CMPUT201—Assignment 2 (Winter 2025)

R. Hackman

Due Date: Friday February 28th, 8:00PM

Per course policy you are allowed to engage in collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question. However, for purposes of your learning it is highly recommended that you be able to complete each of these questions on your own.

In each assignment you should test against the sample executables. The sample executables may try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case. **Important:** provided sample test cases are just that - *samples*. They are **NOT** sufficient to test your program! You must be writing additional test cases if you want to ensure your program works correctly.

Compilation Flags: each of your programs should be compiled with the following command:

```
gcc -Wall -Wvla -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

Allowed libraries for this assignment: `stdio.h`. No other libraries are allowed.

1. **Rotating Numbers**

   In this question you will be solving the task of "rotating" a number to the right. The process of rotating a number to the right once is simply taking its least significant digit and moving it to the most significant digit. For example if the number `5347` was rotated once to the right it would be the number `7534`.

   Write the program `rotate.c` which reads one integer from the standard input stream and prints out, one per line, *every* rotation of that number which was read in. For example, if the input to your program was the integer `5347` then your program would print:

   ```
   5347
   7543
   3754
   4375
   ```

   You may assume that every rotation of the integer read in will be within the bounds of an integer.

   Deliverables For this question include in your final submission zip your c source code file named `rotate.c`

2. **Credit Card Verification**

If you've ever attempted to purchas an item online with a credit card, you may have accidentally entered your credit card information incorrectly. Upon entering your credit card information incorrectly you may have immediately been informed that the provided credit card was not valid. Have you ever wondered how, out of all the possible credit card numbers, the online store immediately knew the number you entered was not a valid credit card? This is because credit card numbers are verified with an algorithm known as *Luhn's Algorithm*. This is a very simple *checksum* algorithm that once could choose to use to specify what set of values are valid versus those that are not. Note, Luhn's algorithm does not provide any security it is simply a way to quickly validate if a number is possibly valid, and the major credit card companies all use it to do so (so all valid credit card numbers from these companies meet the checksum determined by Luhn's algorithm).

Given an "account number" Luhn's algorithm verifies the number is potentially valid with a simple checksum. given an account number of n digits of the form $d_1d_2d_3d_4...d_n$ then the following procedure is followed.

(a) The nth digit is the check digit, and is only used at the end to verify validity.

(b) Moving left from the nth digit each digit has its value added to a sum. However, we follow a different procedure for every second digit.

(c) The first digit to the left of our check digit has its value doubled, if that product is larger than or equal to 10 then 9 is subtracted from the product to give us the value to add to our sum.

(d) The next digit to the left just has its value added to our sum.

(e) Repeat that process moving left until there are no more digits: doubling one subtracting 9 if necessary, then not doubling the next.

(f) Once the sum is calculated it should be multiplied by 9, the result of this product should have a remainder when divided by 10 that is equal to the check digit.

For example, if we want to check if an account number `34682` is valid, we would do the following procedure:

- Our last digit is our check digit, it is 2.
- The next digit to the left is 8, so since this is the first digit to the left we multiply it by 2 which leaves us with 16. Since 16 is larger than or equal to 10 we subtract 9 from it leaving us with 7 to add to our sum.
- The next digit to the left is 6, since it follows a digit we just doubled we do not double it and simply add 6 to our sum.
- The next digit to the left is 4, since we didn't double the last digit we now double this one leaving us with 8 to add to our sum.
- The next digit to the left is 3, since it follows a digit we just doubled we do not double it and simply add it to our sum.
- So our sum is now 7+6+8+3=24. 24*9=216, the remainder when dividing 216 by 10 is 6, so since our check digit is not 6 this account number is *invalid*.

For this question you must complete the program `luhns.c` which reads digits from the standard input stream until the first non-digit character is received (indicating the end of the account number). Your program should print `Valid` if the account number read in passes Luhn's algorithm and should print `Invalid` if the account number does not pass Luhn's algorithm.

Constraints: For this question you are not allowed to use arrays. Use of arrays or pointers of any kind will result in 0 on this question. You do not need an array to solve this question, and part of the challenge of this question is thinking of how to tackle the task without using an array.

Deliverables For this question include in your final submission zip your c source code file named `luhns.c`

3. Roman Numerals

Roman numerals refers to a numeral system originating in ancient rome - this numeral system is often used today for various purposes (such as superbowl, the upcoming superbowl being LIX). We are going to write a program that can translate numbers written in a simplified version of the Roman numeral system into their representation in our commonly used Arabic numeral system. The simplified Roman numeral system we will translate will have the following properties and rules.

- Our symbols for representing numbers will be I, V, X, L, C, D, and M representing 1, 5, 10, 50, 100, 500, and 1000 respectively.
- Our numbers are represented by a sequence of these characters, however there are rules about how these form numbers.
- The first rule, is that if a numeral is placed after a larger (or equal) numeral, then its value should be added to our total.
- The second opposing rules, is that if a numeral is placed after a smaller numeral, then the value of the smaller numeral should be deducted from our total.

For example, the number `IV` represents 4, because the `I` immediately precedes the larger numeral `V`, so the 1 that the `I` represents is subtracted from our total while the `V` which represents 5 is added to our total.

This can be tricky though, consider the number `XLI` which represents 41, because the `X` immediately precedes a larger numeral `L` so it is negated, meanwhile the `L` and `I` are not followed by numerals smaller than themselves, so they add themselves to our total for a sum of -10+50+1=41. However, if we consider a similar looking number `XLIV` this number actually represents 44, because the `I` here is actually negative due to immediately preceding the larger numeral `V`, leaving us with a sum of -10+50-1+5=44.

You must write the program `numerals.c` which reads *one* number written in Roman numerals from the standard input stream, ignoring leading whitespace, and then prints out that number in the Arabic numeral system (that is, print out its integer value as we are used to seeing integers). You may assume that the roman numeral read in will be able to within the bounds of an integer.

Deliverables: For this question include in your final submission zip your c source code file named `numerals.c`

How to submit: Create a zip file `a2.zip`, make sure that zip file contains your C source code files `rotate.c`, `luhns.c`, and `numerals.c`. Assuming all three of these files are in your current working directory you can create your zip file with the command

```
$ zip a2.zip rotate.c luhns.c numerals.c
```

Upload your file `a2.zip` to the a2 submission link on eClass.