Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 6 по курсу дискретного анализа: Длинная арифметика

Студент: А. О. Тояков
Преподаватель: А. Н. Ридли
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва
2021

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:

1. Сложение (+).
2. Вычитание (-).
3. Умножение (*).
4. Возведение в степень ($\wedge$).
5. 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:

1. Больше (>).
2. Меньше (<).
3. Равно (=).

Метод решения

Для начала нужно определить подходящий base для моей системы счисления. В моей СС base = 1000000, то есть один разряд моего реализуемого числа принадлежит промежутку 0 <= x < 1000000. Сложность операции деления и возведения в степень будут указаны, пренебрегая сложностью внутренних операций.

1. Операция сложения: данная операция реализуется с помощью прохода по двум числам от младшего разряда к старшему с последующим сложением. Сложение реализуется по правилу сложения столбиком. Сложность O(n), где n - размер максимального числа среди двух данных.
2. Операция вычитания: данная операция практически аналогична сложению, только происходит вычитание. Также если разность двух разрядов числа отрицательна, то необходимо взять 1 с большего разряда. Вычитание также реализуется по правилу вычитания в столбик. Сложность O(n), где n - размер максимального числа среди двух данных.

3. Операция умножения: данная операция аналогична операции умножения в столбик. Для начала определяется максимальный размер числа result, которое может получиться в процессе умножения. result инициализируется нулями, а затем последовательно от младшего разряда к старшему умножаются разряды первого множителя на второй. Сложность O(n * m), где n - длина первого множителя, а m -второго.

4. Операция деления: логика схожа с делением уголком. Начиная со старшего разряда, мы получаем делимое и находим частное с помощью бинарного поиска при делении на делитель. Затем мы отнимаем полученное частное, умноженное на делитель, от делимого. Сложность O(n * log m), где n - число разрядов делителя, а m - сложность бинарного поиска.

5. Операция возведения в степень: реализовано бинарное возведение в степень. Сложность O(log n), где n - степень числа.

Описание программы

В данной программе для хранения числа я использовал шаблон std::vector, а также тип long long. В классе BigInt перегружены все операторы, необходимые по заданию. В данной работе понадобилось три конструктора: конструктор по умолчанию, конструктор, инициализирующий вектор нулями и конструктор, принимающий строку на вход. Операция возведения в степень вызывается как функция, где на вход подаётся число типа int, являющееся степенью числа. Также перегружен оператор вывода(»). Число печатается слева-направо, и в процессе вывода разряды числа дополняются ведущими нулями.

Исходный код

```
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>
#include <stdlib.h>
#include <vector>

const int base = 1000000;
const int bpow = 6;

class TBigInt {
private:
    std::vector<long long> number;
```

```
    void DeleteZero() {
        while (number.size() > 1 && number.back() == 0) {
            number.pop_back();
        }
    }

public:
    TBigInt(const std::string& input) {
        std::stringstream tempstr;
        for (auto i = (long long)input.size(); i > 0; i-= bpow) {
            if (i > bpow) {
                tempstr << input.substr(i - bpow,bpow);
                long long tempnumber;
                tempstr >> tempnumber;
                number.push_back(tempnumber);
                tempstr.clear();
            } else {
                tempstr << input.substr(0,i);
                long long tempnumber;
                tempstr >> tempnumber;
                number.push_back(tempnumber);
                tempstr.clear();
            }
        }
        DeleteZero();
    }

    TBigInt() : number(0)
    {}

    TBigInt(int n) : number(n,0)
    {}

    friend std::ostream &operator <<(std::ostream& stream, const TBigInt& other) {
        if (other.number.size() == 0) {
            return stream;
        }
        stream << other.number[other.number.size() - 1];
        for (int i = other.number.size() - 2;i >= 0;--i)
        {
            stream << std::setfill('0') << std::setw(bpow) << other.number[i];
        }
```

```
    return stream;
}

TBigInt operator +(const TBigInt &other) const {
    size_t size = std::max(number.size(), other.number.size());
    TBigInt result;
    long long r = 0;
    long long k = 0;
    for (size_t i = 0; i < size; i++) {
        if (number.size() <= i) {
            k = other.number[i];
        } else if (other.number.size() <= i) {
            k = number[i];
        } else {
            k = number[i] + other.number[i];
        }
        k += r;
        result.number.push_back(k % base);
        r = k / base;
    }
    if (r != 0) {
        result.number.push_back(r);
    }
    return result;
}

TBigInt operator -(const TBigInt &other) const {
    size_t size = std::max(number.size(),other.number.size());
    TBigInt result;
    long long r = 0;
    long long k = 0;// для взятия недостатка большего числа(0 или -1)
    for (size_t i = 0; i < size;i++) {
        long long res = 0;
        if (other.number.size() <= i) {
            res = number[i] + k;
        } else {
            res = number[i] - other.number[i] + k;
        }
        k = 0;
        if (res < 0) {
            res += base;
            k = -1;
```

```
        }
        r = res % base;
        result.number.push_back(r);
    }
    result.DeleteZero();
    return result;
}

TBigInt operator *(const TBigInt &other) const {
    size_t size = number.size() * other.number.size();
    TBigInt result(size + 1);
    long long k = 0;
    long long r = 0;
    for (size_t i = 0; i < number.size(); i++) {
        for (size_t j = 0; j < other.number.size(); j++) {
            k = other.number[j] * this->number[i] + result.number[i+j];
            r = k / base;
            result.number[i + j + 1] = result.number[i + j + 1] + r;
            result.number[i + j] = k % base;
        }
    }
    result.DeleteZero();
    return result;
}

TBigInt operator /(const TBigInt &other) const {
    TBigInt cv(1);
    TBigInt result(number.size());
    for (auto i = (long long)(number.size() - 1); i >= 0; --i) {
        cv.number.insert(cv.number.begin(), number[i]);
        if (!cv.number.back()) {// уборка первого нуля
            cv.number.pop_back();
        }
        long long x = 0, l = 0, r = base;
        // бинарным поиском нахождение частного
        while (l <= r) {
            long long m = (l + r) / 2;// middle
            TBigInt cur = other * TBigInt(std::to_string(m)); // находим самое прибл
            if (cur < cv || cur == cv) {// когда нашли
                x = m;
                l = m + 1;
            } else {
```

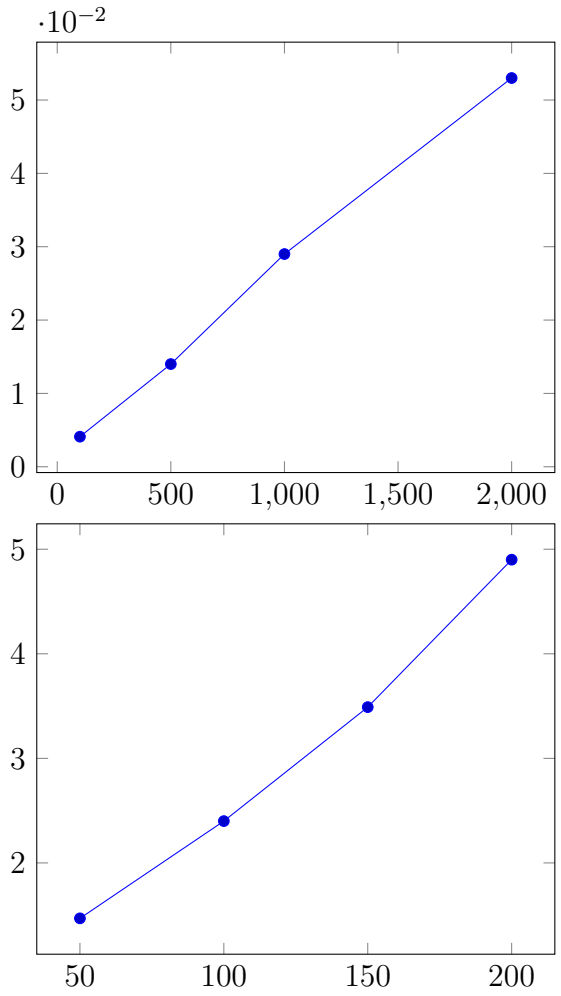
```
                r = m - 1;
            }
        }
        // x - частное, потом отнимается от cv (other * частное(деление уголком))
        result.number[i] = x;
        cv = cv - other * TBigInt(std::to_string(x));
    }
    result.DeleteZero();
    return result;
}

TBigInt Power(int power) {
    // бинарное возведение в степень
    TBigInt result("1");
    while (power) {
        if (power % 2) {
            result = result * (*this);
        }
        (*this) = (*this) * (*this);
        power /= 2;
    }
    return result;
}

bool operator >(const TBigInt &other) const {
    if (number.size() != other.number.size()) {
        return number.size() > other.number.size();
    }
    for (auto i = (long long)(number.size() - 1); i >= 0; i--) {
        if (number[i] != other.number[i]) {
            return number[i] > other.number[i];
        }
    }
    return false;
}

bool operator <(const TBigInt &other) const {
    if (number.size() != other.number.size()) {
        return number.size() < other.number.size();
    }
    for (auto i = (long long)(number.size() - 1); i >= 0; i--) {
        if (number[i] != other.number[i]) {
```

```
                return number[i] < other.number[i];
            }
        }
        return false;
    }

    bool operator ==(const TBigInt &other) const {
        if (number.size() != other.number.size()) {
            return false;
        }
        for (auto i = (long long)(number.size() - 1); i >= 0; i--) {
            if (number[i] != other.number[i]) {
                return false;
            }
        }
        return true;
    }
};

int main() {
    std::string fp,sp,op;
    while(std::cin >> fp >> sp >> op) {
        if (op == "+") {
            TBigInt first(fp);
            TBigInt second(sp);
            std::cout << first + second << std::endl;
        } else if (op == "-") {
            TBigInt first(fp);
            TBigInt second(sp);
            if (first < second) {
                std::cout << "Error" << std::endl;
                continue;
            }
            std::cout << first - second << std::endl;
        } else if (op == "/") {
            TBigInt first(fp);
            TBigInt second(sp);
            if (second == TBigInt(1)) {
                std::cout << "Error" << std::endl;
                continue;
            }
            std::cout << first / second << std::endl;
```

```
        } else if (op == "*") {
            TBigInt first(fp);
            TBigInt second(sp);
            std::cout << first * second << std::endl;
        } else if (op == "^") {
            TBigInt first(fp);
            int n = atoi(sp.c_str());
            if (first == TBigInt(1) && n == 0) {
                std::cout << "Error" << std::endl;
                continue;
            }
            std::cout << first.Power(n) << std::endl;
        } else if (op == ">") {
            TBigInt first(fp);
            TBigInt second(sp);
            std::cout << ((first > second) ? "true" : "false") << std::endl;
        } else if (op == "=") {
            TBigInt first(fp);
            TBigInt second(sp);
            std::cout << ((first == second) ? "true" : "false") << std::endl;
        } else if (op == "<") {
            TBigInt first(fp);
            TBigInt second(sp);
            std::cout << ((first < second) ? "true" : "false") << std::endl;
        }
    }
    return 0;
}
```

Тест производительности

Тесты представляют из себя набор чисел, где размер числа равен принадлежит промежутку от [1, 10000]. Я продемонстрирую работу операций сложения и умножения. Ось Ox - количество операций, ось O(y) - время в секундах.



Сверху представлен график тестрирования операции сложения, а снизу умножения.

Выводы

В некоторых языках программирования длинная арифметика может быть предусмотрена самим языком или реализована в виде отдельной библиотеки. В рамках данной лабораторной я реализовал простейший калькулятор, работающий с числа типа BigInt, который однако далеко не идеальный. Его можно улучшить, добавив, например, возможность работать с отрицательными числами или усовершенствовать некоторые алгоритмы. Так умножение было реализовано без использования алгоритма Карацубы, который гораздо быстрее работает с очень большими числами.