HW #5 (Overloading: 45-digit bigint)

- A well-known drawback of the native 32-bit int type in C++ is that it cannot store large integers of more than 10 decimal digits.
- In this project, you will implement in C++ a **bigint** class that behaves like the **unsigned int** type, but can handle unsigned integers of up to 45 digits.

- Big integers have found widely varying applications in scientific and societal domains, such as recording distances between stars or planets in the universe, and keeping track of the number of atoms or cells in organisms.

HW #5 (2)

- You are to build a class named **bigint** that can deal with up to **45** decimal digits of unsigned integers. The **bigint** class is specified as follows.
- The **bigint** class should have a **five**-element array, (**int v[5];**), as its **private** data member. Each element can store **9** decimal digits in the range of 0 up to 999999999.
- For example, to store the integer 1,300,000,000,000,900,000,000,100,000,000. You should have the internal storage as v[**0**] = 100000000 (the **last** nine digits), v[**1**] = 900000000, v[**2**] =0, v[**3**]= 1300, and v[**4**] = 0. This way, we can store successfully a large enough integer with up to 45 digits.

HW #5 (3)

- The **bigint** class should support the following constructors:

  bigint(); // default constructor, set the value to be 0

  bigint(int x0); // set the value to be x0

  bigint(int x0, int x1); // set the value to be $x1*10^9+x0$

  // set the value to be $x2*10^{18}+x1*10^9+x0$

  bigint(int x0, int x1, int x2);

  // set the value to be $x3*10^{27} + x2*10^{18}+x1*10^9+x0$

  bigint(int x0, int x1, int x2, int x3);

  // set the value to be $x4*10^{36} + x3*10^{27} + x2*10^{18}+x1*10^9+x0$

  bigint(int x0, int x1, int x2, int x3, int x4);

HW #5 (4)

- To make **bigint** behave like int, you must overload the following operators:
- The extraction operator **>>** for reading a **bigint** number from an input stream. The input number can have up to 45 digits. Following are two **bigint** numbers that can appear as input:

  1300000000009000000100000000

  999999999999999999999999999999999999999
- To implement the >> operator, you can first read the stream of digits into a character array (or string) and then manually convert it into a **bigint** object. Here you may use **cin.get()** or **getline()** functions to accept the input from input streams.

HW #5 (5)

- The insertion operation **<<** for output a **bigint** number to an output stream. The number must be output like a regular **int** number starting from most significant bits (v[**4**]) all the way down to least significant bits (v[**0**]).

- Here you may need to output potentially 5 integers. You may use **setw(9)** and **setfill('0')** to output each value besides the most significant non-zero integer. If so, you have to have '#include <iomanip>' in your code in order to use these functions.

HW #5 (6)

- The following line is an example to use these functions.

  s << setfile('0') << setw(9) << a.v[j];

- For instance,

  bigint x(1, 123);

  cout << x << endl;

- The expected output will be 123000000001 (the last 1 is padded with eight 0s on its left).

HW #5 (7)

- Two arithmetic operators + and – ;
- For addition (a+b), please pay special attention to the carry bit to the higher significant bits. Moreover, adding two **bigint**s, a and b, may result in overflows, if a.v[4]+b.v[4] > 999999999. In this case, we will omit the carry bit (to the even higher significant bits), and only keep the remaining raw values as is. For example:

900000000 000000000 000000000 000000000 900000000

\+ 900000000 000000000 000000000 000000000 900000000

= 800000000 000000000 000000000 000000001 800000000

- Note here 900000000 + 900000000 = 1800000000, while the leading carry bit is omitted because of overflow.

HW #5 (8)

- For subtraction (a-b), if a < b, the result will be 0, because we only use **bigint** to maintain unsigned integers. If a >= b, we follow the regular arithmetic logic of subtraction for calculation, and please pay special attention to the carry bit during the computation.

- Six comparison operators <, >, <=, >=, ==, and !=
- **bigint** objects are still integers, so they can be compared and ordered. Overloading these operators will support comparisons between different **bigint** objects.

```
// test.cpp --> a.out
#include <iostream>
#include "bigint.h“

using namespace std;

void test1() {
  bigint x;
  bigint y;

  if (x == y) cout << x << " is equal to " << y << "." << endl;
  if (x != y) cout << x << " is not equal to " << y << "." << endl;
  if (x > y) cout << x << " is larger than " << y << "." << endl;
  if (x >= y) cout << x << " is larger than or equal to " << y << "." << endl;
  if (x < y) cout << x << " is smaller than " << y << "." << endl;
  if (x <= y) cout << x << " is smaller than or equal to " << y << "." <<
     endl;
  cout << endl;
```

```
bigint x1(123456789, 111, 111, 111, 111);
bigint y1(111111111, 111, 111, 111, 111);

if (x == y) cout << x << " is equal to " << y << "." << endl;
if (x != y) cout << x << " is not equal to " << y << "." << endl;
if (x > y) cout << x << " is larger than " << y << "." << endl;
if (x >= y) cout << x << " is larger than or equal to " << y << "." << endl;
if (x < y) cout << x << " is smaller than " << y << "." << endl;
if (x <= y) cout << x << " is smaller than or equal to " << y << "." <<
   endl;
cout << endl;
```

```
bigint x2(999999999, 999999999, 999999999, 20);
cout << x2 << "+1 = " << x2 + 1 << endl;

bigint y2(000000000, 000000000, 000000000, 000000000,
   000000001);
cout << y2 << "-1 = " << y2-1 << endl;

cout << "x1 + x2 - y2 + 1 = " << x1+x2-y2+1 << endl;
cout << endl;
```

```
  for (int i=0; i<3; i++) {
     bigint x, y;
     cin >> x >> y;
     cout << "x = " << x << endl;
     cout << "y = " << y << endl;

     cout << "x+y=" << x+y << endl;
     cout << "x-y=" << x-y << endl;
  }
}

int main() {
  test1();
  return 0;
}
```

a.out < test.txt > output.txt

test.txt is as follows:

```
11111119999999999999999999
10000000000000000000001
22222200000000000000000000
10000000000003000000002
30000002000000000000000001
20000001000000000000000002
```

output.txt is as follows:

0 is equal to 0.

0 is larger than or equal to 0.

0 is smaller than or equal to 0.

11100000011100000011100000011123456789 is not equal to 11100000011100000011100000011111111111.

11100000011100000011100000011123456789 is larger than 11100000011100000011100000011111111111.

11100000011100000011100000011123456789 is larger than or equal to 11100000011100000011100000011111111111.

20999999999999999999999999+1 = 21000000000000000000000000

10000000000000000000000000000000-1 = 99999999999999999999999999999999

x1 + x2 - y2 + 1 = 11000000013200000011100000011123456789

x = 11111119999999999999999999
y = 10000000000000000000001
x+y=11211120000000000000000000
x-y=11011119999999999999999998
x = 22222200000000000000000000
y = 10000000000003000000000002
x+y=23222200000000030000000002
x-y=21222199999999969999999998
x = 30000002000000000000000001
y = 20000001000000000000000002
x+y=50000003000000000000000003
x-y=10000000999999999999999999