TP 3: BigNum

1 La classe BigNum

Pour représenter les nombres entiers en Java on utilise soit le type `int`, soit le type `long`, qui ont une capacité de stockage de 32 et 64 bits respectivement. Comment peut-on représenter des nombres pour lesquels 64 bits ne suffisent pas?

Le but de cet exercice est de programmer une classe de Java qui va être capable de stocker un nombre entier quelconque sans perte de precision. Autrement dit, votre classe doit être capable de stocker des valeurs enormes, sans problèmes de "overflow". En plus, votre classe doit donner à l'utilisateur la possibilité d'effectuer des opérations élémentaires dans les objets `BigNum`, comme l'addition/soustraction, la multiplication, la division, les comparaisons, etc.

Pour implémenter une telle classe vous pouvez utiliser des tableaux des `int`: chaque objet de la classe va contenir un tableau, de taille qui dépend de la valeur stockée. Chaque élément du tableau représentera un chiffre décimal du nombre. (**NB:** Pour simplifier les opérations d'affichage, vous pouvez utiliser que des éléments entre $0 - 9$, c'est-à-dire que des chiffres décimaux, même si ce n'est pas très efficace en termes d'utilisation de mémoire.)

La signature de la classe doit contenir au moins les éléments suivants.

1. Un constructeur `BigNum(int)` qui intialise un object de la classe qui représente le nombre entier donné, et un constructeur par défaut.
2. Une méthode `toString()`.
3. Une méthode `BigNum add(BigNum b)` qui retourne un nouveau `BigNum` dont la valeur et la somme de l'objet actuel et `b`.
4. Une méthode `BigNum subtract(BigNum b)` qui retourne un nouveau objet dont la valeur est la différence **absolue** entre l'objet actuel et `b`.
5. Une méthode `boolean equals(BigNum b)` qui decide si `b` représente la même valeur que l'objet actuel.
6. Une méthode `boolean isBigger(BigNum b)` qui fait une comparaison entre l'objet actuel et `b`.
7. Une méthode `BigNum multiply(BigNum b)` qui retourne un nouveau objet qui est le produit de l'objet actuel et `b`.
8. Une méthode `BigNum div(BigNum b)` qui fait de la division entiere entre deux BigNums.

Conseils

- Vous pouvez supposer que tous les nombres traités sont non-negatifs, et l'opération de soustraction retourne toujours un entier non-negatif.

- Faites attention aux algorithmes que vous allez utiliser pour la multiplication et la division. Une implémentation naïve (avec des additions/soustractions repetées) ne va pas être efficace. Rappelez-vous qu'on veut traiter des nombres énormes : pour calculer la valeur de $10^{75}/10^{25}$ on voudrait eviter de répéter l'opération $x - 10^{25}$, sinon il faut la répéter 10^{50} fois!
- Par contre, vous ne devez pas être super-efficaces en termes d'utilisation de mémoire (par exemple, vous pouvez utiliser un `int` pour représenter un seul chiffre décimal).

Vérification

Pour tester que votre classe fonctionne correctement, vous devez écrire quelques fonctions qui l'utilisent pour faire des opérations sur des entiers trop grands pour être stockés dans un `int`.

Donnez des fonctions pour les tâches suivantes:

1. Une fonction `static BigNum fact(int n)` qui retourne $n!$ en forme `BigNum`.
2. Une fonction `static BigNum fibo(int n)` qui retourne le n-ième terme de la suite de Fibonacci.
3. Une fonction `static BigNum pow(BigNum b, int n)` qui retourne b^n.

Pour verifier que toutes les fonctions et les méthodes de la classe fonctionnent correctement:

1. Donner une boucle qui affiche les 100 premiers termes de la suite de Fibonacci, et pour chaque terme n calcule la valeur de l'expression `fibo(n)-fibo(n-1)-fibo(n-2)` (qui doit évidemment être toujours 0).
2. Donner une boucle qui vérifie la formule du binôme de Newton pour les valeurs de $n \leq 100$. Pour faire cette vérification utilisez l'identité

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Rappel : l'opération $\binom{n}{k}$ est définie ainsi $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Vous devez donc écrire une boucle qui, étant donné n, calcule et affiche tous les termes $\binom{n}{i}$, et puis vérifie que leur somme est bien ègale à 2^n.