Лабораторная работа № 6 по курсу дискретного анализа: Калькулятор

Выполнил студент группы М8О-308Б-20 *Ядров Артем*.

Условие

Необходимо разработать программную библиотеку на языке C или $C++$, реализующую простейшие арифметические действия и проверку условий над целыми неотрицательными числами. На основании этой библиотеки, нужно составить программу, выполняющую вычисления над парами десятичных чисел и выводящую результат на стандартный файл вывода.

Список арифметических операций:

- Сложение (+).
- Вычитание (-).
- Умножение (*).
- Возведение в степень ($\hat{)}$.
- Деление (/).

В случае возникновения переполнения в результате вычислений, попытки вычесть из меньшего числа большее, деления на ноль или возведении нуля в нулевую степень, программа должна вывести на экран строку $Error$.

Список условий:

- Больше (>).
- Меньше (<).
- Равно (=).

  В случае выполнения условия, программа должна вывести на экран строку $true$, в противном случае — $false$.

Метод решения

Структура числа

Для того, чтобы хранить большие числа я решил использовать другую систему счисления: основание системы должно быть достаточно большим, но при этом необходима замкнутость относительно операции умножения "цифр"системы (при умножении во избежании переполнения типов результат должен оставаться в типе int). Удобнее всего было бы взять степень двойки для битовых операций, однако для удобства при считывании я принял решения в качестве основания взять число 10^4.

Длинное число хранится в виде вектора его цифр в порядке убывания разрядов (последний разряд хранится в начале, первый - в конце). Таким образом добавление разряда в начало числа не требует сдвига всего вектора, который можно выполнить за $O(n)$. Также несмотря на то, что в задаче мы оперируем с целыми неотрицательными числами, я добавил представление отрицательных чисел: для этого существует флаг, который отвечает за знак числа.

Простые операции

Конструкторы класса $BigInt$ слишком примитивны, чтобы их описывать. Для реализации сравнения достаточно было реализовать две операции: я выбрал операцию "меньше"и "равно". Операция равенства достаточно проста и требует проверки вектора и флага отрицательного числа на равенство, в то время как в операции "меньше"необходимо учесть случаи, когда числа имеют разные знаки.

Далее идет реализация операций сложения и вычитания. Они выполняются известным школьным методом: в столбик. Необходимо учитывать перенос разрядов. В общем вся операция выполняется за $O(n)$, где n - максимальное количество разрядов среди двух чисел.

Умножение

Операцию умножения можно реализовать 3 способами: школьное умножение "столбиком"($O(n^3)$), метод Карацубы ($O(n^{\log_2 3})$) и с помощью дискретного преобразования Фурье($O(n \log n)$). Я выбрал третий способ, так как он самый быстрый и уже знакомый. Итак, преобразование Фурье. Известно, что любое n-значное число в p-ричной системе счисления можно записать в виде многочлена следующего вида:

$$A(p) = a_0p^0 + a_1p^1 + a_2p^2 + \ldots + a_{n_1}p^{n-1} = \sum_{k=0}^{n-1} a_kp^k$$

То есть каждому числу соответствует единственный многочлен и каждому многочлену соответствует единственное число. Но многочлены с многочленами можно оперировать в любом поле, результат не изменится.

Любой многочлен можно представить в виде набора коэффициентов, при этом умножение в таком виде будет происходить за время $O(n^2)$, либо в виде n различных точек, при этом умножение будет происходить за $O(n)$ путем умножения соответствующих точек. Дискретным преобразованием Фурье называется переход от представления в виде коэффициентов к представлению в виде набора точек. Обратным дискретным преобразованием Фурье называется переход от представления в виде набора точек к представлению в виде набора коэффициентов. Если существует алгоритм прямого и обратного преобразования, который эффективнее, чем $O(n^2)$, то можно перемножать многочлены путем прямого преобразования, перемножения точек и обратного преобразования за время, меньшее чем при наивном алгоритме. Оказывается, существует алгоритм быстрого преобразования Фурье (БПФ), названный в честь Кули и Таки. Его временная оценка $O(n \log n)$. Основывается алгоритм на хорошо известном методе "разделяй и властвуй". Основная идея состоит в выборе точек: в качестве точек берутся корни n-ой степени из единицы в комплексном поле. Известно, что все корни являются степенями числа $e^{\frac{2\pi i}{n}}$.

Теперь приступим к преобразованию. Пусть имеется многочлен $A(x)$ степени $n = 2^k, (k > 0)$. В любом случае многочлен можно дополнить до степени двойки нулевыми коэффициентами.

1. Разделим многочлен на два: один - с четными коэффициентами, другой - с нечетными.

   $A_0(x) = a_0x^0 + a_2x^1 + a_4x^2 + \ldots + a_{n/2}x^{n/2-1}$

   $A_1(x) = a_1x^0 + a_3x^1 + a_5x^2 + \ldots + a_{n/2}x^{n/2-1}$

   При этом $A(x) = A_0(x^2) + xA_1(x^2)$.

2. Рекурсивно преобразуем многочлены A_0 и A_1.

3. Пусть мы получили набор $\{y_k^0\}_{k=0}^{n/2-1}$ - преобразование многочлена A_0 и набор $\{y_k^1\}_{k=0}^{n/2-1}$ - преобразование многочлена A_1. Восстановим точки y_k исходного многочлена.

$$y_k = y_k^0 + \omega_n^k y_k^1 \qquad \forall k = 0, 1, \ldots, n/2 - 1$$

$$y_{k+n/2} = A(\omega_n^{k+n/2}) = A_0(\omega_n^{2k+n}) + \omega_n^{k+n/2} A_1(\omega_n^{2k+n}) = A_0(\omega_n^{2k}\omega_n^n) + \omega_n^k \omega_n^{n/2} A_1(\omega_n^{2k}\omega_n^n) =$$
$$= A_0(\omega_n^{2k}) - \omega_n^k A_1(\omega_n^{2k}) = y_0^k - \omega_n^k y_1^k$$

Итак, мы получили формулы для вычисления всего вектора $\{y_k\}$.

Обратное преобразование Фурье почти не отличается от прямого: вместо ω_n^k необходимо использовать ω_n^{-k} и каждый элемент результата разделить на n.

Возведение в степень

Очевидно, что возведение в степень прямым способ длинных чисел будет выполняться очень долго. К счастью, существует алгоритм, работающий более быстро: бинарное возведение в степень, который выполняет всего лишь $O(\log n)$ умножений.

Заметим, что для любого числа a и для любого четного числа n выполняется следующее равенство:

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

Оно и является основным в методе бинарного возведения в степень. Действительно, для чётного n мы показали, как, потратив всего одну операцию умножения, можно свести задачу к вдвое меньшей степени.

Осталось понять, что делать, если степень n нечётна. Здесь мы поступаем очень просто: перейдём к степени $n-1$, которая будет уже чётной:

$$a^n = a^{n-1} \cdot a$$

Итак, мы фактически нашли рекуррентную формулу: от степени n мы переходим, если она чётна, к $n/2$, а иначе — к $n-1$. Понятно, что всего будет не более $2\log n$ переходов, прежде чем мы придём к $n=0$ (базе рекуррентной формулы). Таким образом, мы получили алгоритм, работающий за $O(\log n)$ умножений.

Деление

Немного про деление: это тоже довольно неочевидная операция. Существует алгоритм деления путем вычисления обратного к делителю с точностью до $1/2$ путем нахождения нуля функции $f(x)$ методом Ньютона, где

$$f(x) = 1 - Bx$$

Далее делитель умножается на число, обратное делителю, и находится частное. Зная частное, всегда можно найти остаток от деления.

Однако я решил выбрать более простой метод и реализовал школьное деление "в столбик".

Исходный код

Реализацию класса $BigInt$ я решил разбить на несколько файлов:

1. Файл с конструкторами ($Constructors.cpp$)
2. Файл с операциями сравнения ($Predicates.cpp$)
3. Файл с операциями сложения, вычитания, инкремента, декремента и прочими ($ArithmeticOperations.cpp$)
4. Файл с операциями умножения, деления, сдвига ($BigInt.cpp$)

Для преобразования Фурье я создал отдельный класс $Fourier$, содержащий два статических метода: преобразования Фурье и умножения полиномов.

```
#ifndef INC_6LAB_FOURIER_H
#define INC_6LAB_FOURIER_H

#include <vector>
#include <complex>
```

```
#define all(v) v.begin(), v.end()

typedef std::complex<double> complex;

class Fourier {
public:
    static void fft(std::vector<complex> &, bool invert);

    static void multiply(const std::vector<int> &a, const std::vector<int>
↪  &b,
                         std::vector<unsigned long long> &res);
};


#endif //INC_6LAB_FOURIER_H
```

```
#include "Fourier.h"

void Fourier::fft(std::vector<complex> &a, bool invert) {
    int n = (int) a.size();
    if (n == 1) return;

    std::vector<complex> a0(n / 2), a1(n / 2);
    for (int i = 0, j = 0; i < n; i += 2, ++j) {
        a0[j] = a[i];
        a1[j] = a[i + 1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * M_PI / n * (invert ? -1 : 1);
    complex w(1), wn(std::cos(ang), std::sin(ang));
    for (int i = 0; i < n / 2; ++i) {
        a[i] = a0[i] + w * a1[i];
        a[i + n / 2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n / 2] /= 2;
        }
        w *= wn;
    }
```

```cpp
}

void Fourier::multiply(const std::vector<int> &a, const std::vector<int> &b,
                       std::vector<unsigned long long> &res) {
    std::vector<complex> fa(all(a)), fb(all(b));
    size_t n = 1;
    while (n < std::max(fa.size(), fb.size())) {
        n <<= 1;
    }
    n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for (size_t i = 0; i < n; ++i) {
        fa[i] *= fb[i];
    }
    fft(fa, true);
    res.resize(n);
    for (size_t i = 0; i < n; ++i) {
        res[i] = (unsigned long long) (fa[i].real() + 0.5);
    }
}
```

```cpp
#ifndef INC_6LAB_BIGINT_H
#define INC_6LAB_BIGINT_H

#include <iostream>
#include <vector>
#include <exception>

#include "Fourier.h"

class BigInt {
public:
    BigInt();

    BigInt(const std::string &str);

    BigInt(signed char);
```

```
BigInt(unsigned char);
BigInt(signed short);
BigInt(unsigned short);
BigInt(signed int);
BigInt(unsigned int);
BigInt(signed long);
BigInt(unsigned long);
BigInt(signed long long);
BigInt(unsigned long long);
friend std::ostream &operator<<(std::ostream &os, const BigInt &anInt);
explicit operator std::string() const;
friend bool operator==(const BigInt &left, const BigInt &right);
friend bool operator<(const BigInt &left, const BigInt &right);
friend bool operator!=(const BigInt &left, const BigInt &right);
friend bool operator>(const BigInt &left, const BigInt &right);
friend bool operator>=(const BigInt &left, const BigInt &right);
friend bool operator<=(const BigInt &left, const BigInt &right);
BigInt operator+() const;
BigInt operator-() const;
friend BigInt operator+(const BigInt &left, const BigInt &right);
friend BigInt operator-(const BigInt &left, const BigInt &right);
```

```
    BigInt &operator+=(const BigInt &val);

    BigInt &operator-=(const BigInt &val);

    BigInt &operator++();

    BigInt operator++(int);

    BigInt &operator--();

    BigInt operator--(int);

    BigInt &operator*=(const BigInt &val);

    friend BigInt operator*(const BigInt &left, const BigInt &right);

    friend BigInt operator/(const BigInt &left, const BigInt &right);

    friend BigInt operator/(const BigInt &left, long long right);

    [[nodiscard]] long long toLL() const;

    [[nodiscard]] long double toDouble() const;

    BigInt &operator/=(const BigInt &val);

    friend BigInt operator^(BigInt a, BigInt n);

    friend BigInt operator^(BigInt a, unsigned long long n);

    int operator[](size_t index) const;

    bool odd(); // нечетное число

    bool even(); // четное число

    void shiftRight(); // смещение вправо на один разряд

private:
    void deleteLeadingZeroes();
```

```cpp
    bool negative_;
    std::vector<int> digits_;
    static const int BASE = 10000;
    static const int LENGTH = 4;
};


#endif //INC_6LAB_BIGINT_H
```

```cpp
#include <iomanip>
#include "BigInt.h"

BigInt operator*(const BigInt &left, const BigInt &right) {
    BigInt result;
    result.negative_ = left.negative_ or right.negative_;
    std::vector<unsigned long long> resultDigits;
    Fourier::multiply(left.digits_, right.digits_, resultDigits);
    unsigned long long perenos = 0;
    result.digits_.resize(resultDigits.size());
    for (auto &resultDigit: resultDigits) {
        resultDigit += perenos;
        perenos = resultDigit / BigInt::BASE;
        resultDigit %= BigInt::BASE;
    }
    while (perenos) {
        resultDigits.emplace_back(perenos % BigInt::BASE);
        perenos /= BigInt::BASE;
    }
    result.digits_.resize(resultDigits.size());
    for (size_t i = 0; i < resultDigits.size(); ++i) {
        result.digits_[i] = (int) resultDigits[i];
    }
    result.deleteLeadingZeroes();
    return result;
}

bool BigInt::odd() {
    if (digits_.empty()) {
        return false;
    }
```

```
    return digits_[0] & 1;
}

bool BigInt::even() {
    return !odd();
}

BigInt &BigInt::operator*=(const BigInt &val) {
    return (*this) = (*this * val);
}

BigInt operator/(const BigInt &left, const BigInt &right) {
    if (right == 0) {
        throw std::runtime_error("Divide by zero");
    }
    BigInt b = right;
    b.negative_ = false;
    BigInt result, current;
    result.digits_.resize(left.digits_.size());
    for (long long i = static_cast<long long>(left.digits_.size()) - 1; i >=
    ↪  0; --i) {
        current.shiftRight();
        current.digits_[0] = left.digits_[i];
        current.deleteLeadingZeroes();
        int x = 0, l = 0, r = BigInt::BASE;
        while (l <= r) {
            int m = (l + r) / 2;
            BigInt t = b * m;
            if (t <= current) {
                x = m;
                l = m + 1;
            } else r = m - 1;
        }

        result.digits_[i] = x;
        current = current - b * x;
    }

    result.negative_ = left.negative_ ^ right.negative_;
    result.deleteLeadingZeroes();
    return result;
}
```

```cpp
BigInt &BigInt::operator/=(const BigInt &val) {
    return (*this) = (*this / val);
}

BigInt operator^(BigInt a, BigInt n) {
    BigInt result(1);
    while (n != 0) {
        if (n.odd()) {
            result *= a;
        }
        a *= a;
        n /= 2;
    }
    return result;
}

BigInt operator^(BigInt a, unsigned long long int n) {
    BigInt result(1);
    while (n != 0) {
        if (n & 1) {
            result *= a;
        }
        a *= a;
        n >>= 2;
    }
    return result;
}

BigInt operator/(const BigInt &left, long long int right) {
    BigInt res(left);
    bool negative = right < 0;
    res.negative_ = left.negative_ ^ (negative);
    right = negative ? right * -1 : right;
    unsigned long long reserve = 0;
    for (size_t i = res.digits_.size() - 1; i < res.digits_.size(); --i) {
        unsigned long long cur = res.digits_[i] + reserve * BigInt::BASE;
        res.digits_[i] = static_cast<int>(cur / right);
        reserve = cur % right;
    }
    res.deleteLeadingZeroes();
    return res;
```

```
}

long long BigInt::toLL() const {
    if (digits_.size() < 2) {
        return digits_[0] * (negative_ ? -1 : 1);
    }
    return (digits_[0] + BASE * digits_[1]) * (negative_ ? -1 : 1);
}

long double BigInt::toDouble() const {
    long double res = 0;
    for (size_t i = digits_.size() - 1; i < digits_.size(); --i) {
        res = res * BASE + digits_[i];
    }
    return res;
}

void BigInt::shiftRight() {
    if (this->digits_.empty()) {
        this->digits_.push_back(0);
        return;
    }
    this->digits_.push_back(this->digits_[this->digits_.size() - 1]);
    // здесь размер массива равен как минимум двум и перебор идет до
↪   предпоследнего разряда,
    // поэтому i имеет "верный" тип size_t
    for (size_t i = this->digits_.size() - 2; i > 0; --i) {
        this->digits_[i] = this->digits_[i - 1];
    }
    this->digits_[0] = 0;
}

int BigInt::operator[](size_t index) const {
    return (index < digits_.size()) ? digits_[index] : 0;
}
```

```
#include "BigInt.h"
#include <iomanip>

BigInt::BigInt() : negative_(false), digits_(1, 0) {}
```

```
BigInt::BigInt(signed char c) {
    negative_ = (c < 0);
    digits_.emplace_back(std::abs(c));
}

BigInt::BigInt(unsigned char c) {
    negative_ = false;
    digits_.emplace_back(c);
}

BigInt::BigInt(short sh) {
    negative_ = (sh < 0);
    digits_.emplace_back(std::abs(sh));
}

BigInt::BigInt(unsigned short c) {
    negative_ = false;
    digits_.emplace_back(c);
}

BigInt::BigInt(int l) {
    negative_ = (l < 0);
    l = std::abs(l);
    do {
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(unsigned int l) {
    negative_ = false;
    do {
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(long l) {
    negative_ = (l < 0);
    l = std::abs(l);
    do {
```

```
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(unsigned long l) {
    negative_ = false;
    do {
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(long long int l) {
    negative_ = (l < 0);
    l = std::abs(l);
    do {
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(unsigned long long int l) {
    negative_ = false;
    do {
        this->digits_.emplace_back(l % BigInt::BASE);
        l /= BigInt::BASE;
    } while (l != 0);
}

BigInt::BigInt(const std::string &str) {
    negative_ = false;
    if (str.empty()) {
        digits_.resize(1, 0); // для однозначного представления нуля
    } else {
        std::string digits = str;
        if (str[0] == '-' or str[0] == '+') {
            digits = str.substr(1);
            negative_ = (str[0] == '-');
        }
        for (long long i = digits.size(); i > 0; i -= LENGTH) {
            if (i < LENGTH) {
```

```
                digits_.emplace_back(std::atoi(digits.substr(0,
↪   i).c_str()));
            } else {
                digits_.emplace_back(std::atoi(digits.substr(i -
↪   BigInt::LENGTH, BigInt::LENGTH).c_str()));
            }
        }
        deleteLeadingZeroes();
    }
}

std::ostream &operator<<(std::ostream &os, const BigInt &anInt) {
    if (anInt.digits_.empty()) {
        os << 0;
        return os;
    }
    if (anInt.negative_) {
        os << '-';
    }
    os << anInt.digits_.back();
    /*if (anInt.digits_.size() > 1) {
        os << "\'";
    }*/
    for (long long i = anInt.digits_.size() - 2; i >= 0; --i) {
        os << std::setw(BigInt::LENGTH) << std::setfill('0') <<
↪   anInt.digits_[i];
/*        if (i != 0) {
            os << "\'";
        }*/
    }
    return os;
}

BigInt::operator std::string() const {
    std::stringstream ss;
    ss << *this;
    return ss.str();
}

void BigInt::deleteLeadingZeroes() {
    while (digits_.size() > 1 and digits_.back() == 0) {
        digits_.pop_back();
```

```
    }
    if (digits_.size() == 1 and digits_.back() == 0) {
        negative_ = false;
    }
}
```

```
#include "BigInt.h"

bool operator==(const BigInt &left, const BigInt &right) {
    if (left.negative_ != right.negative_ or left.digits_.size() !=
 ↪  right.digits_.size()) {
        // если разного знака или разного размера, то они не равны (имея
 ↪  однозначное представление 0, можно не опасаться ошибок)
        return false;
    }
    for (size_t i = 0; i < left.digits_.size(); ++i) {
        if (left.digits_[i] != right.digits_[i]) {
            return false;
        }
    }
    return true;
}

bool operator<(const BigInt &left, const BigInt &right) {
    // случаи разных знаков
    if (left.negative_ and !right.negative_) {
        return true; // отрицательное число меньше положительного
    } else if (!left.negative_ and right.negative_) {
        // положительное число не меньше отрицательного
        return false;
    }
    // случай одинаковых знаков
    if (left.digits_.size() != right.digits_.size()) {
        return (left.negative_) ? left.digits_.size() > right.digits_.size()
                                : left.digits_.size() <
 ↪  right.digits_.size();
    }
    for (long long i = left.digits_.size() - 1; i >= 0; --i) {
        if (left.digits_[i] != right.digits_[i]) {
            return left.digits_[i] < right.digits_[i];
```

```
        }
        if (i == 0) {
            break;
        }
    }
    // если оба числа равны
    return false;
}

bool operator!=(const BigInt &left, const BigInt &right) {
    return !(left == right);
}

bool operator>(const BigInt &left, const BigInt &right) {
    return (right < left);
}

bool operator>=(const BigInt &left, const BigInt &right) {
    return !(left < right);
}

bool operator<=(const BigInt &left, const BigInt &right) {
    return !(left > right);
}
```

```
#include "BigInt.h"

BigInt BigInt::operator+() const {
    return {*this};
}

BigInt BigInt::operator-() const {
    BigInt copy(*this);
    copy.negative_ = !this->negative_;
    return {copy};
}

BigInt operator+(const BigInt &left, const BigInt &right) {
    if (left.negative_) {
        if (right.negative_) {
```

```cpp
            // оба отрицательных
            return -(-left + (-right));
        } else {
            // левое число отрицательное
            return right - (-left);
        }
    } else if (right.negative_) {
        // право число отрицательное
        return left - (-right);
    }
    // оба числа положительные
    int perenos = 0; // не нашел адекватного названия на англе
    BigInt res(left);
    size_t size = std::max(left.digits_.size(), right.digits_.size());
    res.digits_.resize(std::max(left.digits_.size(), right.digits_.size()));
    for (size_t i = 0; i < size; ++i) {
        res.digits_[i] += perenos + right[i];
        perenos = res.digits_[i] / BigInt::BASE;
        res.digits_[i] %= BigInt::BASE;
    }
    if (perenos) {
        res.digits_.emplace_back(perenos);
    }
    return res;
}

BigInt operator-(const BigInt &left, const BigInt &right) {
    if (right.negative_) {
        // - на - дает +
        return left + (-right);
    }
    if (left.negative_) {
        // вычитание = сложение двух отрицательных => выносим -
        return -(-left + right);
    }
    if (left < right) {
        // меняем местами, чтобы меньшее число вычиталось из большего
        return -(right - left);
    }
    BigInt res(left);
    int perenos = 0;
    for (size_t i = 0; i < res.digits_.size(); ++i) {
```

```
        res.digits_[i] -= perenos + right[i];
        perenos = res.digits_[i] < 0;
        if (perenos) {
            res.digits_[i] += BigInt::BASE;
        }
    }
    res.negative_ = false;
    res.deleteLeadingZeroes();
    return res;
}

BigInt &BigInt::operator+=(const BigInt &val) {
    return *this = (*this + val);
}

BigInt &BigInt::operator-=(const BigInt &val) {
    return *this = (*this - val);
}

BigInt &BigInt::operator++() {
    (*this) += 1;
    return *this;

}

BigInt BigInt::operator++(int) {
    BigInt old(*this);
    (*this) += 1;
    return old;
}

BigInt &BigInt::operator--() {
    (*this) -= 1;
    return *this;
}

BigInt BigInt::operator--(int) {
    BigInt old(*this);
    (*this) -= 1;
    return old;
}
```

```cpp
#include <iostream>

#include "BigInt.h"

int main() {
    std::string a, b, sign;
    while (std::cin >> a >> b >> sign) {
        BigInt first(a), second(b);
        if (sign == "+") {
            std::cout << (first + second) << std::endl;
        } else if (sign == "-") {
            if (first < second) {
                std::cout << "Error" << std::endl;
            } else {
                std::cout << (first - second) << std::endl;
            }
        } else if (sign == "*") {
            std::cout << (first * second) << std::endl;
        } else if (sign == "^") {
            if (first == 0 and second == 0) {
                std::cout << "Error" << std::endl;
            } else {
                std::cout << (first ^ second) << std::endl;
            }
        } else if (sign == "/") {
            if (second == 0) {
                std::cout << "Error" << std::endl;
            } else {
                std::cout << (first / second) << std::endl;
            }
        } else if (sign == ">") {
            std::string ans = (first > second) ? "true" : "false";
            std::cout << ans << std::endl;
        } else if (sign == "<"){
            std::string ans = (first < second) ? "true" : "false";
            std::cout << ans << std::endl;
        } else if (sign == "="){
            std::string ans = (first == second) ? "true" : "false";
            std::cout << ans << std::endl;
        }
    }
```

```
    return 0;
}
```

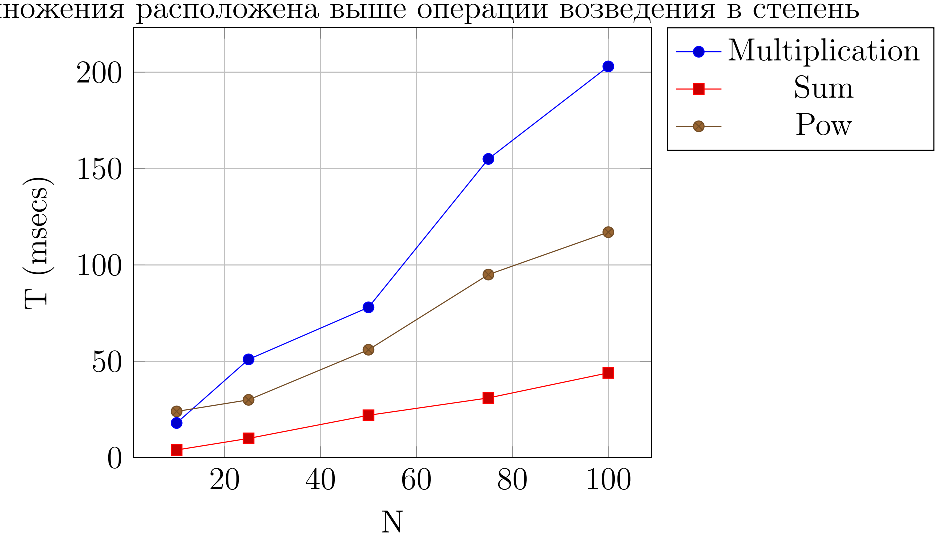
Дневник отладки

Во время реализация я столкнулся с небольшими багами в программе, над устранением которых я долго ломал мозг. Приведу некоторые из них:

- Изначально в качестве основания системы счисления я взял число 10^9. Переполнений было не избежать, как бы я не старался. В конечном итоге я выбрал число поменьше.
- При сложении долгое время плохо работал перенос разрядов в виду из-за моего неправильного алгоритма сложения.
- Перенос при вычитании тоже работал неправильно. Я добавлял к текущему разряду число, но забыл убирать его у предыдущего разряда.
- Операция сравнения могла работать неправильно из-за утечки памяти. При инициализации переменной итерирования я устанавливал значение, равное длине вектора. Правильным было значение длины вектора - 1.

Тест производительности

Протестируем работу моей программы на входных данных различной длинны. Возьмем основные операции: сложение (выполняется за $O(n)$), умножение (выполняется за $O(n \log n)$), возведение в степень (выполняется за $O(\log n)$, где n - степень). В качестве n возьмем следующие числа: 10, 25, 50, 75, 100. Для возведения степень основание степени возьмем равным число вида $999\ldots99$ размера 256. Операции сложения и умножения замерялись группой по 100 операций. Поэтому неудивительно, что на графике операция умножения расположена выше операции возведения в степень



Выводы

Длинная арифметика может применяться в криптографии. Большинство систем подписывания и шифрования данных используют целочисленную арифметику по модулю m, где m — очень большое натуральное число, не обязательно простое. Например, при реализации метода шифрования RSA, криптосистемы Рабина или схемы Эль-Гамаля требуется обеспечить точность результатов умножения и возведения в степень порядка 10^{309} (сам пока не сталкивался, но верю Википедии на слово).

Я доволен, что смог реализовать не самую простую реализацию длинной арифметики(сумел оптимизировать умножение). Конечно, жаль, что я не смог реализовать быстрое деление, но всё же существующая реализация достаточно быстрая, поэтому не стоит отчаиваться. Надеюсь, данная библиотека поможет мне в будущем.