PIC 10A Section 2 - Homework #7 (due Monday, November 21, by 11:59 pm)

You should upload each .cpp and .h file separately and submit them to CCLE. Submit this before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and the .cpp files requested of you.

Be sure you upload files with the precise names that you used in your editor environment and those requested of you in this assignment, otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2022.

ARBITRARY PRECISION FOR UNSIGNED INTEGERS

This homework serves as further practice with functions. You will be writing functions that compute the sum of two nonnegative integers *in any base* and functions that convert between binary and decimal representations of arbitrarily large nonnegative integers represented as **std::string**s.

Read this carefully: as you should be aware, the integers we store as **int**, **unsigned long long int**, etc., only have a finite number of bytes. There is a maximum possible size they can accommodate before overflow happens, etc. *The entire purpose of this assignment* is that you provide functionality to do some simple integer arithmetic for integers whose lengths may be arbitrarily long. *You will not be able* to get away with using ordinary built-in integer types to do most of the calculations. You have been warned.

A **main.cpp** file is provided to you, as are **Numbers.h** and **Numbers.cpp**. In this assignment, you will modify both **Numbers.h** and **Numbers.cpp** and resubmit these two files. You will not submit **main.cpp**; this contains the main routine your homework must work for.

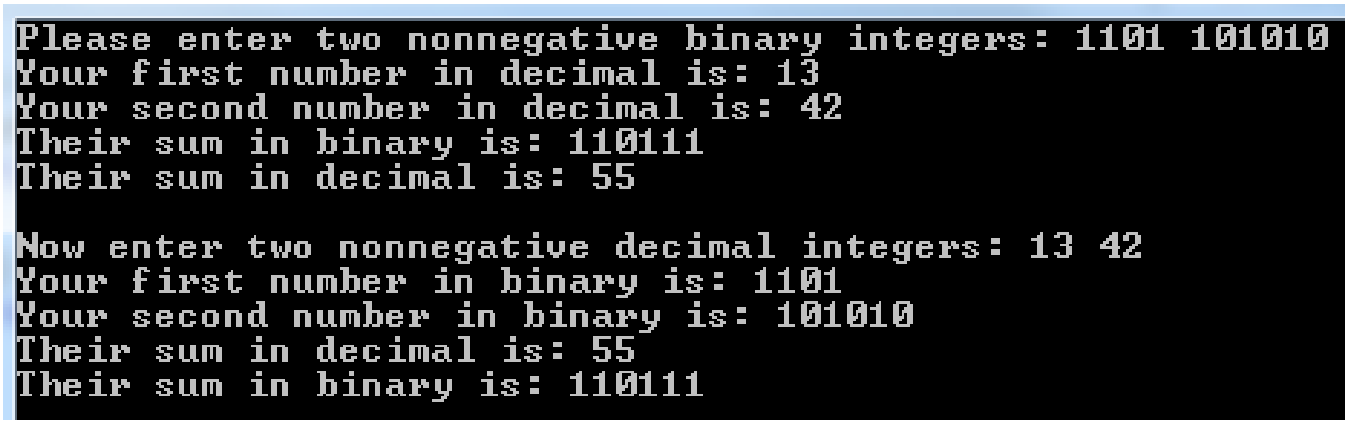
Note that some implementation steps are already provided for you. The minimum requirements for this assignment (but you can add more functions if you like) are to

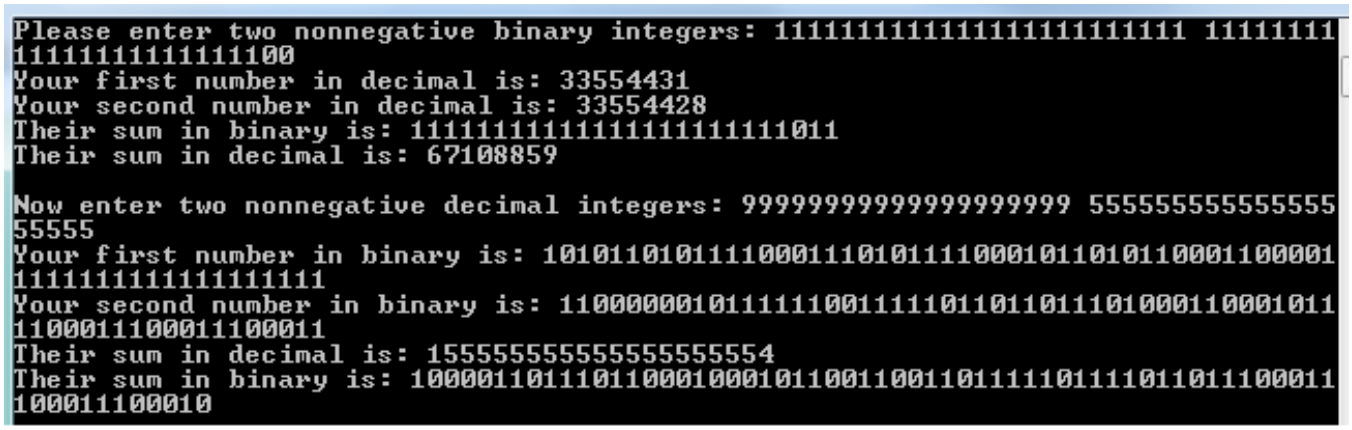
- provide a declaration and definition for **decimalToBinary** that accepts a string representation of a nonnegative integer in decimal and outputs its binary representation as a **std::string**;
- provide a declaration and definition for **binaryToDecimal** that accepts a string

representation of a nonnegative integer in binary and outputs its decimal representation as a **std::string**;

- provide a declaration and definition for **reverseString** that can be passed an **std::string**, reverses the characters of the input, and returns nothing;
- provide a definition for **add** that has already been declared in the header file;
- provide a definition for **charToDigit** that has already been declared in the header file;
- provide a definition for **digitToChar** that has already been declared in the header file; and
- provide a definition for **twoPower** that has already been declared in the header file.

Your program must be able to work on arbitrarily long integers, numbers far bigger than **ULLONG_MAX**, the largest possible **unsigned long long int**. See the example outputs.



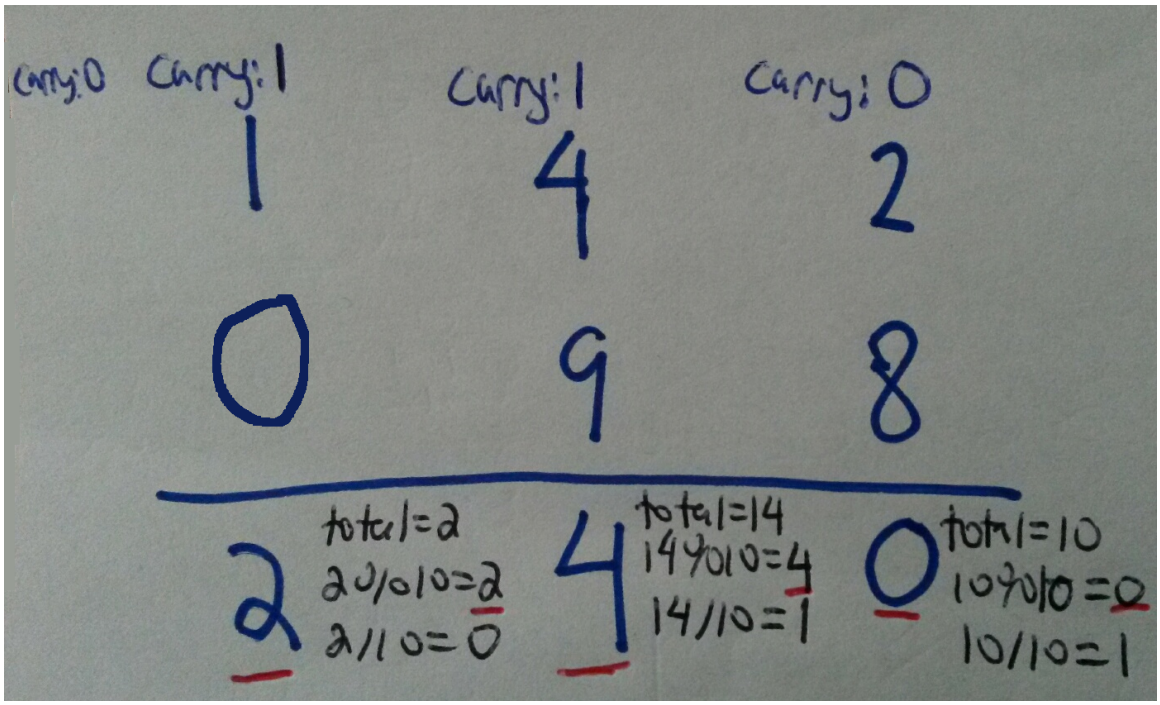


Important Hints and Procedures

You should read this part carefully so you can have a solid footing in how to approach these implementations.

Adding two numbers in a given base Here is the pseudo-code. Also see figure below.

- Obtain **num1** and **num2**
- If one string is shorter than the other, copy it, prepend it with 0's so they have the same length, and now consider adding the two strings with the same length
- Make **result** = ""
- Set **carry = 0**
- For **index** = last to first
  - Compute **total = carry + num1[index] + num2[index]**
  - Append **total % base** to **result**
  - Set **carry = total / base** via integer division
- If **carry** $\neq$ **0**, append it to the end of **result**
- Reverse **result**.



Converting decimal to binary Here is a description of the methodology to convert **dec** to binary. Also see worked example.

Find the largest power of 2, **maxPower** so that $2^{\text{maxPower}} \leq$ dec.

Starting with **pow = maxPower**, if $2^{\text{pow}} \leq$ dec then place a "1", subtract 2^{pow} from dec, and decrement **pow**. Otherwise, place a "0" and decrement **pow**.

Stop the procedure after **pow** is 0.

Example: consider converting 11_{10} to binary. We find:

$$\begin{aligned} 2^0 = 1 &\leq 11 \\ 2^1 = 2 &\leq 11 \\ 2^2 = 4 &\leq 11 \\ 2^3 = 8 &\leq 11 \\ 2^4 = 16 &> 11 \end{aligned}$$

so **maxPow** is **3**.

With **pow** as **3**: $2^8 \leq 11$, so we have a **"1"**, and we subtract to find **11-8 = 3** and decrement **pow**.

With **pow** as **2**: $2^2 > 3$, so we have **"10"** and we decrement **pow**.

With **pow** as **1**: $2^1 \leq 3$, so we have **"101"**, and we subtract to find **3 - 2 = 1** and we decrement **pow**.

With **pow** as **0**: $2^0 \leq 1$, so we have **"1011"**, and we subtract to find **1-1 = 0** and we stop because we are finished with **pow=0**.

Converting binary to decimal Recall that a number written in binary such as 11011 means, $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$.

You'll need to figure out the logic here.

Hints/directions: You should write the functions in the following order:

- **charToDigit**.
- **digitToChar**.
- **reverseString**: this is the easiest and shortest function. You can check it works by passing it a valid **std::string** object and then checking the value of the object after the function call. Remember: it should modify its input.
- **add**: this will require some time, but the other two functions depend upon **add**; even **subtract** depends upon **add**. Once you have written it, you can test it by **cout << add("3", "11", 10);**, for example, because it should work and output 14.
- **twoPower**: start a result at "1". Keep adding the result to itself as many times as the power requires. That's the answer. Pay attention here: the result starts at "1", not 1. You should be using your own functions with appropriate calls to **add**.
- **binaryToDecimal**: this is a very easy function to write once **add** is done. Then you can test the first half of the main routine provided.
- **decimalToBinary**: this is moderately harder than **binaryToDecimal**, but the prior functions should be a good warm-up to writing this one.