```cpp
/*
 * Implémentation des méthodes de la classe BigInt Pour simplifier,
 * mais au détriment de l'efficacité, les méthodes de division et du
 * reste de la division sont implémentées avec les méthodes d'addition
 * et de soustraction (algo des soustractions successives)
 *
 *
 * @author Vincent Granet (vg@unice.fr)
 * @version 1.0.1
 *
 *    Creation @date: 12-Dec-2016 13:32
 *  Last file update:  6-Jan-2017 16:45
 */
#include <string>
#include <cassert>
#include <iostream>
#include <ctype.h>

#include "bigint.hpp"

class Div0Exception : public std::exception {};
class BadBigIntException : public std::exception {};

// Les constructeurs

/*
 * Rôle : initialise le BigInt courant à partir d'un
 *        int
 *
 * Note : les chiffres de poids faible sont placés en tête
 */
BigInt::BigInt(int n) {
  if (n>=0) this->setSigneMoins(false);
  else {
    this->setSigneMoins(true);
    n = -n;
  }
  // extraire tous les chiffres de n
  std::string s= "";
  do {
    s += std::to_string(n % BigInt::base);
    n /= BigInt::base;
  } while (n!=0);
  this->leschiffres = s;
}

/*
 * Antécédent : s représente un entier valide
 *
 * Rôle : initialise le BigInt courant à partir d'une
 *        string qui contient les chiffres du nombre
 *
 * Note : les chiffres de poids faible sont placés en tête
 */
BigInt::BigInt(const std::string s, bool pos) {
  if (!isdigit(s[0]) && s[0]!='+' && s[0]!='-')
    throw BadBigIntException();
  //
  this->setSigneMoins(pos);
  int i=0;
  if (s[0]=='+' || s[0]=='-') {
    this->setSigneMoins(s[0]=='-');
    i=1;
  }
  else
    this->setSigneMoins(false);
```



```cpp
  // enlever tous les 0 de tête
  while (s[i]=='0' &&  i<s.length()) i++;
  //
  if (i == s.length()) { // cas particulier du 0
    this->setSigneMoins(false);
    this->leschiffres = "0";
  }
  else
    for ( ; i<s.length() ; i++) {
      if (!isdigit(s[i]))
        throw BadBigIntException();
      this->leschiffres = s[i] + this->leschiffres;
    }
}

/*
 * Rôle : renvoie le nombre de chiffres contenus dans
 *        le BigInt courant
 */
int BigInt::size() const {
  return this->leschiffres.length();
}

/*
 * Rôle : détermine le min et le max de *this et n
 *
 * Conséquent  : min = minimum(*this,n)
 *               max = maximum(*this,n)
 */
inline void BigInt::minmax(const BigInt &n, const BigInt* &min, const BigInt* &m
ax) const {
  if (*this<=n) { min=this; max=&n; }
  else { min=&n; max=this; }
}

/*
 * Antécédent : *this>=0 et n>=0
 * Rôle : renvoie *this+n
 *
 */
BigInt BigInt::addNaturel(const BigInt& n) const {
  assert(*this>=0 && n>=0);
  const BigInt *xmin, *xmax;
  this->minmax(n,xmin,xmax);
  std::string s="";
  int ret=0;
  for (int i=0; i<xmax->size(); i++) {
    // ajouter le chiffre courant de xmax + la retenue
    int c = xmax->leschiffres[i]-'0'+ret;
    // s'il existe, ajouter le chiffre courant de xmin
    if (i<xmin->size()) c+=xmin->leschiffres[i]-'0';
    // la somme des chiffres provoque-t-elle un retenue ?
    if (c<BigInt::base) ret=0;
    else { // il y a une retenue
      ret=1; c-=BigInt::base;
    }
    s = std::to_string(c) + s;
  }
  // une dernière retenue ?
  if (ret==1) s = "1"+ s;
  return BigInt(s);
}

/*
 * Antécédent : *this>=0 et n>=0
 * Rôle : renvoie *this-n
```

```
 *
 */
BigInt BigInt::subNaturel(const BigInt& n) const {
  assert(*this>=0 && n>=0);
  const BigInt *xmin, *xmax;
  bool negatif= false;
  if (*this<=n) { xmin=this; xmax=&n; negatif=true;}
  else { xmin=&n; xmax=this; }
  // faire la soustraction des chiffres
  std::string s="";
  int ret=0;
  for (int i=0; i<xmax->size(); i++) {
    // soustraire le chiffre courant de xmax + la retenue
    int c = xmax->leschiffres[i]-'0'-ret;
    // s'il existe, ajouter le chiffre courant de xmin
    if (i<xmin->size())
      c-=xmin->leschiffres[i]-'0';
    // la somme des chiffres provoque-t-elle un retenue ?
    if (c>=0) ret=0;
    else { // il y a une retenue
      ret=1; c+=BigInt::base;
    }
    s = std::to_string(c) + s;
  }
  return negatif ? -BigInt(s) : BigInt(s);
}

BigInt BigInt::multNaturel(const BigInt& n) const {
  assert(*this>=0 && n>=0);
  const BigInt *xmin, *xmax;
  this->minmax(n,xmin,xmax);
  // faire la multiplication des chiffres des chiffres
  BigInt p=0;
  int shifts=0;
  for (int i=0; i<xmin->size(); i++) {
    std::string s="";
    int ret=0;
    for (int j=0; j<xmax->size(); j++) {
      // ajouter le chiffre courant de xmax + la retenue
      int c = xmax->leschiffres[j]-'0';
      // multiplier le chiffre courant de xmax et ajouter la retenue
      c = c*(xmin->leschiffres[i]-'0')+ret;
      // la somme des chiffres provoque-t-elle un retenue ?
      if (c<BigInt::base) ret=0;
      else { // il y a une retenue
        ret=c/BigInt::base; c%=BigInt::base;
      }
      s = std::to_string(c) + s;
    }
    // une dernière retenue ?
    if (ret!=0) s =  std::to_string(ret) + s;
    // faire shifts décalages
    for (int k=0; k<shifts; k++) s = s + "0";
    shifts++;
    p = p + BigInt(s);
  }
  return p;
}

/*
 * Rôle : renvoie la somme *this+n
 */
BigInt BigInt::add(const BigInt& n) const {
  if (n==0) return *this;
  if (*this==0) return n;
  if (this->hasNotSigneMoins()) {
```



```
    if (n.hasNotSigneMoins()) return this->addNaturel(n);
  } else
    if (n.hasSigneMoins())
      return -(-*this).addNaturel(-n);
  // un est positif ou negatif et l'autre pas
  return this->hasNotSigneMoins()
    ? /* *this-n */   *this - -n
    : /* n-*this */   n - -*this;
}

/*
 * Rôle : renvoie la somme *this-n
 */
BigInt BigInt::sub(const BigInt& n) const {
  if (*this == n) return 0;
  if (this->hasNotSigneMoins()) {
    if (n.hasSigneMoins()) return *this + -n;
  }
  else // this<0
    if (n.hasNotSigneMoins()) return -(-*this+n);
  // this et n sont tous les 2 positifs ou negatifs
  return this->hasNotSigneMoins()
    ? /* *this-n   */  this->subNaturel(n)
    : /* -n -*this */  (-n).subNaturel(-*this);
}

/*
 * Rôle : renvoie le produit *this*n
 */
BigInt BigInt::mult(const BigInt& n) const {
  if (*this==0 || n==0) return 0;
  if (*this==1) return n;
  if (*this==-1) return -n;
  if (n==1)  return  *this;
  if (n==-1) return -*this;
  // faire le produit
  BigInt x = this->abs();
  BigInt p = x.multNaturel(n.abs());
  // déterminer le signe de p
  if (this->hasNotSigneMoins() && n.hasNotSigneMoins()) return p;
  return this->hasSigneMoins() && n.hasSigneMoins() ? p : -p;
}

/*
 * Rôle : renvoie le quotient *this/n
 */
BigInt BigInt::div(const BigInt& n) const {
  if (n==0)  throw Div0Exception();
  if (n==1)  return *this;
  BigInt q=0, r = this->abs();
  BigInt b = n.abs();
  while (r>=b) {
    r=r-b;
    q++;
  }
  // déterminer le signe de q
  if (this->hasNotSigneMoins() && n.hasNotSigneMoins() || q==0) return q;
  return this->hasSigneMoins() && n.hasSigneMoins() ? q : -q;
}

/*
 * Rôle : renvoie le reste de la division  *this/n
 */
BigInt BigInt::mod(const BigInt& n) const {
  return *this-n*(*this/n);
}
```

```
/*
 * Rôle : renvoie -*this
 */
BigInt BigInt::oppose() const {
  BigInt b(*this);
  b.signeMoins = !b.signeMoins;
  return b;
}

/*
 * Rôle : renvoie |*this|
 */
BigInt BigInt::abs() const {
  BigInt b(*this);
  if (b.hasSigneMoins()) b = -b;
  return b;
}

/*
 * Antécédent : n>=0 et this>=0
 * Rôle : renvoie :
 *             -1 si *this<n
 *              1 si *this>n
 *              * si *this=n
 */
int BigInt::compareNaturel(const BigInt& n) const {
  if (this->size() < n.size()) return -1;
  if (this->size() > n.size()) return 1;
  // this et n ont la même taille
  for (int i=this->size()-1 ; i >=0; i--) {
    if (this->leschiffres[i]<n.leschiffres[i]) return -1;
    if (this->leschiffres[i]>n.leschiffres[i]) return  1;
  }
  // *this et n sont égaux
  return 0;
}

/*
 * Rôle : renvoie :
 *             -1 si *this<n
 *              1 si *this>n
 *              * si *this=n
 */
int BigInt::compareTo(const BigInt& n) const {
  if (this->hasSigneMoins()) {
    if (n.hasNotSigneMoins()) return -1; // this<n
  }
  else
    if (n.hasSigneMoins()) return 1; // this>n
  // this et n sont tous les 2 positifs ou negatifs
  int cmp = this->compareNaturel(n);
  if (this->hasNotSigneMoins()) return cmp;
  // this et n sont tous les 2 negatifs
  return cmp==0 ? 0 : -cmp;
}

// Les surcharges

// comparaisons entre bigInt
bool BigInt::operator<(const BigInt &b) const {
  return this->compareTo(b)<0;
}
bool BigInt::operator<=(const BigInt &b) const {
  return this->compareTo(b)<=0;
}
```



```
bool BigInt::operator>(const BigInt &b) const {
  return this->compareTo(b)>0;
}
bool BigInt::operator>=(const BigInt &b) const {
  return this->compareTo(b)>=0;
}
bool BigInt::operator==(const BigInt &b) const {
  return this->compareTo(b)==0;
}
bool BigInt::operator!=(const BigInt &b) const {
  return this->compareTo(b)!=0;
}
// comparaisons avec un int
bool BigInt::operator<(const int n) const {
  return *this<BigInt(n);
}
bool BigInt::operator<=(const int n) const {
  return *this<=BigInt(n);
}
bool BigInt::operator>(const int n) const {
  return *this>BigInt(n);
}
bool BigInt::operator>=(const int n) const {
  return *this>=BigInt(n);
}
bool BigInt::operator==(const int n) const {
  return *this==BigInt(n);
}
bool BigInt::operator!=(const int n) const {
  return *this!=BigInt(n);
}

BigInt BigInt::operator+(const BigInt &b) const {
  return this->add(b);
}

BigInt BigInt::operator-(const BigInt &b) const {
  return this->sub(b);
}
BigInt BigInt::operator*(const BigInt &b) const {
  return this->mult(b);
}
BigInt BigInt::operator/(const BigInt &b) const {
  return this->div(b);
}

BigInt BigInt::operator%(const BigInt &b) const {
  return this->mod(b);
}

/*
 * contraitement à C, en C++ l'opérateur d'incrémentation renvoie une
 * adresse. On peut donc écrire ++a=3, mais quel est l'intérêt ?
 */
BigInt & BigInt::operator++() { // incr préfixe
  return *this = *this+1;
}

BigInt BigInt::operator++(int) { // incr postfixe
  BigInt b(*this);
  *this = *this+1;
  return b;
}

BigInt BigInt::operator-() const {
  return this->oppose();
```

```
}

/*
 * Rôle : surcharge de l'opérateur <<
 *        écrit le BigInt b sur le flot f
 */
std::ostream& operator<< (std::ostream& f, const BigInt& b)  {
  if (b.hasSigneMoins())
    f<< "-";
  // écrire les chiffres 1 à 1
  for (int i = b.size()-1; i>=0; i--)
    f<< b.leschiffres[i];
  return f;
}
```



```
/*
 * Définition de la classe BigInt qui représente des entiers de taille
 * infinie. Toutes les opérations de base de l'arithmétique entière
 * sont définies. Les chiffres sont mémorisés en base 10 dans une
 * chaîne de caractères. Un autre choix de représentation serait plus
 * judicieux et efficace :-)
 *
 * @author Vincent Granet (vg@unice.fr)
 * @version 1.0.1
 *
 *    Creation @date: 12-Dec-2016 13:32
 *  Last file update: 30-Jan-2017 19:40
 */

#ifndef BIGINT_HPP
#define BIGINT_HPP

#include <iostream>
#include <string>

class BigInt {
protected:
  static const int base = 10;
  // variables membres
  std::string leschiffres;
  bool signeMoins;
  // méthodes protégées
  bool hasNotSigneMoins() const {return !this->signeMoins;}
  bool hasSigneMoins() const {return this->signeMoins;}
  void setSigneMoins(bool b) { this->signeMoins = b; }
  int compareNaturel(const BigInt& n) const;
  BigInt addNaturel(const BigInt& n) const;
  BigInt subNaturel(const BigInt& n) const;
  BigInt multNaturel(const BigInt& n) const;
  void minmax(const BigInt &n, const BigInt* &min, const BigInt* &max) const;
public:
  // constructeurs
  BigInt() : leschiffres("0"), signeMoins(false) {};
  BigInt(int n);
  BigInt(const std::string s, const bool signeMoins=false);
  // les méthodes
  int size() const; // nombre de chiffres du BigInt courant
  BigInt add(const  BigInt& n) const;
  BigInt sub(const  BigInt& n) const;
  BigInt mult(const BigInt& n) const;
  BigInt div(const  BigInt& n) const;
  BigInt mod(const  BigInt& n) const;
  BigInt oppose() const;
  BigInt abs() const;
  //
  int compareTo(const BigInt& n) const;
  // les surcharges des opérateurs arithmétiques
  BigInt operator+(const BigInt &b) const;
  BigInt operator-(const BigInt &b) const;
  BigInt operator*(const BigInt &b) const;
  BigInt operator/(const BigInt &b) const;
  BigInt operator%(const BigInt &b) const;
  BigInt operator-() const;
  // les surcharges des opérateurs de comparaison
  bool operator<(const BigInt &b) const;
  bool operator>(const BigInt &b) const;
  bool operator<=(const BigInt &b) const;
  bool operator>=(const BigInt &b) const;
  bool operator==(const BigInt &b) const;
  bool operator!=(const BigInt &b) const;
```

```
  bool operator<(const int n) const;
  bool operator>(const int n) const;
  bool operator<=(const int n) const;
  bool operator>=(const int n) const;
  bool operator==(const int n) const;
  bool operator!=(const int n) const;

  BigInt &operator++(); // préfixe
  BigInt operator++(int);   // postfixe

  friend std::ostream &operator<<(std::ostream &f, const BigInt& b);
};

#endif
```