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/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

1.1 Présentation

Est-ce que le mot "BigNum" vous dit quelque chose ? Vous vous souvenez ? Le contrôle de Caml ? C'était il y a longtemps, mais voici une nouvelle chance de comprendre et d'implémenter des opérations sur des nombres potentiellement infinis. C'est le principe de la Bistromathique, dont nous (vous) allons réaliser une version miniature cette semaine.
Si l'origine et les principes de la Bistromathique vous intrigue, n'hésitez pas à jeter un coup d'œil à "The Hitchhiker's Guide to the Galaxy".

1.2 Objectifs

L'objectif de ce TP est d'implémenter une petite partie de la fameuse Bistromathique (d'ou le "tiny"...). En effet, votre programme n'aura qu'à gérer des opérations entre deux nombres de même signe. Mais n'oubliez pas, le faire est un bonus !

2 Cours (Rappel)

2.1 La surcharge d'opérateurs

Comme vu lors du TPC#9, il est possible de surcharger les opérateurs pour effectuer de nouvelles opérations sur des objets. C'est là dessus que va reposer une grande partie de ce TP ! Vous trouverez donc dans le squelette des fonctions de la forme :

```
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  }
```

Et ainsi il est ensuite possible d'effectuer des opérations comme suit :

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

2.2 Maths niveau primaire

Oui vous avez bien lu. Ce n'est pas parce que vous faisiez ça au primaire que ça en devient facile à implémenter. Avant de vous lancer dans du code, prenez le temps de poser des opérations basiques, comme quand vous étiez enfants. Ça vous sera très utile.

3 Exercices

3.1 BigNum

Dans cette partie, vous allez devoir implémenter la classe **BigNum** qui représente un nombre potentiellement infini. Or, comme vous le savez, les entiers que nous pouvons manipuler (*int, uint64* (...)) sont codés sur un certain nombre de bits et ont donc une limite de taille (cf TPC#0). Il m'est par exemple impossible de manipuler un nombre à 30 chiffres avec un *int*.

Pour pallier cela, nous (vous !) allons utiliser une liste pour stocker chacun des chiffres de notre grand nombre. Si vous avez bien suivi vos cours d'algo, vous savez qu'il est préférable d'insérer un élément en queue de liste plutôt qu'en tête. Ainsi, pour faciliter les opérations et gagner en complexité, les nombres seront stockés à l'envers. Le nombre 12563 sera donc dans la liste [3][6][5][2][1] et le nombre 0 sera représenté par une liste vide [ ].

Votre classe contiendra donc une liste privée d'int nommée **digits**.

3.1.1 Constructeur

La première étape est de coder le constructeur d'un **BigNum**. Celui ci prend une *string* contenant un nombre parsé (plus tard) en paramètre et construit la liste. Le nombre stocké dans la *string* est dans le bon sens. Si jamais la string est incorrecte, une *ArgumentException* doit être lancée.

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

Voici un petit exemple :

```
1  string number = "25555";
2  BigNum test = new BigNum(number); //La liste est [5][5][5][5][2]
```

3.1.2 GetNumDigits

Vous allez maintenant implémenter des fonctions d'aide qui vont vous être très utiles dans la manipulation des **BigNum**.

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

Cette méthode retourne le nombre de chiffre que contient le **BigNum**. Pour le nombre 128469, cette fonctions retournerait 6.

3.1.3 AddDigit

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

Cette méthode prend en paramètre un chiffre à ajouter à la fin de la liste des chiffres du BigNum (donc en première position du nombre en écriture dans le bon sens). Exemple :

```
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)
```

Cette méthode retourne le chiffre stocké dans la liste à la position donnée en paramètre. Si la position est supérieure ou égale au nombre de chiffres dans le BigNum, vous devez lancer une exception de type OverflowException. Exemple :

```
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)
```

Cette méthode prend un entier (entre 0 et 9) en argument ainsi qu'une position et permet de changer la valeur du ième chiffre de la liste. Si jamais la position du chiffre à ajouter est plus grande que le nombre de chiffres dans le BigNum, rajoutez autant de zéros que nécessaire. Faites bien attention à retirer les derniers chiffres de la liste qui sont des zéros à la fin de cette fonction. Exemple :

```
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()
```

Cette méthode affiche le **BigNum** dans le bon sens. N'oubliez pas de gérer le cas du 0.

3.1.7 Comparaisons

Dans cette partie, vous allez devoir implémenter les opérations de comparaisons entre deux 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)
```

Ces prototypes parlent d'eux même, mais voici néanmoins quelques exemples :

```
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 Opérations

Dans cette partie, vous devez implémenter les opérations classiques sur des **BigNum**, qui sont +, -, *, /, %. A vous de choisir votre implémentation, nous allons nous contenter de vous mettre quelques noms d'algorithmes. Et en cas de problème : https ://www.google.fr/

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

N'allez pas chercher trop compliqué, la méthode primaire suffit ! N'oubliez pas les retenues !

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

Pareil que pour l'addition, la méthode primaire est suffisante. Pour rappel, le nombre de droite sera toujours plus petit que le nombre de gauche.

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

C'est ici que ça se complique. Le plus simple dans un premier temps est de faire à nouveau comme en primaire. Cependant, les algorithmes de Karatsuba et de Fourier rapide (FFT) sont largement plus efficaces et rapporteront des points bonus (et puis c'est la classe) !

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

Le primaire est toujours d'actualité ! Mais si vous aimez le challenge, The D algorithm.

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

Si le reste marche, ça ne devrait pas vous prendre plus d'une minute à coder !

3.2 Bonus

Il existe beaucoup de bonus possibles pour ce TP. Vous pouvez commencer par optimiser vos opérations en utilisant de meilleurs algorithmes, puis vous pouvez ajouter un champ signe dans vos BigNum et gérer les opérations selon les signes. Vous pouvez aussi implémenter la puissance ou la racine carrée ! Prenez le temps de faire des bonus, et notez les bien dans le **README**.

These violent deadlines have violent ends.