TP C#11 : Tiny Bistro

Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tpcs11-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- TinyBistro/
        |-- TinyBistro.sln
        |-- TinyBistrp/
            |-- Tout sauf bin/ and obj/
```

— You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
— The code **MUST** compile.
— In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space) :

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal :

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 Presentation

Does the word "BigNum" remind you of something ? Do you remenber ? The midterm of Caml ? It was a long time ago, but you have a new opportunity to understand and implement operations on potentially infinite numbers. That is the purpose of the Bistromathic, of which you are going to make a miniature version this week.
If the origin of Bistromathic intrigued you, do not hesitate to take a look at "The Hitchhiker's Guide to the Galaxy".

1.2 Objectifs

The goal of this TP is to implement a small part of the famous Bistromathic (tiny...). Indeed, your program has only to handle operations between two unsigned numbers. But doing operations with signed numbers is a bonus !

2 Course (Reminder)

2.1 Operator Overload

As seen in TPC#9, it is possible to overload operators to perform new operations on objects. This is the key of this TP ! You'll find functions looking like this in the skeleton of the TP :

```
1 public static BigNum operator +(BigNum a, BigNum b)
2 {
3     // Some bright code was deleted her (yes I promise)
4     return new BigNum(/*stuff*/)
5 }
```

Then it is possible to perform operations as follows :

```
1 BigNum a = /* ... */
2 BigNum b = /* ... */
3 BigNum c = a + b;
```

2.2 Primary school maths level

Yes, you read right. It is not because you did it in primary school that it's easy to implement. Before starting to code, take the time to do some basic operations like kids do. It will be very useful.

3 Exercices

3.1 BigNum

In this part, you need to implement the **BigNum** class which represents a potentially infinite number. As you know, the integers we can manipulate (int, uint64 (...)) are encoded on a specific number of bits and, therefore, have a size limit (cf TPC#0). For example, I can not manipulate a 30-digit number with an int.

To overcome this, you will use a list to store each digit of the number. If you have followed your algorithm courses carefully, you know that it is better to insert an item at the end of a list rather than at the start. Thus, to facilitate operations and gain in complexity, the numbers will be stored upside down. The number 12563 will be in the list [3][6][5][2][1] and the number 0 will be represented by an empty list [ ].

Your class will then contain a private list of int named **digits**.

3.1.1 Constructor

The first step is to implement the constructor of a BigNum. This one takes a string containing the parsed number as parameter and builds the list. The number stored in the string is not upside down. If the string is incorrect, an ArgumentException must be triggered.

```
1 public BigNum(string number)
```

Here is an example :

```
1 string number = "25555";
2 BigNum test = new BigNum(number); //the list is [5][5][5][5][2]
```

3.1.2 GetNumDigits

Now you have to implement some help function that will be very useful to manipulate BigNums.

```
1 public int GetNumDigits()
```

This method returns the number of digits in a BigNum. For the number 128469, the function returns 6.

3.1.3 AddDigit

```
1 public void AddDigit(int digit)
```

This method takes as parameter a digit that has to be added at the end of the list of digits (thus in the first position of the number written in the normal sense). Example :

```
1 BigNum a = new BigNum("12345"); //[5][4][3][2][1]
2 a.AddDigit(8); //[5][4][3][2][1][8]
```

3.1.4 GetDigit

```
1  public int GetDigit(int position)
```

This method returns the digit stored in the list at the position given in parameter. If the position is greater or equal to the number of digits in the BigNum, you must throw an exception of type OverflowException. Example :

```
1  BigNum a = new BigNum("12345"); //[5][4][3][2][1]
2  a.GetDigit(8); //OverflowException
3  a.GetDigit(0); //returns 5
```

3.1.5 SetDigit

```
1  public void SetDigit(int digit, int position)
```

This method takes an integer (between 0 and 9) in argument as well as a position and allows the user to change the value of the ith digit of the list. If ever the position of the digit to add is greater than the number of digits in the BigNum, add as many zeros as necessary. Be careful to remove the last digits from the list that are zeros at the end of this function. Example :

```
1  BigNum a = new BigNum("12345"); //[5][4][3][2][1]
2  a.SetDigit(8, 7); //[5][4][3][2][1][0][0][8]
3  a.SetDigit(2, 5); //[5][4][3][2][1][2][0][8]
4  a.SetDigit(0, 7); //[5][4][3][2][1][2]
```

3.1.6 Print

```
1  public void Print()
```

This method prints the BigNum in a readable way. Don't forget to handle the case of 0.

3.1.7 Comparisons

In this part, you have to implement the operations of comparisons between two BigNum :

```
1  public static bool operator <(BigNum a, BigNum b)
```

```
1  public static bool operator >(BigNum a, BigNum b)
```

```
1  public static bool operator ==(BigNum a, BigNum b)
```

```
1  public static bool operator !=(BigNum a, BigNum b)
```

Prototypes are explicit, but here is an example :

```
1  BigNum a = new BigNum("888654");
2  BigNum b = new BigNum("9512");
3  Console.WriteLine(a < b); //false
4  Console.WriteLine(a > b); //true
5  Console.WriteLine(a == b); //false
6  Console.WriteLine(a != b); //true
```

3.1.8 Operations

In this part, you will implement basic operations on BigNum, which are +, -, *, /, %. Implementation is free, but optimization will be rewarded, so we will give you some algorithms names. And in case of problems : https ://www.google.com/

```
1 public static BigNum operator +(BigNum a, BigNum b)
```

The primary school method is enough ! And do not forget the deductions !

```
1 public static BigNum operator -(BigNum a, BigNum b)
```

Just like for addition, primary school method is enough. Reminder, the right number will always be smaller than the left one.

```
1 public static BigNum operator *(BigNum a, BigNum b)
```

Here it gets complicated. Firstly, we advice you to implement the primary school method. However, the Karatsuba and Fast Fourier (FFT) algorithms are much more efficient and will earn bonus points !

```
1 public static BigNum operator /(BigNum a, BigNum b)
```

The primary school method is still a good option. But if you like challenges : The D algorithm.

```
1 public static BigNum operator %(BigNum a, BigNum b)
```

If the other methods work, it should not take you more than a minute to code !

3.2 Bonus

There are many possible bonuses for this TP. You can start by optimizing your operations using better algorithms, then you can add a sign field in your BigNum and manage signed operations. You can also implement power or square root ! Take the time to implement bonuses, and write them down in the README.

These violent deadlines have violent ends.