Objectives

- Implement a data type named **MegaInt** that models arbitrary large integers
- Implement a MegaInt calculator
- Experience operator overloading in a realistic setting
- Get continued practice using the the STL sequential container classes, as well as algorithms, iterators, C++ strings, and I/O facilities
- Practice programming

Your Task

Write an interactive C++ program that simulates a calculator that works with arbitrary large integer values.

Your program should start the calculator as follows:

```
1 int main()
2 {
3    MegaCalc mc; // Create a calculator,
4    mc.run(); // use it,
5    return EXIT_SUCCESS; // done, report success
6 } // not so fast, mc's destructor may throws!
```

The **mc** object starts the calculator at line 4; it accepts input from the keyboard, performs the desired operation, and displays the resulting value on the screen. The displayed value is called the called the *accumulator*. The calculator always uses the accumulator as one of the operands in the next operation. Specifically, if the next operation is a binary operation then the accumulator is used as the left operand in that operation, and if the next operation is a unary operation then the accumulator is used as the only operand in that operation.

Every input line must begin with a single character command. If the command is a binary operator, then it must be followed by an unsigned or a signed integer. For unary commands, the input line must begin and end with the command. Tabs and spaces can appear anywhere in a command line, even within the input number. The command (q) ends the calculator run. Finally, an empty command line repeats the previous command.

Here is the output produced by a sample run of the program:

```
Welcome to MegaCalculator
=========================

Accumulator: +0
Enter input:    =   +   1  2   3

Accumulator: +123
Enter input: *-12345678901234567890123456789O

Accumulator: -151851850485185185048511851850470
Enter input: /-123

Accumulator: +12345678901234567890123456789O
Enter input: =7

Accumulator: +7
Enter input: h
> +7
> +22
> +11
> +34
> +17
> +52
> +26
> +13
> +40
> +20
> +10
> +5
> +16
> +8
> +4
> +2
> +1

length of the hailstone(+7) sequence: +17

Accumulator: +7
Enter input: -2

Accumulator: +5
Enter input: f
+1! = +1
+2! = +2
+3! = +6
+4! = +24
+5! = +120

Accumulator: +5
Enter input: * 111 222 333 444 555 666 777 888 999

Accumulator: +556111667222778333889444995
Enter input: /5

Accumulator: +111222333444555666777888999
Enter input: q
```

Command list

Input	*Command*	*Operand*	*Operation*	Description
$+i$	$+$	i	$a \leftarrow a + i$	add
$++i$	$+$	$+i$	$a \leftarrow a + i$	add
$+-i$	$+$	$-i$	$a \leftarrow a + (-i)$	add
$-i$	$-$	i	$a \leftarrow a - i$	subtract
$-+i$	$-$	$+i$	$a \leftarrow a - i$	subtract
$--i$	$-$	$-i$	$a \leftarrow a - (-i)$	subtract
$*i$	$*$	i	$a \leftarrow a * i$	multiply
$*+i$	$*$	$+i$	$a \leftarrow a * i$	multiply
$*-i$	$*$	$-i$	$a \leftarrow a * (-i)$	multiply
$/i$	$/$	i	$a \leftarrow a/i$	divide
$/+i$	$/$	$+i$	$a \leftarrow a/i$	divide
$/-i$	$/$	$-i$	$a \leftarrow a/(-i)$	divide
$\%i$	$\%$	i	$a \leftarrow a\%i$	modulus
$\%+i$	$\%$	$+i$	$a \leftarrow a\%i$	modulus
$\%-i$	$\%$	$-i$	$a \leftarrow a\%(-i)$	modulus
$=i$	$=$	i	$a \leftarrow i$	reset a
$=+i$	$=$	$+i$	$a \leftarrow i$	reset a
$=-i$	$=$	$-i$	$a \leftarrow (-i)$	reset a
c	c		$a \leftarrow 0$	clear a
n	n		$a \leftarrow -a$	negate a
f	f		$a \leftarrow a!$	factorial of a
h	h		$hailstone($ a $)$	prints hailstone(a), returns its length. Click here for info.
q	q			quit

- 'a' denotes the accumulator value
- 'i' denotes the input number, which may or may not be signed
- input lines like $?+i$ are effectively the same as $?i$, where ? denotes a binary command.

Programming Requirements

a) No dynamic storage allocation; that is, no use of the **new** or **delete** operators.

b) No C-style raw arrays.

c) Must use the STL **deque** sequence container to store the digits of the numbers.

Here is why: the STL offers five sequence container class templates, namely, **array**, **vector**, **deque**, **forward_list**, and **list**.

The **array** class template is of no use in this assignment because it allows only fixed sized storage, whereas our mega numbers can have arbitrary large number of digits.

You will recall from elementary school pencil and paper methods that the sum, difference, and the product of two integers are formed from right to left but the quotient of the division from left to right. Therefore, ruling out **forward_list**, you look for containers that not only let you scan the digits of a number in both directions but also let you insert and delete digits at both ends of that number efficiently. That requirement rules out **vector**, leaving you with **deque** and **list** to choose from. Like **vector**, **deque** provides `operator[]` overloads but **list** does not, meaning that **deque** objects are somewhat easier to use than those of **list**. Although you know that **list** allows efficient insertion and deletion in the middle of the container and **deque** does not, you decide that you have no use for that feature because you only need to push and pop digits at the ends of a number and never in the middle. That will leave you with **deque** as an ideal container class for storing and manipulating the digits of your numbers.

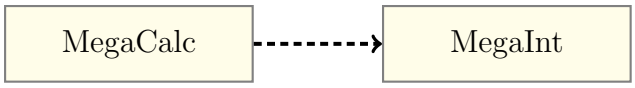
d) Write a class named **MegaInt** to model arbitrary large integers using the sign-magnitude notation. Equip **MegaInt** with the same operators supported by the built-in data types such as **int**. Your implementation does not have to be highly optimized, but it should not be too inefficient either. For example, you might not want to pack a storage unit with multiple digits, but you don't want to waste storage either. Thus, use a **char** to store a single digit instead of a **long**, or an **int**. To facilitate grading of your work, name your container object as follows:

```
deque<char> mega_int; // stores a finite but arbitrary large integer
```

Obviously, storing digits in **char**s necessitates conversion between **char** and **int**. For example:

```
int i7 { 7 };
char c0 { '0' };
//char c7 = char{ i7 + c0 }; // 'int' to 'char' requires a narrowing conversion
char c7 = static_cast<char>( i7 + c0 ); // reassure compiler that we know we are narrowing
mega_int.push_back(c7); // store digit 7 as character '7'
// ...
int digit = mega_int[0] - int{ c0 }; // convert from char '7' back to int 7
// ...
// hide the back and forth conversion mess into ypur operator[] overloads
// or other private/protected member functions.
```

e) Using the dependency relationship shown below implement a class named **MegaCalc** to provide and manage the user interface:



MegaCalc simply accepts user input from the console, determines the command to be performed, and then delegates the command directly to **MegaInt** for answers. If **MegaCalc** offers services such as *fatorial(n)* and *hailstone(n)* that **MegaInt** does not provide, then it implements them itself. For example:

A static member of class **MegaCalc**

```
MegaInt MegaCalc::factorial(const MegaInt& n)
{
   const MegaInt one{"1"}; // similar to int int m{1}
   MegaInt mega_fact{ one }; // similar to int mega_fact {1}
   for (MegaInt mega_k = one; mega_k <= n; ++mega_k) // uses op=, op<=, op++ overloads
   {
      mega_fact *= mega_k; // op*=
      cout << mega_k << "! = " << mega_fact << endl; // op<<
   }
   return mega_fact; // uses copy ctor
}
```

Thus, **MegaCalc** includes a **run()** member function to run the calculator (as suggested by the driver code on page 1), and two `static` functions that implement *factorial(n)* and *hailstone(n)*. Fill free to provide your own helper member functions but include them as `private` or `protected` members.

Suggestions

a) To get a feel for the operations you intend to implement, practice them using pencil and paper.

b) Implement addition and subtraction operations first. Test.

c) To get all four basic arithmetic operations up and running quickly, implement multiplication in terms addition (see algorithm M1) and division in terms subtraction (see algorithm D1). Test. Once you have implemented and test all the operators, implement M2) and D2).

d) Test your addition and multiplication together by computing 50!, which is:

```
30414093201713378043612608166064768844377641568960512000000000000
```

e) As an ultimate test, count the length of the sequence generated by *hailstone(n)*, where $n = 50!$. For example, the length of the sequence "3, 10, 5, 16, 8, 4, 2, 1" generated by *hailstone(3)* is 8. Note that *hailstone(n)* starts at n and ends at 1.

Basic Arithmetic with Large Decimal Integers

1 Abstract

This note presents an introduction to basic arithmetic operations on arbitrary large decimal integers. Using the sign–magnitude representation of integers, it describes algorithms for addition and subtraction of integers. It also provides information that can be useful in designing algorithms for multiplication, division, and modulus on large integers.

Although the algorithms presented here use the decimal base $b = 10$, they remain essentially the same when applied to integers in any number system with $b \geq 2$. Analysis of efficiency of the algorithms presented is straightforward and is left to the reader.

2 Sign–Magnitude Representation of Integers

The most natural and simplest way to represent signed integers is to use the *sign–magnitude* representation, which resembles the way people habitually write numbers. For example, `+10349` represents an integer with `+` as its sign and `10349` as its magnitude (or absolute value). The magnitude can be expressed in any number system with base $b \geq 2$. Without loss of generality, we use the familiar decimal number system in base 10, which has ten digits represented by the symbols 0,1,2,3,4,5,6,7,8, and 9.

Formally, the sign–magnitude representation of an n-digit integer $\mathbf{D_n}$ is expressed as

$$\mathbf{D}_n = s_d D_n = s_d(d_0 d_1 d_2 \cdots d_{n-1}) \qquad n \geq 1 \tag{1}$$

where s_d represents the *sign* of $\mathbf{D}_n$, and $D_n = (d_0 d_1 d_2 \cdots d_{n-1})$, a sequence of n decimal digits, represents the *magnitude* (or absolute value) of $\mathbf{D}_n$. Depending on its sign, $\mathbf{D}_n$ may be be expressed as $-D_n$, $+D_n$, or simply D_n when the sign is $+$. Thus, $D_n = (d_0 d_1 d_2 \cdots d_{n-1})$ denotes an unsigned integer, which by definition is a *non-negative* (≥ 0) integer. In a context where the number of digits n is implied or not needed, the notations $\mathbf{D}$ and D are used for $\mathbf{D_n}$ and D_n, respectively. The notation s_d^c denotes the complement of s_d.

Despite its simplicity, the sign–magnitude representation creates computational problems. It allows two different representations `-0` and `+0` for the number `0`, forcing the systems using the representation to handle the two representations identically. That is, software and hardware systems using this representation require additional software code and/or hardware circuitry to support this "double" representation for the number zero.

Another problem with the sign–magnitude representation is that it allows leading zeros in the representation of numbers. For example, $12345 - 12333 = 00012$. The leading zeros are

not significant; they serve only as placeholders to show the scale of a number. For example, the numbers `+0123` and `+000000123` have different scales but the same magnitude `123`.

In sign–magnitude representation, therefore, special care must be taken when performing arithmetic and relational operations on numbers with leading zeros. In such operations, the problems caused by the presence of leading zeros can be avoided by simply *normalizing* the numbers before operating on them. That is, by removing the leading zeros so that the leftmost digit (d_0) is non-zero whenever $n > 1$. Thus, `+0` and `+123` are in normalized form, but `+00000` and `+0123` are not.

3 Addition and Subtraction

3.1 Background

Let $\mathbf{A} = s_a A$ and $\mathbf{B} = s_b B$ be two integers in sign–magnitude representation. The object is to define addition and subtraction algorithms for computing the integers $\mathbf{A} + \mathbf{B}$ and $\mathbf{A} - \mathbf{B}$.

In terms of the magnitudes A of $\mathbf{A}$ and B of $\mathbf{B}$, the actual computations of $\mathbf{A}+\mathbf{B}$ and $\mathbf{A}-\mathbf{B}$ are each carried out in terms of one of the four forms $A+B$, $A-B$, $B-A$, $-(A+B)$. Since A and B are non-negative integers, $A + B$ is an unsigned integer. However, unless $A = B$, one of the integers $A - B$ and $B - A$ is unsigned and the other signed. To facilitate our tasks, we set out to avoid the signed case altogether.

For example, if $\mathbf{A} = +(5)$ and $\mathbf{B} = -(17)$, the two unsigned magnitudes are (5) and (17). Thus, we want to compute $\mathbf{A} + \mathbf{B}$ and $\mathbf{A} - \mathbf{B}$ using only one of the non-negative integers (5) + (17), (17) + (5), and (17) − (5), and avoiding (5) − (17). We avoid the operation (5) − (17) as follows:

$$\mathbf{A} + \mathbf{B} = +(5) + -(17) = -(17) + +(5) = -(+(17) - +(5)) = -((17) - (5)) = -(B - A)$$

$$\mathbf{A} - \mathbf{B} = +(5) - -(17) = (5) + (17) = +(A + B)$$

The ideas above can be generalized as shown in the following algorithms:

$$\mathbf{A}+\mathbf{B} = \begin{cases} s_a(A+B) & \text{if } s_a = s_b \\ s_a(A-B) & \text{if } s_a \neq s_b \text{ and } A > B \\ s_a^c(B-A) & \text{if } s_a \neq s_b \text{ and } A < B \\ +(0) & \text{if } s_a \neq s_b \text{ and } A = B \end{cases} \qquad \mathbf{A}-\mathbf{B} = \begin{cases} s_a(A+B) & \text{if } s_a \neq s_b \\ s_a(A-B) & \text{if } s_a = s_b \text{ and } A > B \\ s_a^c(B-A) & \text{if } s_a = s_b \text{ and } A < B \\ +(0) & \text{if } s_a = s_b \text{ and } A = B \end{cases}$$

The desired effect is that we always end up adding two unsigned integers and subtracting a smaller unsigned integer from a larger one, each resulting in an unsigned integer.

Therefore, to actually compute **A** + **B** or **A** - **B**, we are going to need two pairs of algorithms we name **add** and **plus**, and **subtract** and **minus**. Algorithms **plus** and **minus** are private helpers that perform the actual *unsigned* operations. Algorithms **add** and **subtract** provide user interface.

3.2 Addition and Subtraction Algorithms

Algorithm add(A, B)
Input : $\mathbf{A} = s_a A$ and $\mathbf{B} = s_b B$
Output: $\mathbf{C} = s_c C$ where $\mathbf{C} = \mathbf{A} + \mathbf{B}$

```
1  if s_a = s_b then
2      s_c = s_a
3      C = plus(A, B)
4  else
5      if A > B then
6          s_c = s_a
7          C = minus(A, B)
8      else if A < B then
9          s_c = s_a^c
10         C = minus(B, A)
11     else
12         s_c = +
13         C =(0)
14 Normalize C
15 return C
```

Algorithm subtract(A, B)
Input : $\mathbf{A} = s_a A$ and $\mathbf{B} = s_b B$
Output: $\mathbf{C} = s_c C$ where $\mathbf{C} = \mathbf{A} - \mathbf{B}$

```
1  if s_a ≠ s_b then
2      s_c = s_a
3      C = plus(A, B)
4  else
5      if A > B then
6          s_c = s_a
7          C = minus(A, B)
8      else if A < B then
9          s_c = s_a^c
10         C = minus(B, A)
11     else
12         s_c = +
13         C =(0)
14 Normalize C
15 return C
```

In C++ you can hide the algorithm pairs **add** and **plus**, and **subtract** and **minus** altogether, by providing them only through overloaded operators. Specifically, you implement `operator+=` using **add**, and `operator-=` using **subtract**.

Consider using similar techniques to separate interface and representation for multiplication and division. For example, define **multiply(A, B)** and **product(A, B)** where **multiply** provided user interface and **product** computes the product; then hide them both, and implement `operator*=` using **multiply**.

Note that in C++ you can always define an arithmetic *operatorX* using *operatorX=*.

3.3 Helper Algorithm plus

Algorithm plus(X, Y)

Input : $X = (x_0x_1x_2 \cdots x_{m-1})$, $Y = (y_0y_1y_2 \cdots y_{n-1})$, with $n \geq 1$ and $m \geq 1$.
Output : $Z = (z_0z_1z_2 \cdots z_{p-1})$, where $p = Max(m, n) + 1$, and $Z = X + Y$.

1 $carry \leftarrow 0$
2 $i \leftarrow p - 1$
3 $j \leftarrow m - 1$
4 $k \leftarrow n - 1$
5 **while** $(j \geq 0$ && $k \geq 0)$ **do**
6 $\quad t \leftarrow x_j + y_k + carry$
7 $\quad z_i \leftarrow t\%10$
8 $\quad carry \leftarrow t/10$
9 $\quad i \leftarrow i - 1$
10 $\quad j \leftarrow j - 1$
11 $\quad k \leftarrow k - 1$
12 // Propagate last *carry* to unprocessed portion of X
13 **while** $(j \geq 0)$ **do**
14 $\quad t \leftarrow x_j + carry$
15 $\quad z_i \leftarrow t\%10$
16 $\quad carry \leftarrow t/10$
17 $\quad i \leftarrow i - 1$
18 $\quad j \leftarrow j - 1$
19 // Propagate last *carry* to unprocessed portion of Y
20 **while** $(k \geq 0)$ **do**
21 $\quad t \leftarrow y_k + carry$
22 $\quad z_i \leftarrow t\%10$
23 $\quad carry \leftarrow t/10$
24 $\quad i \leftarrow i - 1$
25 $\quad k \leftarrow k - 1$
26 // Complete the operation
27 $z_0 \leftarrow carry$
28 `return` Z

Algorithm 1: Algorithm plus(X, Y)

Example:

carry →	+	0	10	010	1010	11010	111010	111010
left operand →	+	627	627	627	627	627	627	627
rightoperand →	+	99567	99567	99567	99567	99567	99567	99567
sum →			4	94	194	0194	00194	100194

3.4 Helper Algorithm minus

Algorithm minus(X, Y)

Input : $X = (x_0x_1x_2\cdots x_{m-1})$, $Y = (y_0y_1y_2\cdots y_{n-1})$, $n \geq 1$, $m \geq 1$, and $X \geq Y$.

Output : $Z = (z_0z_1z_2\cdots z_{p-1})$, where $p = Max(m, n)$, and $Z = X - Y$

```
1  borrow ← 0
2  i ← p − 1
3  j ← m − 1
4  k ← n − 1
5  while j ≥ 0 and k ≥ 0 do
6  |  t ← x_j − (y_k + borrow)
7  |  borrow ← 0
8  |  if t < 0 then
9  |  |  borrow ← 1
10 |  |  t ← 10 + t
11 |  z_i ← t
12 |  i ← i − 1
13 |  j ← j − 1
14 |  k ← k − 1
15 // Propagate last borrow to unprocessed portion of X
16 while j ≥ 0 do
17 |  t ← x_j − borrow
18 |  borrow ← 0
19 |  if t < 0 then
20 |  |  borrow ← 1
21 |  |  t ← 10 + t
22 |  z_i ← t
23 |  i ← i − 1
24 |  j ← j − 1
25 if borrow = 1 or k ≥ 0 then
26 |  throw "X cannot be less than Y in (X − Y)" // Impossible! |X| < |Y| in (X − Y)
27 else
28 |  return Z
```

Example:

X: left operand →	+	32045	32045	32045	32045	32045	32045
Y: right operand →	-	327	327	327	327	327	327
borrow →	-	0	10	010	1010	01010	001010
		———	———	———	———	———	———
Z: result →			8	18	718	1718	31718

4 Multiplication

The object is to compute $a = b \times c$ where the operands b and c are arbitrary long decimal integers, one called *multiplicand* and the other *multiplier*.

Here are only three alternative methods for computing $a = 345 \times 6789$. For efficiency purposes, we choose as *multiplier* the smaller of b and c and denote it by m, and the other as *multiplicand* and denote it by n. thus, $m = 345$ and $n = 6789$.

M1. Simply compute the result of adding n to itself m times; that is, compute $a = \sum_{k=1}^{m} n$. For example, $a = 6789 * 345 = \sum_{k=1}^{345} 6789 = 2342205$. This is not an efficient method especially when a and m are large and close each other.

M2. This is the traditional paper and pencil algorithm taught at elementary schools, which is by far more efficient than method M1 above. It is also easy to implement.

6789	←	*multiplicand n*	
345	←	*multiplier m*	*partial sums*
			0
33945	←	*multiply 5 by n and then left-shift 0 positions*	
		add 33945 *to partial sum* →	33945
271560	←	*multiply 4 by n and then left-shift 1 position*	
		add 271560 *to partial sum* →	305505
2036700	←	*multiply 3 by n and then left-shift 2 positions*	
		add 2036700 *to partial sum* →	2342205
2342205	→	*product*	

M3.

This method[a] uses integer addition and comparison operations only. The method starts by listing in a column headed 2^k all successive powers of 2 not greater than m; that is, starting with 2^0 and stopping at 2^k, where k is the largest integer such that $2^k \leq m$. In a parallel column headed $2^k n$, it lists n, $2n$, $4n$,$\cdots$, $2^k n$. Next, the method marks those entries in the column headed 2^k that add up to m. Finally, to compute a, it sums the entries in the column headed $2^k n$ that face the marked entries in the column headed 2^k.

2^k		$2^k n$
1	→	6789
2		13578
4		27156
8	→	54312
16	→	108624
32		217248
64	→	434496
128		868992
256	→	1737984
345		2342205
↑		↓
m		a

[a] This method, and the method D_3 on the following page, are based on binary arithmetic. They were invented by the ancient Egyptians and in use until around 540. The mathematical justification is straightforward and is left to the curious students.

5 Division

Given two integers $n \geq 0$ and $d > 0$, the object is to compute the quotient q and remainder r such that $n = dq + r$ with $0 \leq r < d$. Thus, n denotes the *dividend* and d the *divisor*. Here are some alternative algorithms.

D1. Repeatedly reduce n by d as long as the successive remainders are greater than or equal zero. The number of reductions made gives q, and the last remainder gives r.

D2. This algorithm is based on the traditional paper and pencil method we learned at elementary school for dividing n by d.[1] Specifically, it is based on a trivial observation that the traditional paper and pencil method computes the quotient q *one digit at a time* from left to right, $q_0, q_1, \cdots$. Each digit q_k in q in turn is the quotient of a division in which *the dividend* n' *has at most one digit more than the digits in the divisor* d.

Here is an example, where $n = 1594347364730$ and $d = 7777$. Since, there are 4 digits in d, we initially pick the first 4 digits $n' = 1234$ from n to start the division process:

Given $n = 1594347364730$ and $d = 7777$ compute $q = n/d$ and $r = n\%d$

k	n	n'	$n' < d$	q_k	$d * q_k$	$r' = n' - d * q_k$	q in progress
0	1594347364730	1594	✓	0	0	1594	0
1	1594347364730	15943		2	15554	389	02
2	1594347364730	3894	✓	0	0	3894	020
3	1594347364730	38947		5	38885	62	0205
4	1594347364730	623	✓	0	0	623	02050
5	1594347364730	6236	✓	0	0	6236	020500
6	1594347364730	62364		8	62216	148	0205008
7	1594347364730	1487	✓	0	0	1487	02050080
8	1594347364730	14873		1	7777	7096	020500801
9	1594347364730	70960		9	69993	967	0205008019

Thus, $q = 1594347364730/7777 = 205008019$ and $r = 1594347364730\%7777 = 967$, both normalized.

Note that the leading digits in n' shown in black come form r' in the previous row.

A check-marked (✓) entry in a row k indicates a trivial case where the current dividend is less than the divisor (i.e., $n' < d$), implying the trivial solution $q_k = 0$ and $r' = n'$.

[1] See `http://en.wikipedia.org/wiki/Long_division`

Now this tricky question: how do we compute the single-digit quotients q_k when $n' \geq d$? For example, how do we decide that $q_9 = 9$ in the last row or $q_8 = 1$ in the row above it? The answer is we follow the paper and pencil tradition and take "an initial guess" $q_k^{'}$ and then refine $q_k^{'}$ towards the actual answer q_k, which is always a single-digit from 1 through $9 = 10 - 1$.

In base 10, we might decide to run the guesses $q_k^{'}$ from 9 through 1 and stop as soon as we find a $q_k^{'}$ for which the remainder r' is ≥ 0 or equivalently $n' \geq d * q_k^{'}$. For example, when $n' = 70960$, we find $q_9 = q_9^{'} = 9$ immediately, but when $n' = 15943$, we find $q_1 = q_1^{'} = 2$ after 8 tries.

To reduce the number of tries in base 10, we may start with the initial guess in the middle at $q_k^{'} = 5$, getting the initial remainder $r' = n' - d * 5$. Then, if $r' = 0$ then we have the answer $q_k = 5$ with $r' = 0$. If $r' < 0$ we keep decrementing our guess $q_k^{'}$ and compute r' until $r' \geq 0$; otherwise (i.e., $r' > 0$), we keep incrementing $q_k^{'}$ until we find a $q_k^{'}$ for which $r' \leq d$.

However, for higher bases, we would get help from the leading digits in n' and d in order to take an "educated" initial guess for q_k.[2] It is not uncommon for an arbitrary precision calculators or a computer programs to use every bit in an **unsigned int** of size, say, 4 bytes to represent all 4294967296 "digits" in base $2^{32} = 4294967296$.

D3. This method is essentially the reverse of method M3 above. It starts by listing in a column, headed $m_j = 2^j d$, the power 2 multiples of d, which consist of the values d, $2d$, $4d$, $8d$, $\cdots$, $m_k = 2^k d$ such that k is the largest integer for which $m_k \leq n$. It also lists in a parallel column headed 2^j the binary powers 2^0, 2^1, $\cdots$, 2^k.

To compute r, the method subtracts from n all multiples of d in k steps in the bottom-up direction, producing a list of non-negative remainders r_k, r_{k-1}, $\cdots$,r_1, r_0, starting with $r_k = n - m_k$. For $j = k-1, k-2, \cdots, 1, 0$, if $r_{j+1} \geq m_j$ then m_j is marked and $r_j = r_{j+1} - m_j$; otherwise, $r_j = r_{j+1}$. Finally, $r = r_0$.

To compute q, it sums the binary powers 2^j that correspond to the marked m_j entries.

The figure shown below illustrates the entire process in the context of an example where $n = 10,000,000$ and $d = 1234$. After the column headed $m_j = 2^j d$ has been completed, we find $k = 12$. To further clarify the process, we construct a column headed r_j listing the successive remainder values, starting at the bottom with $r_{12} = n - m_{12}$ and going upward and ending at the top with r_0.

[2]See Donald Knuth, Vol. 2, Sec. 4.3. page 250.

j	2^j		$m_j = 2^j d$	r_j		
0	1	$\leftarrow$	1234	$r_1 - m_0$	898	$\rightarrow r$
1	2	$\leftarrow$	2468	$r_2 - m_1$	2132	
2	4	$\leftarrow$	4936	$r_3 - m_2$	4600	
3	8		9872	r_4	9536	
4	16		19744	r_5	9536	
5	32	$\leftarrow$	39488	$r_6 - m_5$	9536	
6	64		78976	r_7	49024	
7	128	$\leftarrow$	157952	$r_8 - m_7$	49024	
8	256	$\leftarrow$	315904	$r_9 - m_8$	206976	
9	512	$\leftarrow$	631808	$r_{10} - m_9$	522880	
10	1024	$\leftarrow$	1263616	$r_{11} - m_{10}$	1154688	
11	2048	$\leftarrow$	2527232	$r_{12} - m_{11}$	2418304	
12	4096	$\leftarrow$	5054464	$n - m_{12}$	4945536	$\uparrow$
	8103	$\rightarrow q$				

Sample program run 2: MegaClac computing 50!

```
Accumulator: +50
Enter input: f
+1! = +1
+2! = +2
+3! = +6
+4! = +24
+5! = +120
+6! = +720
+7! = +5040
+8! = +40320
+9! = +362880
+10! = +3628800
+11! = +39916800
+12! = +479001600
+13! = +6227020800
+14! = +87178291200
+15! = +1307674368000
+16! = +20922789888000
+17! = +355687428096000
+18! = +6402373705728000
+19! = +121645100408832000
+20! = +2432902008176640000
+21! = +51090942171709440000
+22! = +1124000727777607680000
+23! = +25852016738884976640000
+24! = +620448401733239439360000
+25! = +15511210043330985984000000
+26! = +403291461126605635584000000
+27! = +10888869450418352160768000000
+28! = +304888344611713860501504000000
+29! = +8841761993739701954543616000000
+30! = +265252859812191058636308480000000
+31! = +8222838654177922817725562880000000
+32! = +263130836933693530167218012160000000
+33! = +8683317618811886495518194401280000000
+34! = +295232799039604140847618609643520000000
+35! = +10333147966386144929666651337523200000000
+36! = +371993326789901217467999448150835200000000
+37! = +13763753091226345046315979581580902400000000
+38! = +523022617466601111760007224100074291200000000
+39! = +20397882081197443358640281739902897356800000000
+40! = +815915283247897734345611269596115894272000000000
+41! = +33452526613163807108170062053440751665152000000000
+42! = +1405006117752879898543142606244511569936384000000000
+43! = +60415263063373835637355132068513997507264512000000000
+44! = +2658271574788448768043625811014615890319638528000000000
+45! = +119622220865480194561963161495657715064383733760000000000
+46! = +5502622159812088949850305428800254892961651752960000000000
+47! = +258623241511168180642964355153611979969197632389120000000000
+48! = +12413915592536072670862289047373375038521486354677760000000000
+49! = +608281864034267560872252163321295376887552831379210240000000000
+50! = +30414093201713378043612608166064768844377641568960512000000000000

Accumulator: +50
Enter input:
```

6 Deliverables

1. Header files: **MegaInt.h** and **MegaCalc.h**
2. Implementation files: **MegaInt.cpp**, **MegaCalc.cpp**, **mega_driver.cpp**
3. A **README.txt** text file (see the course outline).

7 Marking scheme

50%	Arithmetic operations 10% addition 10% subtraction 10% multiplication (M1 is worth only 5%, M2 10%) 10% division (D1 is worth only 5%, D2 10%) 10% modulus (D1 is worth only 5%, D2 10%)
30%	Arithmetic operators `+=, +, -=, -, *=, *, /=, /, %=, %, ++, --` Relational operators: `==, <, !=, <=, >, >=` Subscript operators `[]` `operator<<`
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program