Правительство Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Кафедра «Компьютерной безопасности»

Отчет к лабораторной работе №7

по дисциплине

«Языки программирования»

Работу выполнил студент группы СКБ242 _____ П.В. Жучков

подпись, дата

Работу проверил _____ С.А. Булгаков

подпись, дата

Москва, 2024

Содержание

1. Постановка задачи

Доработать класс «целое произвольной длины» из Лабораторной работы №6. Для класса разработать метод `lcm()`, вычисляющий НОК своих аргументов.

Для проверки разработанного функционала подготовить набор тестов используя GoogleTest.

2. Алгоритм решения

Для нахождения НОК чисел будем увеличивать наибольшее, прибавляя к нему самого себя, до тех пор, пока оно не станет делится на наименьшее. Будем использовать ранее реализованным сложением и взятием остатка. Для реализованной функции напишем набор тестов, используя GoogleTest.

3. Реализация задачи

Для нахождения НОК изначально определим какое из чисел наибольшее, а какое наименьшее. Это нужно сделать для того, чтобы приходилось меньше складывать и делить числа. Брать будем только абсолютные значения, т.к. НОК число неотрицательное. НОК для нуля будем считать равным нулю.

```
BigInt BigInt::lcm(const BigInt& other) const{
    BigInt little, big;
    if (other.value == "0" || (*this).value == "0") return
BigInt(0);
    if(firstBigger((*this).value, other.value)){
        big = (*this).value;
        little = other.value;
    } else{
        big = other.value;
        little = (*this).value;
    }
    while (!(big % little == 0)){
        big = big + big;
    }
    return big;
}
```

4. Тестирование

Рассмотрим положительные, отрицательные и большие значения. Сами тесты показаны в приложении А.

tests/MyTests

[==========] Running 12 tests from 3 test suites.

[----------] Global test environment set-up.

[----------] 5 tests from arithmeticTests

[RUN   ] arithmeticTests.SumTest

[OK] arithmeticTests.SumTest (0 ms)

[RUN   ] arithmeticTests.DifTest

[OK] arithmeticTests.DifTest (0 ms)

[RUN   ] arithmeticTests.DivTest

[OK] arithmeticTests.DivTest (0 ms)

[RUN   ] arithmeticTests.ModTest

[OK] arithmeticTests.ModTest (0 ms)

[RUN   ] arithmeticTests.MultiplyTest

[OK] arithmeticTests.MultiplyTest (0 ms)

[----------] 5 tests from arithmeticTests (0 ms total)

[----------] 4 tests from GCDTests

[RUN   ] GCDTests.ZeroTest

[OK] GCDTests.ZeroTest (0 ms)

[RUN   ] GCDTests.PositiveTest

[OK] GCDTests.PositiveTest (0 ms)

[RUN   ] GCDTests.NegativeTest

[OK] GCDTests.NegativeTest (0 ms)

[RUN   ] GCDTests.BigNumberTest

[OK] GCDTests.BigNumberTest (0 ms)

[----------] 4 tests from GCDTests (0 ms total)

```
[----------] 3 tests from LCMTests
[ RUN      ] LCMTests.PositiveTest
[       OK ] LCMTests.PositiveTest (0 ms)
[ RUN      ] LCMTests.NegativeTest
[       OK ] LCMTests.NegativeTest (0 ms)
[ RUN      ] LCMTests.BigNumberTest
[       OK ] LCMTests.BigNumberTest (0 ms)
[----------] 3 tests from LCMTests (0 ms total)

[----------] Global test environment tear-down
[==========] 12 tests from 3 test suites ran. (1 ms total)
[  PASSED  ] 12 tests.
```

Приложение А (tests.cpp)

```
#include <iostream>
#include <string>
#include "../class.h"

#include <gtest/gtest.h>

// ArithmeticTests ArithmeticTests

TEST(arithmeticTests, SumTest) {
  BigInt a, b;

  a = "1003"; b = "410";
  EXPECT_EQ(a + b, (1003 + 410));
  EXPECT_EQ(b + a, (410 + 1003));
  a = "84736"; b = "1234";
  EXPECT_EQ(a + b, (84736 + 1234));
  EXPECT_EQ(b + a, (1234 + 84736));
  a = "150000000000000"; b = "1234";
  EXPECT_EQ(a + b, ("150000000001234"));
  EXPECT_EQ(b + a, ("150000000001234"));
}
TEST(arithmeticTests, DifTest) {
  BigInt a, b;

  a = "1003"; b = "410";
  EXPECT_EQ(a - b, (1003 - 410));
  EXPECT_EQ(b - a, (410 - 1003));
  a = "84736"; b = "1234";
  EXPECT_EQ(a - b, (84736 - 1234));
  EXPECT_EQ(b - a, (1234 - 84736));
  a = "150000000000000"; b = "1234";
  EXPECT_EQ(a - b, ("149999999998766"));
  EXPECT_EQ(b - a, ("-149999999998766"));
}
TEST(arithmeticTests, DivTest) {
  BigInt a, b;

  a = "1003"; b = "410";
  EXPECT_EQ(a / b, (1003 / 410));
  EXPECT_EQ(-a / b, (-1003 / 410));
  EXPECT_EQ(-a / -b, (-1003 / -410));
  EXPECT_EQ(a / -b, (1003 / -410));
  EXPECT_EQ(b / a, (410 / 1003));
  EXPECT_EQ(-b / a, (-410 / 1003));
```

```
  EXPECT_EQ(-b / -a, (-410 / -1003));
  EXPECT_EQ(b / -a, (410 / -1003));
  a = "84736"; b = "1234";
  EXPECT_EQ(a / b, (84736 / 1234));
  EXPECT_EQ(-a / b, (-84736 / 1234));
  EXPECT_EQ(-a / -b, (-84736 / -1234));
  EXPECT_EQ(a / -b, (84736 / -1234));
  EXPECT_EQ(b / a, (1234 / 84736));
  EXPECT_EQ(-b / a, (-1234 / 84736));
  EXPECT_EQ(-b / -a, (-1234 / -84736));
  EXPECT_EQ(b / -a, (1234 / -84736));
  a = "100000000000000000000", b = "3";
  EXPECT_EQ(a / b, ("33333333333333333333"));
  EXPECT_EQ(-a / b, ("-33333333333333333333"));
  EXPECT_EQ(-a / -b, ("33333333333333333333"));
  EXPECT_EQ(a / -b, ("-33333333333333333333"));
  EXPECT_EQ(b / a, (0));
  EXPECT_EQ(-b / a, (0));
  EXPECT_EQ(-b / -a, (0));
  EXPECT_EQ(b / -a, (0));
}
TEST(arithmeticTests, ModTest) {
  BigInt a, b;
  a = "1003"; b = "410";
  EXPECT_EQ(a % b, (1003 % 410));
  EXPECT_EQ(-a % b, (-1003 % 410));
  EXPECT_EQ(-a % -b, (-1003 % -410));
  EXPECT_EQ(a % -b, (1003 % -410));
  EXPECT_EQ(b % a, (410 % 1003));
  EXPECT_EQ(-b % a, (-410 % 1003));
  EXPECT_EQ(-b % -a, (-410 % -1003));
  EXPECT_EQ(b % -a, (410 % -1003));
  a = "84736"; b = "1234";
  EXPECT_EQ(a % b, (84736 % 1234));
  EXPECT_EQ(-a % b, (-84736 % 1234));
  EXPECT_EQ(-a % -b, (-84736 % -1234));
  EXPECT_EQ(a % -b, (84736 % -1234));
  EXPECT_EQ(b % a, (1234 % 84736));
  EXPECT_EQ(-b % a, (-1234 % 84736));
  EXPECT_EQ(-b % -a, (-1234 % -84736));
  EXPECT_EQ(b % -a, (1234 % -84736));
  a = "100000000000000000000", b = "3";
  EXPECT_EQ(a % b, (1));
  EXPECT_EQ(-a % b, (-1));
  EXPECT_EQ(-a % -b, (-1));
```

```
  EXPECT_EQ(a % -b, (1));
  EXPECT_EQ(b % a, (3));
  EXPECT_EQ(-b % a, (-3));
  EXPECT_EQ(-b % -a, (-3));
  EXPECT_EQ(b % -a, (3));
}
TEST(arithmeticTests, MultiplyTest) {
  BigInt a, b;
  a = "1003"; b = "410";
  EXPECT_EQ(a * b, (1003 * 410));
  EXPECT_EQ(-a * b, (-1003 * 410));
  EXPECT_EQ(-a * -b, (-1003 * -410));
  EXPECT_EQ(a * -b, (1003 * -410));
  EXPECT_EQ(b * a, (410 * 1003));
  EXPECT_EQ(-b * a, (-410 * 1003));
  EXPECT_EQ(-b * -a, (-410 * -1003));
  EXPECT_EQ(b * -a, (410 * -1003));
  a = "84736"; b = "1234";
  EXPECT_EQ(a * b, (84736 * 1234));
  EXPECT_EQ(-a * b, (-84736 * 1234));
  EXPECT_EQ(-a * -b, (-84736 * -1234));
  EXPECT_EQ(a * -b, (84736 * -1234));
  EXPECT_EQ(b * a, (1234 * 84736));
  EXPECT_EQ(-b * a, (-1234 * 84736));
  EXPECT_EQ(-b * -a, (-1234 * -84736));
  EXPECT_EQ(b * -a, (1234 * -84736));
  a = "100000000000000000000", b = "3";
  EXPECT_EQ(a * b, ("300000000000000000000"));
  EXPECT_EQ(-a * b, ("-300000000000000000000"));
  EXPECT_EQ(-a * -b, ("300000000000000000000"));
  EXPECT_EQ(a * -b, ("-300000000000000000000"));
  EXPECT_EQ(b * a, ("300000000000000000000"));
  EXPECT_EQ(-b * a, ("-300000000000000000000"));
  EXPECT_EQ(-b * -a, ("300000000000000000000"));
  EXPECT_EQ(b * -a, ("-300000000000000000000"));

}

// GCD GCD GCDG CDGCGDCGDGCGDCDGCDGDGD
TEST(GCDTests, ZeroTest){
  BigInt a = 0;
  EXPECT_EQ(a.gcd(12), 12);
  EXPECT_EQ(a.gcd(89), 89);
  EXPECT_EQ(a.gcd(0), 0);
}
TEST(GCDTests, PositiveTest){
```

```
  BigInt a = 16;
  EXPECT_EQ(a.gcd(9), 1);
  EXPECT_EQ(a.gcd(16), 16);
  EXPECT_EQ(a.gcd(32), 16);
  EXPECT_EQ(a.gcd(127), 1);
  EXPECT_EQ(a.gcd(8), 8);
}
TEST(GCDTests, NegativeTest){
  BigInt a = 16;
  EXPECT_EQ(a.gcd(-9), 1);
  EXPECT_EQ(a.gcd(-16), 16);
  EXPECT_EQ(a.gcd(-32), 16);
  EXPECT_EQ(a.gcd(-127), 1);
  EXPECT_EQ(a.gcd(-8), 8);
  EXPECT_EQ((-a).gcd(-8), 8);
  EXPECT_EQ((-a).gcd(-9), 1);
  EXPECT_EQ((-a).gcd(-32), 16);
  EXPECT_EQ((-a).gcd(-127), 1);
  EXPECT_EQ((-a).gcd(8), 8);
}
TEST(GCDTests, BigNumberTest){
  BigInt a = 42557659200; //6 * 12 * 36 * 14 * 19 * 5 *
12345
  EXPECT_EQ(a.gcd(12345), 12345);
  EXPECT_EQ(a.gcd(6 * 12), 6 * 12);
  EXPECT_EQ(a.gcd(127), 1);
  EXPECT_EQ(a.gcd(12345 * 12), 12345 * 12);
}

//LCMLCMLCMLCMLCM
TEST(LCMTests, PositiveTest){
  BigInt a = 8;
  EXPECT_EQ(a.lcm(8), 8);
  EXPECT_EQ(a.lcm(12), 24);
  EXPECT_EQ(a.lcm(4), 8);
  EXPECT_EQ(a.lcm(10), 40);
}
TEST(LCMTests, NegativeTest){
  BigInt a = 8;
  EXPECT_EQ(a.lcm(-8), 8);
  EXPECT_EQ(a.lcm(-12), 24);
  EXPECT_EQ((-a).lcm(8), 8);
  EXPECT_EQ((-a).lcm(12), 24);
  EXPECT_EQ((-a).lcm(-8), 8);
  EXPECT_EQ((-a).lcm(-12), 24);
```

```
}
TEST(LCMTests, BigNumberTest){
  BigInt a = "87126341";
  EXPECT_EQ(a.lcm(BigInt("362")), BigInt("174252682"));
}

int main(int argc, char **argv) {

  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Приложение Б (class.cpp)

```
#include "class.h"
#include <string>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <stdexcept>
#include <cstdlib>
#include <sstream>
#include <utility>

//constructors
BigInt::BigInt() {value = "0"; sign =
true;}           //defolt constructor
BigInt::BigInt(bool insign, std::string str) {     //bool +
string

    if (str[0] == '0' && str.length() != 1){
        std::cerr << "ERROR: use numbers without leading
zeros\n";
        exit(EXIT_FAILURE);
    }
    for (int i = 0; int(i < int(str.length())); ++i){
        if (str[i] > '9' || str[i] < '0'){
            std::cout << "ERROR: use only numeral values for
BigInt\n";
            exit(EXIT_FAILURE);
        }
    }
    sign = insign;
    value = str;
    if (value == "0") sign = true;
}
BigInt::BigInt(std::string str) {                  //string
constructor

    if (str[0] == '-'){
        if (str.length() == 1 || (str.length() == 2 &&
str[1] == '0')){
            std::cerr << "ERROR: wtf use NUMBERS\n";
            exit(EXIT_FAILURE);
        }
        if (str[1] == '0'){
            std::cerr << "ERROR: use numbers without leading
zeros\n";
```

```
            exit(EXIT_FAILURE);
        }
    } else {
        if (str[0] == '0' && str.length() != 1){
            std::cerr << "ERROR: use numbers without leading
zeros\n";
            exit(EXIT_FAILURE);
        }
    }
    for (int i = 0; i < int(str.length()); ++i){
        if ((str[i] > '9' || str[i] < '0')){
            if (i != 0 || str[0] != '-'){
                std::cerr << "ERROR: use only integer values
for BigInt\n";
                exit(EXIT_FAILURE);
            }
        }
    }
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);


}
BigInt::BigInt(const char* str)
{                   //cstring constructor

    if (str[0] == '-'){
        if (strlen(str) == 1 || (strlen(str) == 2 && str[1]
== '0')){
            std::cerr << "ERROR: wtf use NUMBERS\n";
            exit(EXIT_FAILURE);
        }
        if (str[1] == '0'){
            std::cerr << "ERROR: use numbers without leading
zeros\n";
            exit(EXIT_FAILURE);
        }
    } else {
        if (str[0] == '0' && strlen(str) != 1){
            std::cerr << "ERROR: use numbers without leading
zeros\n";
            exit(EXIT_FAILURE);
        }
    }
    for (int i = 0; i < int(strlen(str)); ++i){
```

```
        if (str[i] > '9' || str[i] < '0'){
            if (i != 0 || str[0] != '-'){
                std::cout << "ERROR: use only numeral values
for BigInt\n";
                exit(EXIT_FAILURE);
            }
        }
    }
    sign = str[0] == '-' ? false : true;
    value = sign ? std::string(str) : std::string(str + 1);
}
BigInt::BigInt(int num){                                //int
constructor
    std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);

}
BigInt::BigInt(short num){                              //short
constructor
     std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(long
num){                               //long  constructor
    std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(char
num){                               //char  constructor
     std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
```

```
BigInt::BigInt(BigInt const &obj){                    //copy
constructor
    value = obj.value;
    sign = obj.sign;
}
BigInt::~BigInt(){}                                   //destru
ctour

//dop functions
std::string stringDif(const std::string& big, const
std::string& small){     //substract abs values
    std::string result;
    int b = big.size() - 1; int s = small.size() - 1;
    int dop = 0;
    while (b >= 0 || s >= 0){
        int dif, right = (s >= 0 ? small[s--] - '0' : 0),
left = big[b--] - '0';
        dif = left - right - dop;
        if (dif < 0){
            dif += 10;
            dop = 1;
        } else {
            dop = 0;
        }
        result.push_back(dif + '0');
    }
    while(result.size() > 1 && result[result.size() - 1] ==
'0') {
            result.erase(result.size() - 1, 1);
    }

    std::reverse(result.begin(), result.end());
    return result.empty() ? "0" : result;
}
std::string stringSum(const std::string& adin, const
std::string& dva){     //sumarize abs values
    std::string result;
    int i = adin.size() - 1, j = dva.size() - 1;
    int dop = 0;
    while (i >= 0 || j >= 0 || dop == 1){
        int sum = 0;
        sum = (i >= 0 ? adin[i--] - '0' : 0) + (j >= 0 ?
dva[j--] - '0' : 0) + dop;
        dop = sum / 10;
        result.push_back((sum % 10) + '0');
```

```
    }
    std::reverse(result.begin(), result.end());
    return result;
}
std::string stringMultiply(const std::string& adin, const
std::string& dva){    //multiply
    int a, d = dva.size() - 1;
    int dop = 0;
    std::string result = "0", tmp;
    while (d >= 0){
        tmp = "";
        a = adin.size() - 1;
        for (int i = 0; i < (int)dva.size() - d - 1; ++i){
            tmp.push_back('0');
        }
        while(a >= 0 || dop){
            int prod;
            prod = (a >= 0 ? adin[a--] - '0' : 0) * (dva[d]
- '0') + dop;
            dop = prod / 10;
            tmp.push_back((prod % 10) + '0');
        }
        d--;
        std::reverse(tmp.begin(), tmp.end());
        //std::cout << tmp << ' ' << result << "\n";
        result = stringSum(result, tmp);
    }
    while (result.length() > 1 && result[0] == '0') {
            result.erase(0, 1);
    }
    return result;
}
bool firstBigger(const std::string& adin, const std::string&
dva){     //compare abs values

    if(adin.length() > dva.length()) return true;
    if(adin.length() < dva.length()) return false;
    for (int i = 0; i < int(adin.length()); ++i){
        if (adin[i] > dva[i]) return true;
        if (adin[i] < dva[i]) return false;
    }
    return true;
}
std::pair<std::string, std::string> stringDivision(const
std::string& dividend, const std::string& divisor){
```

```
    std::pair<std::string, std::string> ans;
    const size_t divsize = divisor.length();
    std::string dim = dividend.substr(0, divsize);
    std::string quotient = "", rem;
    size_t p = divsize;
    std::string sub[11];
    sub[1] = divisor;

    if(divisor == "0"){
        std::cerr << "ERROR: Division by zero\n";
        exit(EXIT_FAILURE);
    }
    if (divisor == dividend) return std::make_pair("1",
"0");
    if (firstBigger(dividend, divisor) == false) return
std::make_pair("0", dividend);

    while(p != dividend.length() || divsize ==
dividend.length()){

        bool addZero = false;
        while(firstBigger(sub[1], dim) && sub[1] != dim){
            if (addZero) quotient += '0';
            if (p != dividend.length()){
                addZero = true;
                if(dim != "0") dim += dividend[p];
                else dim = (dividend[p] == '0' ? "" : dim =
dividend[p]);
                p++;
            } else {
                return std::make_pair(quotient, (dim == "" ?
"0" : dim));
            }
        }

        for (int i = 2; i <= 10; i++){
            if (sub[i] == "") sub[i] = stringSum(sub[i-1],
divisor);
            if (firstBigger(sub[i], dim)){
                rem = stringDif(dim, (sub[i] == dim ? sub[i]
: sub[--i]));
                rem = (rem == "0" ? "" : rem);
                quotient += (i + '0');
                break;
            }
```

```
        }
        dim = rem;
    }
    return std::make_pair(quotient, (rem == "" ? "0" :
rem));
}

//methods

const std::string& BigInt::get_value() const{
    return value;
}
void BigInt::set_value(const std::string& str) {
    value = str;
}
const bool BigInt::get_sign() const{
    return sign;
}
void BigInt::set_sign(const bool s) {
    sign = s;
}
BigInt BigInt::gcd(const BigInt& other) const{
    BigInt dividend, divisor, remainder;
    if ((*this).value == "0") return other;
    if (other.value == "0") return *this;
    if (firstBigger((*this).value, other.value)){
        dividend = (*this).value;
        divisor = other.value;
    } else {
        dividend = other.value;
        divisor = (*this).value;
    }
    remainder = dividend % divisor;
    while (!(remainder == BigInt(0))){
        dividend = divisor;
        divisor = remainder;
        remainder = dividend % divisor;
    }
    return divisor;
}
BigInt BigInt::lcm(const BigInt& other) const{
    BigInt little, big;
    if (other.value == "0" || (*this).value == "0") return
BigInt(0);
    if(firstBigger((*this).value, other.value)){
```

```
        big = (*this).value;
        little = other.value;
    } else{
        big = other.value;
        little = (*this).value;
    }
    while (!(big % little == 0)){
        big = big + big;
    }
    return big;
}

//operators

BigInt BigInt::operator=(const BigInt& other) {
    if (this != &other) {
        value = other.value;
        sign = other.sign;
    }
    return *this;
}
BigInt BigInt::operator=( short num){
    *this = BigInt(num);
    return *this;
}
BigInt BigInt::operator=( char num){
    *this = BigInt(num);
    return *this;
}
BigInt BigInt::operator=( int num){
    *this = BigInt(num);
    return *this;
}
BigInt BigInt::operator=( long num){
    *this = BigInt(num);
    return *this;
}

std::ostream& operator<<(std::ostream& out, const BigInt&
num) {
    if (num.sign == false) out << '-';
    out << num.value;
    return out;
}
std::istream& operator>>(std::istream& in, BigInt& num){
```

```
    std::string input;
    in >> input;
    num = BigInt(input);
    return in;
}

BigInt operator+(const BigInt& left, const BigInt& right) {
    if (left.sign == right.sign) {
        return BigInt(left.sign, stringSum(left.value,
right.value));
    }
    if (left.sign){
        if(firstBigger(left.value, right.value)){
            return BigInt(true, stringDif(left.value,
right.value));
        }
        return BigInt(false, stringDif(right.value,
left.value));
    } else {
        if(firstBigger(right.value, left.value)){
            return BigInt(true, stringDif(right.value,
left.value));
        }
        return BigInt(false, stringDif(left.value,
right.value));
    }
}
BigInt operator-(const BigInt& left, const BigInt& right){
    return left + BigInt(!right.sign, right.value);
}
BigInt operator*(const BigInt& left, const BigInt& right) {
    if (left.sign == right.sign)
        return BigInt(true, stringMultiply(left.value,
right.value));
    else
        return BigInt(false, stringMultiply(left.value,
right.value));
}
BigInt operator/(const BigInt& left, const BigInt& right) {
        if (left.sign == right.sign)
        return BigInt(true, stringDivision(left.value,
right.value).first);
    else
        return BigInt(false, stringDivision(left.value,
right.value).first);
```

```
}
BigInt operator%(const BigInt& left, const BigInt& right) {
        if (left.sign)
        return BigInt(true, stringDivision(left.value,
right.value).second);
    else
        return BigInt(false, stringDivision(left.value,
right.value).second);
}
bool operator==(const BigInt& left, const BigInt& right){
    return (left.value == right.value && left.sign ==
right.sign);
}
BigInt BigInt::operator-(){                          //unary
minus
    BigInt a;
    a.value = value;
    a.sign = !sign;
    return a;
}

//+++++++++++++
BigInt operator+(int left, const BigInt& right)  {return
BigInt(left) + right;}
BigInt operator+(std::string left, const BigInt& right)
{return BigInt(left) + right;}
BigInt operator+(short left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(long left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(char left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(const BigInt& left, int right) {return left
+ BigInt(right);}
BigInt operator+(const BigInt& left, std::string
right){return left + BigInt(right);}
BigInt operator+(const BigInt& left, long right){return left
+ BigInt(right);}
BigInt operator+(const BigInt& left, short right){return
left + BigInt(right);}
BigInt operator+(const BigInt& left, char right){return left
+ BigInt(right);}

//-------------
```

```
BigInt operator-(int left, const BigInt& right)  {return
BigInt(left) - right;}
BigInt operator-(std::string left, const BigInt& right)
{return BigInt(left) - right;}
BigInt operator-(short left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(long left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(char left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(const BigInt& left, int right) {return left
- BigInt(right);}
BigInt operator-(const BigInt& left, std::string
right){return left - BigInt(right);}
BigInt operator-(const BigInt& left, long right){return left
- BigInt(right);}
BigInt operator-(const BigInt& left, short right){return
left - BigInt(right);}
BigInt operator-(const BigInt& left, char right){return left
- BigInt(right);}

//*****************
BigInt operator*(int left, const BigInt& right)  {return
BigInt(left) * right;}
BigInt operator*(std::string left, const BigInt& right)
{return BigInt(left) * right;}
BigInt operator*(short left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(long left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(char left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(const BigInt& left, int right) {return left
* BigInt(right);}
BigInt operator*(const BigInt& left, std::string
right){return left * BigInt(right);}
BigInt operator*(const BigInt& left, long right){return left
* BigInt(right);}
BigInt operator*(const BigInt& left, short right){return
left * BigInt(right);}
BigInt operator*(const BigInt& left, char right){return left
* BigInt(right);}

//////////////////
```

```
BigInt operator/(int left, const BigInt& right)  {return
BigInt(left) / right;}
BigInt operator/(std::string left, const BigInt& right)
{return BigInt(left) / right;}
BigInt operator/(short left, const BigInt& right) {return
BigInt(left) / right;}
BigInt operator/(long left, const BigInt& right) {return
BigInt(left) / right;}
BigInt operator/(char left, const BigInt& right) {return
BigInt(left) / right;}
BigInt operator/(const BigInt& left, int right) {return left
/ BigInt(right);}
BigInt operator/(const BigInt& left, std::string
right){return left / BigInt(right);}
BigInt operator/(const BigInt& left, long right){return left
/ BigInt(right);}
BigInt operator/(const BigInt& left, short right){return
left / BigInt(right);}
BigInt operator/(const BigInt& left, char right){return left
/ BigInt(right);}

//%%%%%%%%%%%%%%
BigInt operator%(int left, const BigInt& right)  {return
BigInt(left) % right;}
BigInt operator%(std::string left, const BigInt& right)
{return BigInt(left) % right;}
BigInt operator%(short left, const BigInt& right) {return
BigInt(left) % right;}
BigInt operator%(long left, const BigInt& right) {return
BigInt(left) % right;}
BigInt operator%(char left, const BigInt& right) {return
BigInt(left) % right;}
BigInt operator%(const BigInt& left, int right) {return left
% BigInt(right);}
BigInt operator%(const BigInt& left, std::string
right){return left % BigInt(right);}
BigInt operator%(const BigInt& left, long right){return left
% BigInt(right);}
BigInt operator%(const BigInt& left, short right){return
left % BigInt(right);}
BigInt operator%(const BigInt& left, char right){return left
% BigInt(right);}

//== == == == == == == ==
```

```
bool operator==(int left, const BigInt& right)  {return
BigInt(left) == right;}
bool operator==(std::string left, const BigInt& right)
{return BigInt(left) == right;}
bool operator==(const char* left, const BigInt& right)
{return BigInt(left) == right;}
bool operator==(short left, const BigInt& right) {return
BigInt(left) == right;}
bool operator==(long left, const BigInt& right) {return
BigInt(left) == right;}
bool operator==(char left, const BigInt& right) {return
BigInt(left) == right;}
bool operator==(const BigInt& left, int right) {return left
== BigInt(right);}
bool operator==(const BigInt& left, std::string
right){return left == BigInt(right);}
bool operator==(const BigInt& left, const char*
right){return left == BigInt(right);}
bool operator==(const BigInt& left, long right){return left
== BigInt(right);}
bool operator==(const BigInt& left, short right){return left
== BigInt(right);}
bool operator==(const BigInt& left, char right){return left
== BigInt(right);}

//to other types
BigInt::operator std::string() const {
    return (sign ? "" : "-") + value;
}
BigInt::operator int() const {
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator short() const {
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator char() const {
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator long() const {
    return (sign ? 1 : -1) * std::atol(value.c_str());
}
```

Приложение В (class.h)

```
#ifndef CLASS_H
#define CLASS_H

#include <string>
#include <iostream>
#include <algorithm>

class BigInt{
    private:
        std::string value;
        bool sign;

    public:
    //constructors
        BigInt();
        ~BigInt();
        BigInt(bool sign, std::string str);
        BigInt(std::string str);
        BigInt(const char* str);
        BigInt(int num);
        BigInt(short num);
        BigInt(long num);
        BigInt(char num);
        BigInt(const BigInt& obj);



    //methods

        const std::string& get_value() const;
        void set_value(const std::string& str);
        const bool get_sign() const;
        void set_sign(const bool s);
        BigInt gcd(const BigInt& other) const;
        BigInt lcm(const BigInt& other) const;

    //operators
        friend BigInt operator+(const BigInt& left, const
BigInt& right);
        friend BigInt operator-(const BigInt& left, const
BigInt& right);
        friend BigInt operator*(const BigInt& left, const
BigInt& right);
```

```
        friend BigInt operator/(const BigInt& left, const
BigInt& right);
        friend BigInt operator%(const BigInt& left, const
BigInt& right);
        friend bool operator==(const BigInt& left, const
BigInt& right);
        friend std::ostream& operator<<(std::ostream& out,
const BigInt& num);
        friend std::istream& operator>>(std::istream& in,
BigInt& num);

        BigInt operator-(); //unary minus

        BigInt operator=(const BigInt& other);
        BigInt operator=( short num);
        BigInt operator=( char num);
        BigInt operator=( int num);
        BigInt operator=( long num);

        operator std::string() const;
        operator int() const;
        operator short() const;
        operator long() const;
        operator char() const;
};
//++++++++++++++++++++
BigInt operator+(int left, const BigInt& right);
BigInt operator+(std::string left, const BigInt& right);
BigInt operator+(short left, const BigInt& right);
BigInt operator+(long left, const BigInt& right);
BigInt operator+(char left, const BigInt& right);
BigInt operator+(const BigInt& left, int right);
BigInt operator+(const BigInt& left, std::string right);
BigInt operator+(const BigInt& left, long right);
BigInt operator+(const BigInt& left, short right);
BigInt operator+(const BigInt& left, char right);

//--------------------
BigInt operator-(int left, const BigInt& right);
BigInt operator-(std::string left, const BigInt& right);
BigInt operator-(short left, const BigInt& right);
BigInt operator-(long left, const BigInt& right);
BigInt operator-(char left, const BigInt& right);
BigInt operator-(const BigInt& left, int right);
BigInt operator-(const BigInt& left, std::string right);
```

```
BigInt operator-(const BigInt& left, long right);
BigInt operator-(const BigInt& left, short right);
BigInt operator-(const BigInt& left, char right);

//************/
BigInt operator*(const BigInt& left, int right);
BigInt operator*(const BigInt& left, std::string right);
BigInt operator*(const BigInt& left, short right);
BigInt operator*(const BigInt& left, long right);
BigInt operator*(const BigInt& left, char right);
BigInt operator*(char left, const BigInt& right);
BigInt operator*(long left, const BigInt& right);
BigInt operator*(short left, const BigInt& right);
BigInt operator*(std::string left, const BigInt& right);
BigInt operator*(int left, const BigInt& right);

////////////
BigInt operator/(const BigInt& left, int right);
BigInt operator/(const BigInt& left, std::string right);
BigInt operator/(const BigInt& left, short right);
BigInt operator/(const BigInt& left, long right);
BigInt operator/(const BigInt& left, char right);
BigInt operator/(char left, const BigInt& right);
BigInt operator/(long left, const BigInt& right);
BigInt operator/(short left, const BigInt& right);
BigInt operator/(std::string left, const BigInt& right);
BigInt operator/(int left, const BigInt& right);

//%%%%%%%%%%%%
BigInt operator%(const BigInt& left, int right);
BigInt operator%(const BigInt& left, std::string right);
BigInt operator%(const BigInt& left, short right);
BigInt operator%(const BigInt& left, long right);
BigInt operator%(const BigInt& left, char right);
BigInt operator%(char left, const BigInt& right);
BigInt operator%(long left, const BigInt& right);
BigInt operator%(short left, const BigInt& right);
BigInt operator%(std::string left, const BigInt& right);
BigInt operator%(int left, const BigInt& right);

//== == == ==
bool operator==(const BigInt& left, int right);
bool operator==(const BigInt& left, std::string right);
bool operator==(const BigInt& left, const char* right);
bool operator==(const BigInt& left, short right);
```

```
bool operator==(const BigInt& left, long right);
bool operator==(const BigInt& left, char right);
bool operator==(char left, const BigInt& right);
bool operator==(long left, const BigInt& right);
bool operator==(short left, const BigInt& right);
bool operator==(std::string left, const BigInt& right);
bool operator==(const char* left, const BigInt& right);
bool operator==(int left, const BigInt& right);

std::string stringDif(const std::string& big, const
std::string& small);
std::string stringSum(const std::string& adin, const
std::string& dva);
std::string stringMultiply(const std::string& adin, const
std::string& dva);
std::pair<std::string, std::string> stringDivision(const
std::string& dividend, const std::string& divisor);
bool firstBigger(const std::string& adin, const std::string&
dva);

#endif
```

Приложение Г

BigInt
-value: std::string -sign: bool
+BigInt(); +~BigInt(); +BigInt(bool sign, std::string str); +BigInt(std::string str); +BigInt(const char* str); +BigInt(int num); +BigInt(short num); +BigInt(long num); +BigInt(char num); +BigInt(const BigInt& obj); +const std::string& get_value() const; +void set_value(const std::string& str); +const bool get_sign() const; +void set_sign(const bool s); +BigInt gcd(const BigInt& other) const; +BigInt lcm(const BigInt& other) const; +const std::string& get_value() const; +void set_value(const std::string& str); +const bool get_sign() const; +void set_sign(const bool s); +BigInt gcd(const BigInt& other) const; +BigInt lcm(const BigInt& other) const; +friend BigInt operator+(const BigInt& left, const BigInt& right); +friend BigInt operator-(const BigInt& left, const BigInt& right); +friend BigInt operator*(const BigInt& left, const BigInt& right); +friend BigInt operator/(const BigInt& left, const BigInt& right); +friend BigInt operator%(const BigInt& left, const BigInt& right); +friend bool operator==(const BigInt& left, const BigInt& right); +friend std::ostream& operator<<(std::ostream& out, const BigInt& num); +friend std::istream& operator>>(std::istream& in, BigInt& num); +BigInt operator-(); +operator std::string() const; +operator int() const; +operator short() const; +operator long() const; +operator char() const;