Homework #5

Due date: 12/2

Binary-coded decimal

Binary-coded decimal (BCD) is a format for representing decimal integers in which each decimal digit is stored in a 4-bit *nibble* using the common binary encoding.

By packing two BCD digits into one byte, the 4-byte BCD encoding of 4573 is

```
00000000 00000000 01000101 01110011       (BCD encoding)
                   4   5    7   3
```

Notice that the 32-bit pattern shown above represents the decimal integer 4573, rather than the hexadecimal integer $0x4573$.
Also, compare it with the usual 4-byte binary encoding of 4573:

```
00000000 00000000 00010001 11011101       (Binary encoding)
```

$\text{N.B.}\ 4573 = 4096 + 256 + 128 + 64 + 16 + 8 + 4 + 1$

In this homework, you are asked to perform the addition and subtraction of decimal integers in the following way:

Step 1	Convert decimal integers to equivalent BCDs
Step 2	Add or subtract BCDs to obtain a result in BCD format
Step 3	Convert the resulting BCD back to an equivalent decimal integer

Observe that step 1 performs the conversion: binary encoding → BCD encoding, whereas step 3 does the reverse: BCD encoding → binary encoding.

Here are the type and functions required for this homework:

```
typedef unsigned bcd,decimal;
bcd dec2bcd(decimal);           // convert a decimal to a BCD
decimal bcd2dec(bcd);           // convert a BCD to a decimal
bcd add(bcd,bcd);               // add two BCDs
bcd sub(bcd a,bcd b);           // subtract BCD b from BCD a
```

With these functions and the declarations

`decimal a,b;`

the addition

`a + b`

is performed by

`bcd2dec(add(dec2bcd(a),dec2bcd(b))).`

Similarly, the subtraction

`a - b`

is performed by

`bcd2dec(sub(dec2bcd(a),dec2bcd(b))).`

BCD addition

You may choose between the following two algorithms.

Algorithm A – Elementary pencil-and-paper addition algorithm

For example,

```
    1  1        ← carry
   4  5  7  3
 + 2  4  7  6
 ------------
   7  0  4  9
```

Algorithm B – Addition adjusted by 6

Step 1 First, add the two BCDs using *binary addition*

Step 2 For each pair of decimal digits whose sum causes a carry (as in Algorithm A), add 6 to the sum using *binary addition*

For example,

```
   00000000 00000000 01000101 01110011   (4573)
 + 00000000 00000000 00100100 01110110   (2476)
 -------------------------------------
   00000000 00000000 01101001 11101001   (result of step 1)
 + 00000000 00000000 00000110 01100000   (adjusted by 6)
 -------------------------------------
   00000000 00000000 01110000 01001001   (7049)
```

Observe that the result of step 1 is incorrect (it isn't even a valid BCD, as the red-colored number exceeds 9. Step 2 corrects it by adding 6 (the blue-colored number) to each position that causes a carry.

BCD subtraction

You may choose between the following two algorithms.

Algorithm A – Elementary pencil-and-paper subtraction algorithm
For example,

```
       1  1      ← borrow
    4  5  7  3
  - 2  4  7  6
  ------------
    2  0  9  7
```

Algorithm B – Subtraction adjusted by 6

Step 1 First, subtract the two BCDs using *binary subtraction*

Step 2 For each pair of decimal digits whose difference causes a borrow (as in Algorithm A), subtract 6 from the difference using *binary subtraction*

For example,

```
   00000000 00000000 01000101 01110011   (4573)
 - 00000000 00000000 00100100 01110110   (2476)
 --------------------------------------
   00000000 00000000 00100000 11111101   (result of step 1)
 - 00000000 00000000 00000000 01100110   (adjusted by 6)
 --------------------------------------
   00000000 00000000 00100000 10010111   (2097)
```

Requirements

1 With 4-byte unsigned integers, the BCDs range from 0 to 99,999,999. In case of overflow, the result of the addition or subtraction shall cause a wrap-around (Just like unsigned integer overflows in C/C++). Put differently, you shall compute

`(a + b) mod 100000000`, and

`(a - b) mod 100000000`

For examples,

`1 + 99999999` shall yield 0, and

`0 - 1` shall yield 99999999.

Hint

This is an easy task, as no **`mod`** operation is needed. What you have to do is to ignore the carry out of the most significant digit and feel free to borrow from the place next to the most significant digit, as illustrated on the next page.

```
 1 ← ignore this carry          1 ← feel free to borrow
  0 0 0 0 0 0 0 1                0 0 0 0 0 0 0 0
+ 9 9 9 9 9 9 9 9              - 0 0 0 0 0 0 0 1
-----------------              -----------------
  0 0 0 0 0 0 0 0                9 9 9 9 9 9 9 9
```

2 See sample run below for the require input and output formats. In particular, observe that there may have zero or more spaces before or after the operator symbols **+** and **−**. You may assume all inputs are of correct format.

Sample run

```
Enter a+b or a-b: 1234567 + 8765433
1234567 + 8765433 = 10000000

Enter a+b or a-b: 80000000-1
80000000 - 1 = 79999999

Enter a+b or a-b: 99887766   -   99887766
99887766 - 99887766 = 0

Enter a+b or a-b: 22334455 +5545
22334455 + 5545 = 22340000

Enter a+b or a-b: 22334455- 4456
22334455 - 4456 = 22329999

Enter a+b or a-b: 1 + 99999999
1 + 99999999 = 0

Enter a+b or a-b: 0 - 1
0 - 1 = 99999999

Enter a+b or a-b: ^Z
```