CSCI 338—Algorithm Design and Analysis
Programming Assignment 1: Decimal Addition and Multiplication

Due: Friday, February 3, 2017

This is an individual assignment.

This assignment is to implement the "grade school" algorithms for addition and multiplication of decimal integers, represented as strings. You will implement each algorithm as a function or method and write a simple program to demonstrate the two algorithms. The full specifications are below. Please use exactly the specifications given.

Language

You may use Java, C, C++, or Haskell for this assignment. If you wish to use a different language, you must first get permission from the instructor. Here a few specifics about the different languages:

1. *Java*: Implement the two algorithms as *static* methods in the same class. Use the String class to represent the numbers. The demonstration program should be in a separate class by itself.
2. *C*: Implement the two algorithms as non-static functions with char* (char arrays) for the parameters and return value. Allocate the return array dynamically with malloc() or one of its cousins. The two functions should be in the same .c file, ideally with a separate .h file, the .h file is not required. The demonstration program should be in a separate file, which either includes the .h file (if you have it) or the other .c file. With the .h file, you'll do separate compilation of the two .c file and then link them. The separate compilation is the better way to do this. The demonstration program should free() any dynamically allocated memory before it exits.
3. *C++*: Implement the two algorithms as non-static functions with String class objects for the parameters and return values. You must use a .h file to specify the interface for the functions and separate the demonstration program.
4. *Haskell*: Implement the two algorithms as curried Haskell functions in a module. The functions should use Haskell strings, which are represented as lists of characters. You don't have to do the separate demonstration program if you use Haskell, since it's easy to test functions directly.

The Addition Algorithm

The algorithm specification are as follows:

Input:	Two *n*-digit strings representing non-negative integers. If the two input values are different lengths, the shorter one is padded with leading zeros to make them the same length, before passing it to the algorithm.
Output:	An (*n*+1)-digit string representing the sum of the two input integers. Pad the output with a leading zero if the sum is only *n* digits.
Method:	Implement the grade-school algorithm of right to left addition. Note that the rightmost (least significant) digit is in position *n*-1 and the leftmost (most significant) digit in position 0. You will have a carry in of zero for the rightmost position. You may not use any built-in binary to decimal conversions, but you may convert individual characters between character representation and the corresponding integer

value. You can do this easily by adding or subtracting the code value of '0' to or from the character or digit value. You need to convert the sum of three digits into two components: the sum digit and the carry out digit, although you may leave the components in integer form.

Errors: Throw an exception (Java or C++), return a NULL pointer (C), or zero-length string (Haskell) if there is an error in the parameters: zero-length input, input of different length, or characters other than decimal digits in the input. If the input meets these standards, the algorithm should complete correctly.

The Multiplication Algorithm

The algorithm specification are as follows:

Input: Two *n*-digit strings representing non-negative integers. If the two input values are different lengths, the shorter one is padded with leading zeros to make them the same length, before passing it to the algorithm.

Output: A 2*n*-digit string representing the product of the two input integers. Pad the output with a leading zero if the product is less than 2*n* digits.

Method: Implement the grade-school algorithm of sum of partial products. Note that the rightmost (least significant) digit is in position *n*-1 and the leftmost (most significant) digit in position 0. You will have a carry in of zero for the rightmost position.

To form the partial products, you need an auxiliary (helper) function to multiply an n-digit integer by a single digit, producing an (*n*+1)-digit output; this function should work digit-by-digit just as the addition function does. You may also want an auxiliary function to shift digits to the left. Note that the partial products should be 2*n*-1 digits after shifting.

You can use the addition algorithm to add the partial products to the sum.

Errors: Throw an exception (Java or C++), return a NULL pointer (C), or zero-length string (Haskell) if there is an error in the parameters: zero-length input, input of different length, or characters other than decimal digits in the input. If the input meets these standards, the algorithm should complete correctly.

The Demonstration Program

This program should be very simple. It should read two input strings from standard input; the strings should be entered on separate lines. It should call both of the two functions you developed above, printing the sum and product to standard output on separate lines. Do not add additional output. However, the algorithms return the error code, it should print the string "ERROR" for that output. (You may need to check for NULL return.)