BigInteger : une bibliothèque pour entiers de très grande taille

Manuel de l'utilisateur

Plan du manuel :

1°) Résumé

BigInteger est une bibliothèque C++ utilisable pour programmer en C++ le traitement d'entiers de très grandes tailles, aussi grandes que nécessaire. On peut s'en servir en ajoutant au programme C++ à développer soit les fichiers biginteger.hpp et biginteger.lib ou soit les fichiers biginteger.hpp et biginteger.cpp.

En plus des opérations de base : addition, soustraction, multiplication, division, exponentiation et modulo cela permet d'effectuer diverses autres calculs parmi lesquels : critère de primalité, plus petit diviseur premier, plus grand commun diviseur, plus petit commun multiple, inverse modulaire, exponentiation modulaire, factorisation, etc. On peut aussi accéder à la représentation interne de ces entiers, en affecter la valeur à partir des entiers natifs de C++ ou d'un grand nombre défini par ses chiffres décimaux et en convertir la valeur en std::string en décimal, en hexadécimal ou dans toute autre base.

Les entiers BigInteger sont utilisables de la même manière que les entiers natifs de C++ seuls ou conjointement avec ceux-ci.

En interne, l'entier BigInteger est représenté par :
- son signe
- ses coefficients c[0] à c[n]

La valeur absolue de cet entier est : c[0] + c[1] B + c[2] B^2 + ... + c[n] B^n
La base B utilisable est une puissance de 2 au choix entre 8 et 30
Le signe (-1 ou +1) est rangé dans un int16_t : _signe et les coefficients dans un std::vector<int32_t> : _elems qui ne sont pas accessibles directement.

2°) Les déclarations

Voici plusieurs déclarations valides :

```
BigInteger a;
BigInteger b = 10;
BigInteger c = "12345678901234567890";
BigInteger d = i;    // i est un entier natif du C++
BigInteger n = c;    // c est un BigInteger
```

Le BigInteger est initialisé par défaut à 0 quand il n'est valorisé au moment de sa déclaration.

3°) Les opérateurs

Les opérateurs de BigInteger a un ou deux opérandes et pour les entrées-sorties sont les suivants. Il faut noter qu'il n'y a pas d'opérateur de décalage des bits du BigInteger mais il y a des fonctions qui permettent de les obtenir ou de les modifier.

3-1 Les opérateurs à un seul opérande

3-1-1 Calcul de l'opposé

L'opérateur : - permet d'obtenir l'opposé de l'entier désigné.

3-1-2 Les incrémentations et décrémentations

Ce sont pour les BigInteger les instructions :

```
n++   // incrémentation post-fixée
++n   // incrémentation pré-fixée
n--   // décrémentation post-fixée
--n   // décrémentation préfixée
```

Ils sont identiques à ceux du C++ pour les entiers natifs.

3-2 Les opérateurs à deux opérandes

3-2-1 Les six opérations de base

Les six opérations de base sont :

```
n = a + b;   //  pour l'addition
n = a - b;   //  pour la soustraction
n = a * b;   //  pour la multiplication
n = a / b;   //  pour la division
n = a % b;   //  pour le modulo
n = a ^ b;   //  pour l'exponentiation
```

Il y a aussi des fonctions qui calculent les mêmes résultats.
A noter que l'opérateur ^ signifie l'exponentiation quand l'un au moins de ses deux opérandes est un BigInteger, mais quand ses deux opérandes sont des entités natives du C++ cet opérateur signifie l'opération xor habituelle.

3-2-2 Les affectations

Les affectations sont :

```
n =  a;   // affectation de base
n += a;   // pour n = n + a
n -= a;   // pour n = n - a
n *= a;   // pour n = n * a
n /= a;   // pour n = n / a
n %= a;   // pour n = n % a
n ^= a;   // pour n = n ^ a
```

où l'opérande a est soit un BigInteger, soit un entier natif du C++ ou soit une std::string désignant un grand entier par la suite de ses chiffres.

3-2-3 Les comparaisons

Là aussi, ces opérateurs sont identiques à ceux du C++

```
n == a   // égal à
n != a   // non égal à
n >  a   // plus grand que
n >= a   // plus grand ou égal à
n <  a   // plus petit que
n <= a   // plus petit ou égal à
```

3-3 Les entrées-sorties

Les opérateurs std::cin et std::cout du C++ ont été surchargés pour les BigInteger. Ce qui permet de les utiliser pour les BigInteger comme pour les entiers du C++ par exemples :

```
std::cout << "Entrez n : ";
std::cin >> n;
std::cout << "n : " << n << std::endl;
std::ofstream ofst;
ofst.open(fichier);
ofst << "n : " << n << std::endl;
ofst.close();
std::ifstream ifst;
ifst.open(fichier);
ifst >> n;
ifst.close;
std::stringstream stst;
stst << "n : " << n << std::endl;
std::string message = stst.str();
```

4°) Les fonctions

En plus des fonctions, la bibliothèque BigInteger a quatre constantes qui sont utilisables :

```
const int32_t bitsbi = 20;            // nb de bits des c[]
const int32_t basebi = 1 << bitsbi;   // B = basebi = 2^bitsbi
const BigInteger binul = BigInteger(0); // BigInteger nul
const BigInteger bione = BigInteger(1); // BigInteger un
```

4-1 Pour gérer les bits du BigInteger

Pour obtenir le nombre total de bits du BigInteger x : `x.nBits()`
Pour obtenir le i-ième bit (true ou false) du BigInteger x : `x.getBit(int32_t i)`
Pour mettre à jour le i-ième bit du BigInteger x : `x.setBit(int32_t i, bool v)`
avec v = true pour y mettre : 1 et v = false pour y mettre : 0

4-2 Pour gérer séparément les coefficients du BigInteger

Pour obtenir le nombre de coefficients du BigInteger x : `x.getSize()`
Pour obtenir le i-ième coefficient du BigInteger x : `x.getElemt(int32_t i)`
Pour mettre à jour le i-ième coefficient du BigInteger x avec e < basebi : `x.setElemt(int32_t i, int32_t e)`

Pour supprimer le dernier coefficient du BigInteger x : `x.popElemt()`
Pour supprimer éventuellement les coefficients de x de rang élevé s'ils sont nuls et donc inutiles : `x.finbi()`

4-3 Pour gérer ensemble les coefficients du BigInteger

Pour obtenir tous les coefficients du BigInteger x : `x.getElems()`
le résultat est un : `std::vector<int32_t>`
Pour mettre à jour tous les coefficients de x avec chaque v[i] < basebi :
`x.setElems(std::vector<int32_t> v)`

4-4 Pour gérer le signe du BigInteger

Pour obtenir le signe, 1 ou -1, du BigInteger x : `x.getSigne()`
Pour changer le signe du BigInteger x : `x.chgSigne()`
Le signe de 0 est 1, il n'y a pas de BigInteger nul avec le signe négatif.

4-5 Pour mettre un BigInteger à 0 ou 1

Pour mettre à 0 le BigInteger x : `x.binul()`
Pour mettre à 1 le BigInteger x : `x.bione()`

4-6 Pour comparer deux BigInteger

Pour comparer deux BigInteger : `cmpbi(a, b)`
Le résultat est : -1 si a < b il est : 0 si a = b et : 1 si a > b

4-7 Les six opérations de base

Pour additionner deux BigInteger : `addbi(a, b)`
Pour soustraire deux BigInteger : `subbi(a, b)`
Pour multiplier deux BigInteger : `mulbi(a, b)` ou : `mulbk(a, b)` ou : `mulbt(a, b)`
Pour diviser deux BigInteger : `divbi(&q, &r, n, d)`
Pour le modulo de deux BigInteger : `modbi(n, m)`
Pour l'exponentiation de deux BigInteger : `expbi(n, p)`
La fonction `mulbi(a, b)` utilise en priorité la multiplication rapide basée sur la FFT (Fast Fourier Transform). Mais ce calcul nécessite l'emploi interne des nombres flottants, ce qui peut créer des erreurs d'arrondi pouvant compromettre le résultat. Toutefois cette éventualité est détectable : on a une variable disponible : `fftSquareWorstError` qui indique si l'erreur d'arrondi est acceptable quand elle vaut moins de 0.1 et il y a aussi une autre condition qui entraine la variable `fftOK` à la valeur `false.` Dans ce cas `mulbi(a, b)` enchaîne automatiquement sur la multiplication `mulbt(a, b)` traditionnelle. En plus, il y a aussi la multiplication `mulbk(a, b)` qui utilise l'algorithme de Karatsuba qui est meilleur que la méthode traditionnelle pour de très grands opérandes et qui s'emploie séparément.
La fonction `divbi(&q, &r, n, d)` a deux résultats, le quotient : q et le reste : r qui sont désignés par leurs adresses.
La fonction `expbi(n, p)` fonctionne par récursivité et l'opération : `n^p` aussi.

4-8 Quelques autres fonctions

Pour la racine carrée du BigInteger n : `sqrbi(&s, &r, n)` où le résultat : s et le reste : r sont deux BigInteger désignés par leurs adresses.
La valeur absolue du BigInteger x : `absbi(x)`
Le logarithme népérien du BigInteger x : `logbi(x)` est un long double.
Le factoriel du BigInteger x : `factbi(x)`
La primorielle du BigInteger x : `primbi(x)`
Le plus petit diviseur premier du BigInteger x : `ppdpbi(x)`
Le premier nombre premier suivant algébriquement le BigInteger x : `pnpsbi(x)`
Un diviseur du BigInteger x : `ndivbi(x)` Ce n'est pas toujours le plus petit diviseur ni même un diviseur premier, c'est seulement le premier diviseur trouvé.
La décomposition en facteurs premiers du BigInteger x : `fctpbi(x)`
le résultat est un `std::vector<BigInteger>`
Le pgcd de deux BigInteger : `pgcdbi(a, b)`
Le ppcm de deux BigInteger : `ppcmbi(a, b)`
Un BigInteger au hasard entre deux BigInteger : `randbi(a, b)`
L'inverse modulaire de x modulo m : `invmbi(x, m)` et s'il n'existe pas le résultat est nul.
L'exponentiation modulaire de x puissance p modulo m : `expmbi(x, p, m)` permet de bien plus grandes valeurs de l'exposant que l'exponentiation directe ne le permettrait.

4-9 Trois critères : le résultat est true ou false

Pour déterminer si le BigInteger x est négatif : `isnegabi(x)`
Pour déterminer si le BigInteger x est pair : `ispairbi(x)`
Pour déterminer si le BigInteger x est un nombre premier : `isprembi(x)`
Le critère est basé sur l'algorithme probabiliste de Miller et Rabin avec 12 témoins : s'il détecte l'argument en tant que nombre composé, c'est exact et s'il le détecte en tant que nombre premier c'est très probablement exact avec une très forte probabilité.

4-10 Les conversions

Pour convertir le BigInteger x en std::string : `bi2string(x, radix)` dans une base ayant radix : de 2 à 36 chiffres.
Pour convertir le BigInteger x en std::string en décimal : `bi2deci(x)`
Pour convertir le BigInteger x en std::string en hexadécimal : `bi2hexa(x)`
`bi2deci(x) = bi2string(x, 10)` et `bi2hexa(x) = bi2string(x, 16)` mais `bi2string()` est moins rapide que `bi2deci()` et `bi2hexa()`.
Pour convertir le BigInteger x en int64_t : `bi2int(x)` Le résultat vaut : `x%intmaxi` avec : `intmaxi = 9223372036854775807 = 0x7FFFFFFFFFFFFFFFLL`.

4-10 Les erreurs

La bibliothèque BigInteger est susceptible de signaler des erreurs de mise en œuvre qui sont assez souvent suivies d'une fin d'exécution du programme principal.

4-11 Ci-après le fichier : biginteger.hpp

```
// biginteger.hpp

#ifndef BIGINTEGER_HPP
#define BIGINTEGER_HPP

#include <algorithm>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <cstdint>
#include <cstddef>
#include <cstring>
#include <string>
#include <vector>
#include <cmath>

extern  bool  fftOK;
extern double fftSquareWorstError;

class BigInteger {

public:
    ~BigInteger();
    BigInteger();
    BigInteger(int32_t a);
    BigInteger(int64_t a);
    BigInteger(uint32_t a);
    BigInteger(uint64_t a);
    BigInteger(const std::string& a);
    BigInteger(const char* a);
    BigInteger(const BigInteger& a);
    int32_t nBits() const;
    bool getBit(int32_t n) const;
    void setBit(int32_t n, bool v);
    int32_t getSize() const;
    int32_t getElemt(int32_t n) const;
    void setElemt(int32_t n, int32_t e);
    void popElemt();
    void finbi();
    std::vector<int32_t> getElems() const;
    void setElems(std::vector<int32_t> v);
    int16_t getSigne() const;
    void chgSigne();
    void setbi0();
    void setbi1();
    BigInteger operator -  () const;
    BigInteger operator =  (BigInteger a);
    BigInteger operator += (const BigInteger& a);
    BigInteger operator -= (const BigInteger& a);
    BigInteger operator *= (const BigInteger& a);
    BigInteger operator /= (const BigInteger& a);
    BigInteger operator %= (const BigInteger& a);
    BigInteger operator ^= (const BigInteger& a);
    BigInteger operator ++ ();
    BigInteger operator ++ (int);
    BigInteger operator -- ();
    BigInteger operator -- (int);

private:
    // Le BigInteger est représenté par _signe et _elems :
    // _elems[0] + _elems[1]*basebi + _elems[2]*basebi^2 + ...
    // _elems[i] va de 0 à basebi-1 = (2^bitsbi)-1
    std::vector<int32_t> _elems;
    // signe = -1 ou +1 ( le signe de 0 est +1 )
    int16_t _signe;
};
```

```
BigInteger operator + (const BigInteger& l, const BigInteger& r);
BigInteger operator - (const BigInteger& l, const BigInteger& r);
BigInteger operator * (const BigInteger& l, const BigInteger& r);
BigInteger operator / (const BigInteger& l, const BigInteger& r);
BigInteger operator % (const BigInteger& l, const BigInteger& r);
BigInteger operator ^ (const BigInteger& x, const BigInteger& n);
bool operator == (const BigInteger& l, const BigInteger& r);
bool operator != (const BigInteger& l, const BigInteger& r);
bool operator >  (const BigInteger& l, const BigInteger& r);
bool operator >= (const BigInteger& l, const BigInteger& r);
bool operator <  (const BigInteger& l, const BigInteger& r);
bool operator <= (const BigInteger& l, const BigInteger& r);
std::ostream& operator << (std::ostream& ost, const BigInteger& a);
std::istream& operator >> (std::istream& ist, BigInteger& a);
int32_t cmpbi(const BigInteger& a, const BigInteger& b);
BigInteger addbi(const BigInteger& a, const BigInteger& b);
BigInteger subbi(const BigInteger& a, const BigInteger& b);
BigInteger mulbt(const BigInteger& a, const BigInteger& b);
BigInteger mulbk(const BigInteger& a, const BigInteger& b);
BigInteger mulbi(const BigInteger& a, const BigInteger& b);
BigInteger modbi(const BigInteger& a, const BigInteger& n);
BigInteger expbi(const BigInteger& a, const BigInteger& n);
void divbi(BigInteger* q, BigInteger* r, const BigInteger& n, const BigInteger& d);
void sqrbi(BigInteger* q, BigInteger* r, const BigInteger& n);
BigInteger absbi(BigInteger n);
long double logbi(BigInteger n);
BigInteger factbi(const BigInteger& n);
BigInteger primbi(const BigInteger& n);
BigInteger ppdpbi(const BigInteger& n);
BigInteger pnpsbi(const BigInteger& n);
BigInteger pgcdbi(const BigInteger& a, const BigInteger& b);
BigInteger ppcmbi(const BigInteger& a, const BigInteger& b);
BigInteger randbi(const BigInteger& a, const BigInteger& b);
BigInteger invmbi(const BigInteger& a, const BigInteger& m);
BigInteger expmbi(BigInteger n, BigInteger p, BigInteger m);
bool isnegabi(const BigInteger& n);
bool ispairbi(const BigInteger& n);
bool isprembi(const BigInteger& n);
BigInteger ndivbi(const BigInteger& n);
std::vector<BigInteger> fctpbi(const BigInteger& n);
std::string bi2string(BigInteger x, int16_t radix);
std::string bi2deci(const BigInteger& x);
std::string bi2hexa(const BigInteger& x);
int64_t bi2int(const BigInteger& a);

namespace {
    // bitsbi maximum = 30 (_elems[i] est un int32_t)
    const int32_t bitsbi = 20;
    // basebi = 2^bitsbi
    const int32_t basebi = 1 << bitsbi;
    const BigInteger binul = BigInteger(0);
    const BigInteger bione = BigInteger(1);
}

#endif  // BIGINTEGER_HPP
```