Лабораторная работа № 6 по курсу: Дискретный анализ

Выполнил студент группы М8О-208Б-17 МАИ *Милько Павел.*

Задача

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 (^)
- 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 (=)

Метод решения

Необходимо написать реализацию простейших арифметических и логических операций с длинными числами. В качестве внутреннего представления числа логично выбрать вектор, в который будут добавляться "разряды" длинного числа. Числа в векторе располагаются от младшего разряда к старшему, максимальное значение числа в одном разряде ограничено выбранным основанием системы счисления, я выбрал 10^4.

Сложение *Длина результата максимум на 1 больше наибольшей из длин чисел.*

Реализуется тривиально: начиная с младших разрядов числа (начало вектора), суммируем оба числа. Если результат больше чем основание системы счисления, то записывается остаток от деления результата на основание системы счисления. При этом целая часть прибавляется к старшему разряду.

Вычитание *Длина результата не больше длины вычитаемого числа.*
Реализуется аналогично сложению: из большего вычитается меньшее, если при вычитании разрядов получается отрицательное число, то к нему прибавляется основание системы счисления и занимается единица из старшего разряда.

Умножение *Длина результата не больше суммы длин множителей.*
Алгоритм вычисления такой же, как и для обычных чисел (по разрядам, столбиком), за исключением того случая, когда результат становится больше основания системы счисления. Тогда целую часть от деления результата надо прибавить к следующему результату, а остаток от деления прибавить к разряду с номером, равным сумме позиций умножаемых разрядов двух чисел.

Деление *Длина результата не больше длины делимого числа.*
Осуществляется уголком: выбираем количество старших разрядов делимого числа так, чтобы получившийся срез по длине был равен делителю. Затем находим с помощью бинарного поиска (на отрезке от 0 до основания системы счисления) максимально возможный множитель,такой, что разница между срезом и умноженным делителем минимальна и положительна (0 тоже допустим, это означает что срез меньше делителя). Разницу запоминаем для дальнейшего деления, множитель записываем в старший разряд ответа. В начало остатка от деления записываем следующий старший разряд делителя и продолжаем алгоритм, пока не дойдём до младшего разряда делителя.

Возведение в степень *Длина результата изначально неизвестна.*
Использован алгоритм быстрого возведения в степень, который намного производительнее простого умножения.

Операции сравнения Реализованы поразрядным сравнением элементов и длин.

Исходный код

BigInt.hpp

numbersep

```
#ifndef BIGINT
#define BIGINT

#include <cmath>
#include <cstring>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

const int CELL_LENGTH = 4;
const int CELL_MAX = pow(10, CELL_LENGTH);
static int zero = 0;
class TBigInt {
private:
    std::vector<int> num;
    int& operator[](size_t pos)
    {
        if (num.size() <= pos)
            return zero;
        else
            return num[pos];
    }
    TBigInt& FilterZeros()
    {
        while (num.size() > 1 && !num.back())
            num.pop_back();
        return *this;
    }
    size_t size() const { return num.size(); }

    int cmp(TBigInt& Number);

public:
    TBigInt()
        : num({ 0 })
    {
    }
    TBigInt(std::string);
    TBigInt(size_t n)
        : num(n, 0U)
    {
    }
    ~TBigInt(){};

    TBigInt operator+(TBigInt&);
    TBigInt operator-(TBigInt&);
```

```
    TBigInt operator*(TBigInt&);
    TBigInt operator/(TBigInt&);
    TBigInt operator^(TBigInt);
    bool operator==(TBigInt& Number) { return cmp(Number) == 0; }
    bool operator<(TBigInt& Number) { return cmp(Number) == -1; }
    bool operator>(TBigInt& Number) { return cmp(Number) == 1; }
    friend std::ostream& operator<<(std::ostream&, TBigInt&);
};

#endif
```

BigInt.cpp

numbersep

```
#include "BigInt.hpp"

TBigInt::TBigInt(std::string inp)
    : num({})
{
    if (inp.empty())
        return;
    size_t pos = 0;
    while (inp[pos] == '0')
        ++pos;
    long long start, end = static_cast<long long>(inp.size());
    for (start = end - CELL_LENGTH; start >= pos && end > 0; start -=
        CELL_LENGTH) {
        if (start < 0)
            start = 0;
        num.push_back(atoi(inp.substr(static_cast<size_t>(start), static_cast<
            size_t>(end - start)).c_str()));
        end -= CELL_LENGTH;
    }
}

TBigInt TBigInt::operator+(TBigInt& Number)
{
    TBigInt res(std::max(size(), Number.size()));
    int r = 0;
    for (size_t i = 0; i < res.size(); ++i) {
        res[i] = (*this)[i] + Number[i] + r;
        if (r = res[i] >= CELL_MAX, r)
            res[i] -= CELL_MAX;
    }
    if (r)
        res.num.push_back(r);
    return res;
}

TBigInt TBigInt::operator-(TBigInt& Number)
{
    TBigInt res(size());
```

```
    int r = 0;
    for (size_t i = 0; i < res.size(); ++i) {
        res[i] = (*this)[i] - Number[i] - r;
        if (r = res[i] < 0, r) {
            res[i] += CELL_MAX;
        }
    }
    if (r)
        res.num.push_back(r);
    return res.FilterZeros();
}

TBigInt TBigInt::operator*(TBigInt& Number)
{
    TBigInt res(size() + Number.size());
    int r;
    for (size_t i = 0; i < size(); ++i) {
        if (!(*this)[i])
            continue;
        r = 0;
        for (size_t j = 0; j < Number.size() || r; ++j) {
            res[i + j] += (*this)[i] * Number[j] + r;
            if (r = res[i + j] / CELL_MAX, r)
                res[i + j] -= r * CELL_MAX;
        }
    }
    return res.FilterZeros();
}

int TBigInt::cmp(TBigInt& Number)
{
    if (size() != Number.size())
        return (size() > Number.size()) ? 1 : -1;
    for (size_t i = size() - 1; i + 1 != 0; --i)
        if ((*this)[i] != Number[i])
            return ((*this)[i] > Number[i]) ? 1 : -1;
    return 0;
}

TBigInt TBigInt::operator/(TBigInt& Number)
{
    TBigInt res(size()), cv, help;
    for (size_t i = size() - 1; i + 1 != 0; --i) {
        cv.num.insert(cv.num.begin(), (*this)[i]);
        if (!cv.num.back())
            cv.num.pop_back();
        int l, r, x;
        l = x = 0;
        r = CELL_MAX;
        while (l <= r) {
            int m = (l + r) / 2;
```

```
            help = TBigInt(std::to_string(m)) * Number;
            if (help < cv || help == cv) {
                x = m;
                l = m + 1;
            } else
                r = m - 1;
        }
        res[i] = x;
        help = TBigInt(std::to_string(x)) * Number;
        cv = cv - help;
    }
    return res.FilterZeros();
}

TBigInt TBigInt::operator^(TBigInt Number)
{
    TBigInt res, BigZero, two(std::to_string(2));
    res.num[0] = 1;
    while (!(Number == BigZero)) {
        if (Number.num.front() % 2)
            res = res * (*this);
        (*this) = (*this) * (*this);
        Number = Number / two;
    }
    return res;
}

std::ostream& operator<<(std::ostream& os, TBigInt& num)
{
    size_t n = num.size();
    if (!n)
        return os;
    os << num[n - 1];
    for (size_t i = n - 2; i + 1 != 0; --i)
        os << std::setfill('0') << std::setw(CELL_LENGTH) << num[i];

    return os;
}
```

main.cpp

numbersep

```
#include "BigInt.hpp"

int main()
{
    std::string num1, num2;
    char sym;
    TBigInt BigZero, res;

    while (std::cin >> num1 >> num2 >> sym) {
        TBigInt a(num1);
```

```
    TBigInt b(num2);

    switch (sym) {
    case '+':
        res = a + b;
        std::cout << res << '\n';
        break;
    case '-':
        if (a < b) {
            std::cout << "Error\n";
            break;
        }
        res = a - b;
        std::cout << res << '\n';
        break;
    case '*':
        res = a * b;
        std::cout << res << '\n';
        break;
    case '^':
        if (num1 == "0") {
            if (num2 == "0")
                std::cout << "Error" << std::endl;
            else
                std::cout << "0" << std::endl;
            break;
        }
        res = a ^ b;
        std::cout << res << '\n';
        break;
    case '/':
        if (b == BigZero) {
            std::cout << "Error\n";
            break;
        }
        res = a / b;
        std::cout << res << '\n';
        break;
    case '<':
        std::cout << ((a < b) ? "true" : "false") << '\n';
        break;
    case '>':
        std::cout << ((a > b) ? "true" : "false") << '\n';
        break;
    case '=':
        std::cout << ((a == b) ? "true" : "false") << '\n';
        break;
    default:
        break;
    }
}
```

```
    return 0;
}
```

Генератор тестов

```
     numbersep
#!/usr/bin/env python

import sys
import random

tests = int(sys.argv[1]) if len(sys.argv) > 1 else 10000

operators = {'+': lambda a, b: a + b,
             '-': lambda a, b: a - b,
             '*': lambda a, b: a * b,
             '^': lambda a, b: a ** b,
             '/': lambda a, b: a // b}

IN = open("input", "w")
OUT = open("output", "w")

for i in range(tests):
    operator = random.choice(list(operators.keys()))
    a = random.randint(0, 2000 if operator == '^' else 10 ** 35)
    b = random.randint(0, 2000 if operator == '^' else 10 ** 35)

    if (operator == '/' and b == 0) or (operator is '-' and a < b) or (
        operator == '^' and a == b == 0):
        res = "Error"
    else:
        res = operators[operator](a, b)

    print(a, b, operator, sep='\n', file=IN)
    print(res, end='\n', file=OUT)

IN.close()
OUT.close()
```

Выводы

Длинная арифметика встречается довольно часто (в язык питон она даже встроена изначально) и полезно понимать, как происходят базовые операции с числами, которые не помещаются в типы с фиксированной длиной.

Моя реализация длинной арифметики далека от идеала, например в качестве основания системы счисления можно было выбрать степень двойки. Это позволило бы заменить операции умножения и деления (при оперировании отдельными разрядами)

на битовую конъюнкцию и битовый сдвиг соответственно. Так же в моей реализации не рассматриваются знаковые числа, а уж про дробную часть и говорить нечего.

В целом лабораторная работа была интересной и несложной. Но что-то мне подсказывает, что дальше будет намного сложнее и больнее.