Национальный исследовательский университет
"Московский авиационный институт"
Факультет No8 "Информационные технологии и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

ЛАБОРАТОРНАЯ РАБОТА №6
ПО КУРСУ “ДИСКРЕТНЫЙ АНАЛИЗ”
4 СЕМЕСТР

Выполнил студент: Поляков А.И.
Группа: М80-208Б-19
Оценка:
Подпись:

Москва 2022

Лабораторная работа №6

Вариант №1.

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.

Количество десятичных разрядов целых чисел не превышает 100000. Основание выбранной системы счисления для внутреннего представления «длинных» чисел должно быть не меньше 10000.

Формат входных данных

Входной файл состоит из последовательности заданий, каждое задание состоит из трех строк:

Первый операнд операции.

Второй операнд операции.

Символ арифметической операции или проверки условия (+, -, *, ; /, >, <, =).

Числа, поступающие на вход программе, могут иметь «ведущие» нули.

Формат результата

Для каждого задания из выходного файла нужно распечатать результат на отдельной строке в выходном файле:

Числовой результат для арифметических операций.

Строку Error в случае возникновения ошибки при выполнении арифметической операции.

Строку true или false при выполнении проверки условия.

В выходных данных вывод чисел должен быть нормализован, то есть не содержать в себе «ведущих» нулей.

Описание

Длинная арифметика - набор алгоритмов для поразрядной работы с числами произвольной длины. Она применяется как с относительно небольшими числами, превышающими ограничения типа *long long* в несколько раз, так и с по-настоящему большими числами (чаще всего до 10^{100000}).

Требуется написать реализацию программной библиотеки на C или C++ для работы с длинными числами.

Для сложения и вычитания воспользуемс наивными алгоритмами сложения, вычитания в столбик, сложность *O(n)*, где n — длина самого длинного операнда. При сравнении чисел будем, сначала сравнивать их длины, если они равны, то перейдём к сравнениВ разрядов, начиная со старшего за *O(n)*.

Умножение реализуем наивным алгоритмом умножения в столбик. Сложность $O(n * m)$, где n, m — длины чисел. Также реализуем алгоритм Карацубы, сложность которого $O(n^{log_2 3})$, достигается путём разбиения чисел на части длины $n/2$ и трёх умножений.

Деление реализуем наивным алгоритмом деления в столбик, однако на каждой итерации частное, которое будет записываться в ответ, будем находить бинарным поиском. Получим сложность $O(n * m)$. Также реализуем бинарное возведение в степень, тогда понадобится $O(log_2 * exp)$ умножений, а не $O(exp)$ в случае наивного алгоритма, exp — показателя степени.

Исходный код

Сначала, объявим класс *BigInt_e6*, который будем использовать для работы с длинными числами. Напишем сигнатуры основных методов работы с данными числами и перегрузок операторов арифметических действий. Также, объявим конструктор от строки и от длины числа и значения разряда. Для того, чтобы вычисления выполнялись быстрее, будем хранить сразу по 6 цифр числа в одном разряде.

Объявление класса

```
namespace BigInt_e6 {

class TBigInt_e6 {
    public:
        static const int BASE = 1e6;
        static const int RADIX = 6;

        TBigInt_e6() = default;
        TBigInt_e6(const std::string & str);
        TBigInt_e6(const size_t & length, const long long value = 0);
        void Initialize(const std::string & str);
        std::string GetStr() const;

        size_t Size() const;
        TBigInt_e6 Power(const TBigInt_e6 & degree) const;
        void Shift(const long long degree);
        static TBigInt_e6 KaratsubaMult(TBigInt_e6 && lhs, TBigInt_e6 && r

        TBigInt_e6 & operator=(const TBigInt_e6 & rhs);

        TBigInt_e6 operator+(const TBigInt_e6 & rhs) const;
        TBigInt_e6 operator-(const TBigInt_e6 & rhs) const;
        TBigInt_e6 operator*(const TBigInt_e6 & rhs) const;
        TBigInt_e6 operator/(const TBigInt_e6 & rhs) const;

        TBigInt_e6 operator-(const long long rhs) const;
        TBigInt_e6 operator*(const long long rhs) const;
        TBigInt_e6 operator/(const long long rhs) const;
        long long   operator%(const long long rhs) const;

        bool operator< (const TBigInt_e6 & rhs) const;
        bool operator<=(const TBigInt_e6 & rhs) const;
        bool operator> (const TBigInt_e6 & rhs) const;
        bool operator==(const TBigInt_e6 & rhs) const;
```

```
        bool operator==(const long long     rhs) const;
        bool operator> (const long long     rhs) const;

        friend std::istream& operator>>(std::istream &is, TBigInt_e6 & rhs
        friend std::ostream& operator<<(std::ostream &os, const TBigInt_e6
    private:
        std::vector<long long> digits;

        void DeleteLeadingZeros();
};

}
```

Создадим методы инициализации, удаления ведущих нулей, и два конструктора.

```
TBigInt_e6::TBigInt_e6(const std::string & str) {
    this->Initialize(str);
}

TBigInt_e6::TBigInt_e6(const size_t & length, const long long value)
    : digits(length, value)
{}

void TBigInt_e6::Initialize(const std::string & str) {
    long long startIdx = 0;
    while (startIdx < str.size() && str[startIdx] == '0') {
        ++startIdx;
    }
    if (startIdx == str.size()) {
        digits.push_back(0);
        return;
    }

    digits.clear();
    size_t digitsSize = (str.size() - startIdx) / RADIX;
    if ((str.size() - startIdx) % RADIX != 0) {
        ++digitsSize;
    }
    digits.resize(digitsSize);

    size_t digitsCount = 0;
    for (long long i = str.size() - 1; i >= startIdx; i -= RADIX) {
```

```
            long long currDigit = 0;
            long long digitStart = i - RADIX + 1;
            if (digitStart < 0 || digitStart <= startIdx) {
                digitStart = 0;
            }
            for (long long j = digitStart; j <= i; ++j) {
                currDigit = currDigit * 10 + str[j] - '0';
            }

            digits[digitsCount] = currDigit;
            ++digitsCount;
        }
    }

    void TBigInt_e6::DeleteLeadingZeros() {
        while (Size() > 1 && digits.back() == 0) {
            digits.pop_back();
        }
    }
```

Далее перегрузим операторы ввода-вывода больших чисел.

```
    // Streams operators

    std::istream& operator>>(std::istream & is, TBigInt_e6 & rhs) {
        std::string str;
        is >> str;
        rhs.Initialize(str);
        return is;
    }

    std::ostream& operator<<(std::ostream & os, const TBigInt_e6 & rhs) {
        os << rhs.digits[rhs.Size()- 1];
        for (long long i = rhs.Size() - 2; i >= 0; --i) {
            os << std::setfill('0') << std::setw(TBigInt_e6::RADIX) << rhs.dig
        }
        return os;
    }

    std::string TBigInt_e6::GetStr() const {
        std::stringstream ss;
        ss << *this;
        return ss.str();
    }
```

Перейдем к основным операциям работы с числами. Длинную арифметику часто сравнивают с вычислением “в столбик”. Это достаточно справедливо, так как методы сложения и вычитания основаны на поразрядных операциях.

Реализация умножения немного отличается от алгоритма умножения в столбик, но принцип сохраняется: перемножим каждый разряд одного числа на каждый разряд другого. При умножении разряда i на разряд j добавим результат к разряду i+j произведения (0-индексация). После этого выполним переносы аналогично сложению.

В отличие от других арифметических операций, деление длинного числа на другое длинное реализуется достаточно сложно, и в школьных задачах вам вряд ли придётся им пользоваться.

Деление на короткое число (меньше размера разряда), напротив, реализуется очень просто. Просто делим по очереди каждый разряд длинного числа на короткое, сохраняем целую часть, а остаток переносим в предыдущий (младший) разряд.

```
BigInt_e6 TBigInt_e6::operator+(const TBigInt_e6 & rhs) const {
    size_t resSize = std::max(rhs.Size(), Size());
    TBigInt_e6 res(resSize);
    long long carry = 0;

    for (size_t i = 0; i < resSize; ++i) {
        long long sum = carry;
        if (i < rhs.Size()) {
            sum += rhs.digits[i];
        }
        if (i < Size()) {
            sum += digits[i];
        }
        carry = sum / BASE;
        res.digits[i] = sum % BASE;
    }
    if (carry != 0) {
        res.digits.push_back(carry);
    }
    res.DeleteLeadingZeros();
    return res;
}

TBigInt_e6 TBigInt_e6::operator-(const TBigInt_e6 & rhs) const {
    if (*this < rhs) {
        throw std::logic_error("Error: trying to subtract bigger number fr
    }
```

```
    size_t resSize = std::max(rhs.Size(), Size());
    long long carry = 0;
    TBigInt_e6 res(resSize);

    for (size_t i = 0; i < resSize; ++i) {
        long long diff = digits[i] - carry;
        if (i < rhs.Size()) {
            diff -= rhs.digits[i];
        }

        if (diff < 0) {
            carry = 1;
            diff += BASE;
        } else {
            carry = 0;
        }
        res.digits[i] = diff % BASE;
    }
    res.DeleteLeadingZeros();
    return res;
}

TBigInt_e6 TBigInt_e6::operator*(const TBigInt_e6 & rhs) const {
    TBigInt_e6 res(Size() + rhs.Size());
    for (size_t i = 0; i < Size(); ++i) {
        long long carry = 0;
        for (size_t j = 0; j < rhs.Size() || carry > 0; ++j) {
            long long current = res.digits[i + j] + carry;
            if (j < rhs.Size()) {
                current += digits[i] * rhs.digits[j];
            }
            res.digits[i + j] = current % BASE;
            carry = current / BASE;
        }
    }
    res.DeleteLeadingZeros();
    return res;
}

TBigInt_e6 TBigInt_e6::operator/(const TBigInt_e6 & rhs) const {
    if (rhs == 0) {
        throw std::logic_error("Error: Trying to divide by zer0");
```

```
    }

    TBigInt_e6 res;
    TBigInt_e6 curr;
    for (size_t i = Size() - 1; i < Size(); --i) {
        curr.digits.insert(std::begin(curr.digits), digits[i]);
        curr.DeleteLeadingZeros();

        long long currResDigit = 0;
        long long leftBound = 0;
        long long rightBound = BASE;
        while (leftBound <= rightBound) {
            long long middle = (leftBound + rightBound) / 2;
            TBigInt_e6 tmp = rhs * middle;
            if (tmp <= curr) {
                currResDigit = middle;
                leftBound = middle + 1;
            }
            else {
                rightBound = middle - 1;
            }
        }

        res.digits.insert(std::begin(res.digits), currResDigit);
        curr = curr - rhs * currResDigit;
    }

    res.DeleteLeadingZeros();
    return res;

}


TBigInt_e6 TBigInt_e6::operator*(const long long rhs) const {
    TBigInt_e6 res(Size());
    long long carry = 0;
    for (size_t i = 0; i < Size() || carry > 0; ++i) {
        long long currDigit = carry;
        if (i == Size()) {
            res.digits.push_back(0);
        } else {
            currDigit += digits[i] * rhs;
        }
```

```
        res.digits[i] = currDigit % BASE;
        carry = currDigit / BASE;
    }
    res.DeleteLeadingZeros();
    return res;
}

TBigInt_e6 TBigInt_e6::operator/(const long long rhs) const {
    TBigInt_e6 res(Size());
    long long carry = 0;
    for (size_t i = Size() - 1; i < Size(); --i) {
        long long currDigit = carry * BASE + digits[i];
        res.digits[i] = currDigit / rhs;
        carry = currDigit % rhs;
    }
    res.DeleteLeadingZeros();
    return res;
}
```

Затем реализуем бинарное возведение в степень данного числа.

```
TBigInt_e6 TBigInt_e6::Power(const TBigInt_e6 & degree) const {
    if (*this == 0 && degree == 0) {
        throw std::logic_error("Error: 0^0 is uncertain");
    }

    TBigInt_e6 res("1");
    if (degree == 0) {
        return res;
    }

    TBigInt_e6 curr = *this;
    TBigInt_e6 currDegree = degree;
    while (currDegree > 0) {
        if (currDegree.digits.back() % 2 != 0) {
            res = res * curr;
        }

        curr = curr * curr;
        currDegree = currDegree / 2;
    }
    return res;
}
```

Перейдем к реализации поразрядного сравнения.

```
bool TBigInt_e6::operator< (const TBigInt_e6 & rhs) const {
    if (Size() != rhs.Size()) {
        return Size() < rhs.Size();
    }

    for (size_t i = Size() - 1; i < Size(); --i) {
        if (digits[i] != rhs.digits[i]) {
            return digits[i] < rhs.digits[i];
        }
    }

    return false;
}

bool TBigInt_e6::operator<= (const TBigInt_e6 & rhs) const {
    return !(*this > rhs);
}

bool TBigInt_e6::operator> (const TBigInt_e6 & rhs) const {
    if (Size() != rhs.Size()) {
        return Size() > rhs.Size();
    }

    for (size_t i = Size() - 1; i < Size(); --i) {
        if (digits[i] != rhs.digits[i]) {
            return digits[i] > rhs.digits[i];
        }
    }

    return false;
}

bool TBigInt_e6::operator==(const TBigInt_e6 & rhs) const {
    if (Size() != rhs.Size()) {
        return false;
    }

    for (size_t i = Size() - 1; i < Size(); --i) {
        if (digits[i] != rhs.digits[i]) {
            return false;
        }
```

```
    }

    return true;
}

bool TBigInt_e6::operator==(const long long rhs) const {
    if (Size() != 1) {
        return false;
    }
    return digits.back() == rhs;
}

bool TBigInt_e6::operator> (const long long rhs) const {
    if (Size() > 1) {
        return true;
    }
    return digits.back() > rhs;
}
```

Тест производительности

ест производительности представляет из себя сравнение производительности напи- санного класса TBigInt_e6 с библиотекой GMP. С помощью test_generator.py про- исходит генерация тестового файла, содержащего n - ое количество входных данных для каждой операции. Сравнивается суммарное время выполнения операций каждого вида без учёта ввода для 100, 1000 тестовых данных.

```
user3AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab6$ ./wrapper.sh 100 b
[2021-03-31 21:14:26] [INFO] Compiling...
make: 'benchmark'is up to date.
[2021-03-31 21:14:28] [INFO] Generating tests (examples for each test=[100
[2021-03-31 21:14:39] [INFO] Executing tests...
==============================
===== ADDITION =====
TBiggestInt Time = 34 ms
GMP Time = 20 ms
===== SUBTRACTION =====
TBiggestInt Time = 12 ms
GMP Time = 15 ms
===== MULTIPLICATION =====
TBiggestInt Time = 35 ms
Karatsuba Time = 10333 ms
GMP Time = 24 ms
===== DIVISION =====
TBiggestInt Time = 2998 ms
GMP Time = 22 ms
===== POWER =====
TBiggestInt Time = 57497501 ms
GMP Time = 19158 ms
===== LESS THEN =====
TBiggestInt Time = 4 ms
GMP Time = 4 ms
===== EQUAL TO =====
TBiggestInt Time = 3 ms
GMP Time = 2 ms
```

Вывод

Выполнив шестую лабораторную работу по курсу «Дискретный анализ», я познакомился с длинной арифметикой, реализовал свой класс для работы с длинными числами, что актуально для C++, поскольку он не имеет встроенной поддержки данных чисел.

Полученный опыт может пригодится при работе в математической сфере, криптографии и особенно в спортивном программировании, где достаточно часто встречаются задачи на длинную арифметику.

Литература

[1] Классические алгоритмы поиска образца в строке, Андрей Калинин. 28 октября 2011 г.

[2] Дэн Гасфилд. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — Издательский дом «Невский Диалект», 200ц. Перевод с английского: И. В. Романовского. — 654с. (ISBN 5-7940-0103-8 (рус.))