numbered as it would be in a C++ string:

Your mission in this exercise is to determine where a shorter DNA strand can attach itself to the longer one. If, for example, you were trying to find a match for the strand



the rules for DNA dictate that this strand can bind to the longer one only at position 1:



By contrast, the strand

matches at either position 2 or position 7.

Write a function

***int findDNAMatch*(*string s*1, *string s*2, *int start* = 0);**

that returns the first position at which the DNA strand ***s*1** can attach to the strand ***s*2**. As in the ***find*** method for the ***string*** class, the optional ***start*** parameter indicates the index position at which the search should start. If there is no match, ***findDNAMatch*** should return –1.

Problem 4 Conversion Between Integers And Strings

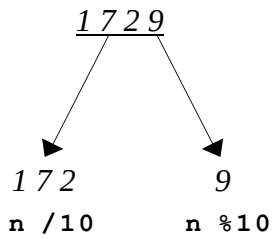
The ***strlib.h*** interface exports the following methods for converting between integers and strings:

***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.