# Computer Science 1 - Program 1

## *Big Integer*

## Assigned: 1/12/11

## Due: 1/26/11 (Wednesday) at 11:55pm (WebCourses time)

**The Problem**

The `unsigned int` type in C requires 4 bytes of memory storage. With 4 bytes we can store integers as large as $2^{32}$-1; but what if we need bigger integers, for example ones having hundreds of digits? If we want to do arithmetic with such very large numbers we cannot simply use the `unsigned` data type, because there is not enough space, in an `int`, to store that large of a value. So what is a workaround? One way of dealing with this is to use a different storage structure for integers, such as an array of digits. We can represent an integer as an array of digits, where each digit is stored in a different array index. Since the integers are allowed to be as large as we like, a dynamically-sized array will prevent the possibility of overflows in representation. However we need new functions to add, subtract, compare, read and write these very large integers.

Write a program that will manipulate such arbitrarily large integers. Each integer should be represented as an array of digits, where the least significant digit is stored in index 0. Your program should we able to read in a string of digits and create a struct that stores the big integer.

Your program should store each decimal digit (0-9) in a separate array element. In order to perform addition and subtraction more easily, it is better to store the digits in the array in the reverse order. For instance, the value `1234567890` would be stored as:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| array | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Note: Although this seems counter-intuitive, it makes the code slightly easier, because in all standard mathematical operations, we start with the least significant digits. When you add 123 and 415 to get 538, you start adding with the least significant digit; meaning, you start by adding the 3 and the 5 to get the 8 in the final answer. Now, with arrays, if there is some operation that you want to perform over each cell of the array, do you want to start performing this operation at the beginning of the array or at the end of the array? Of course, it is easier to start performing any indicated operations at the beginning of the array, and then you loop through the array and perform operations as needed. As a result, since you want to start working on the array from the front of the array AND since you start adding/subtracting numbers from the LEAST significant digit, it makes the most sense that the number is stored in REVERSE order (as depicted above), with the least significant digit being stored in the first cell of the array

.

Your program should include the following functions:

- a function that will convert a string to a struct integer
- a function that will print an integer
- a function that will add two integers and return the result
- a function that compares two integers and returns -1 if the first is less than the second, 0 if they are equal, and 1 if the first is greater than the second
- a function that will subtract one integer from the other and return the result. Since you'll be dealing with positive integers only, the result should be a positive number. To ensure that the result is integer you should subtract the smaller number from the larger one. If they are equal, 0 should be returned

**Input/Output Specification**

**Instead of getting input from the user, you read in input from a file, "bigint.txt".** The name MUST BE "bigint.txt". Have this AUTOMATED. Do not ask the user to enter "bigint.txt". You should read in this automatically. (This will expedite the grading process.)

The input file format is as follows:

The first line will contain a single positive integer, *n*, representing the number of operations to carry out. The next *n* lines will contain one problem each. Each line will have three integers separated by white space. The first integer on each of these lines is guaranteed to either be 1 or 2, indicating addition and subtraction, respectively. The next two integers on the line will be the two operands for the problem. You may assume that these two integers are non-negative, are written with no leading zeros (unless the number itself is 0) and that the number of digits in either of the numbers will never exceed 200.
(Note: Thus, the answer of the operation will never exceed 201 digits.)

*****NOTE***: You should generate your output to a FILE that you will call "out.txt".** In particular, you should generate one line of output per each input case. Your output should fit one of the two following formats:

```
X + Y = Z
X – Y = Z
```

corresponding to which option was chosen. For the first option, always print the first operand first. For the second option, always print the larger of the two operands first. If the two operands are equal, the same number is printed both times.

*****WARNING*****

Your program MUST adhere to this EXACT format (spacing capitalization, use of dollar signs, periods, punctuation, etc). The graders will use very large input files, resulting in very large output files. As such, the graders will use text comparison programs to compare your output to the correct output. If, for example, you have two spaces between in the output when there should be only one space, this will show up as an error even through you may have the program correct. You WILL get points off if this is the case, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as the graders will be forced to go to text editing of your program in order to give you an accurate grade. So as an example, here is the first format shown above (if the instruction was an addition):

```
X + Y = Z
```

First we have one of the “integers”. Then we have ONE space. Then we have a plus sign. Then we have ONE space. Then we have the second “integer”. Then we have ONE space. Then we have the equals sign. Then we have ONE space. Finally, we have the answer with NO spaces afterwards.

Again, your output MUST ADHERE EXACTLY to the line shown above.

**Implementation Restrictions**

You must use the following struct:

```
struct integer {
     int* digits;
     int size;
}
```

Whenever you store or return a big integer, always make sure not to return it with any leading zeros. Namely, make sure that the value stored in index size-1 is NOT zero. The only exception to this rule is if 0 is being stored. 0 should be stored in an array of size 1.

Here are the prototypes of the functions for you to write:

```
//Preconditions: the first parameter is string that stores
//                only contains digits, doesn't start with
//                0, and is 200 or fewer characters long.
//Postconditions: The function will read the digits of the
//              large integer character by character,
//              convert them into integers and return a
//              pointer to the appropriate struct integer.
struct integer* convert_integer(char* stringInt);

//Preconditions: p is a pointer to a big integer.
//Postconditions: The big integer pointed to by p is
//                 printed out.
void print(struct integer *p);

//Preconditions: p and q are pointers to struct integers.
//Postconditions: A new struct integer is created that
//                 stores the sum of the integers pointed to
//                 by p and q and a pointer to it is
//                 returned.
struct integer* add(struct integer *p, struct integer *q);

//Preconditions: p and q are pointers to struct integers.
//Postconditions: A new struct integer is created that
//                 stores the absolute value of the
//                 difference between the two and a pointer
//                 to this is returned.
struct integer* subtract(struct integer *p, struct integer
*q);

//Preconditions: Both parameters of the function are
//                pointers to struct integer.
//Postconditions: The function compares the digits of two
//              numbers and returns:
//    -1 if the first number is smaller than the second,
//     0 if the first number is equal to the second number,
//   1 if the first number is greater than the second.
int compare(struct integer *p, struct integer *q);
```

**Sample Input File**

```
3
1 8888888888 2222222222
2 9999999999 10000000000
2 10000000000 9999999999
```

**Corresponding Output**

```
8888888888 + 2222222222 = 11111111110
10000000000 – 9999999999 = 1
10000000000 – 9999999999 = 1
```

**Deliverables**

Turn in a single file, *bigint.c*, over WebCourses that solves the specified problem. If you decide to make any enhancements to this program, clearly specify them in your header comment so the grader can test your program accordingly.

**Grading Details**

Your program will be graded upon the following criteria:

1) Adhering to the implementation specifications described here.
2) Your algorithmic design.
3) Correctness.
4) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 80-85% on it.)
5) Compatibility to Dev C++. (If your program does not compile in Dev C++, you will get a sizable deduction from your grade.)
7) Your output MUST adhere to the EXACT FORMAT shown in this document.

**Restrictions**

Name the file you create and turn in *bigint.c*. Although you may use other compilers, your program must compile and run using Dev C++. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date. You should also include comments throughout your code, when appropriate.