

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу «Дискретный анализ»

Студент: И. О. Ильин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №6

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 возведения 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.

1 Описание

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

Для сложения и вычитания воспользуемся 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 сравнению разрядов, начиная со старшего за $O(n)$.

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

Также реализуем бинарное возведение в степень, тогда понадобится $O(\log_2 exp)$ умножений, а не $O(exp)$ в случае наивного алгоритма, exp — показатель степени.

2 Исходный код

Сначала, объявим класс *TBiggestInt*, 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.

```
1 namespace NBiggestInt {
2
3 class TBiggestInt {
4     public:
5         static const int BASE = 1e6;
6         static const int RADIX = 6;
7
8         TBiggestInt() = default;
9         TBiggestInt(const std::string & str);
10        TBiggestInt(const size_t & length, const long long value = 0);
11        void Initialize(const std::string & str);
12        std::string GetString() const;
13
14        size_t Size() const;
15        TBiggestInt Pow(const TBiggestInt & degree) const;
16        void Shift(const long long degree);
17        static TBiggestInt KaratsubaMultiplication(TBiggestInt && lhs, TBiggestInt &&
18            rhs);
19
20        TBiggestInt & operator=(const TBiggestInt & rhs);
21
22        TBiggestInt operator+(const TBiggestInt & rhs) const;
23        TBiggestInt operator-(const TBiggestInt & rhs) const;
24        TBiggestInt operator*(const TBiggestInt & rhs) const;
25        TBiggestInt operator/(const TBiggestInt & rhs) const;
26
27        TBiggestInt operator-(const long long rhs) const;
28        TBiggestInt operator*(const long long rhs) const;
29        TBiggestInt operator/(const long long rhs) const;
30        long long operator%(const long long rhs) const;
31
32        bool operator< (const TBiggestInt & rhs) const;
33        bool operator<=(const TBiggestInt & rhs) const;
34        bool operator> (const TBiggestInt & rhs) const;
35        bool operator==(const TBiggestInt & rhs) const;
36        bool operator==(const long long rhs) const;
37        bool operator> (const long long rhs) const;
38
39        friend std::istream& operator>>(std::istream &is, TBiggestInt & rhs);
40        friend std::ostream& operator<<(std::ostream &os, const TBiggestInt & rhs);
41    private:
```

```

41 |         std::vector<long long> digits;
42 |
43 |         void DeleteLeadingZeros();
44 |     };
45 |
46 | } // namespace NBiggestInt

```

Напишем реализации методов инициализации от строки, получения размера, удаления ведущих нулей и два конструктора с оператором копирования.

```

1 | TBiggestInt::TBiggestInt(const std::string & str) {
2 |     this->Initialize(str);
3 | }
4 |
5 | TBiggestInt::TBiggestInt(const size_t & length, const long long value)
6 |     : digits(length, value)
7 | {}
8 |
9 | void TBiggestInt::Initialize(const std::string & str) {
10 |     long long startIdx = 0;
11 |     while (startIdx < str.size() && str[startIdx] == '0') {
12 |         ++startIdx;
13 |     }
14 |     if (startIdx == str.size()) {
15 |         digits.push_back(0);
16 |         return;
17 |     }
18 |
19 |     digits.clear();
20 |     size_t digitsSize = (str.size() - startIdx) / RADIX;
21 |     if ((str.size() - startIdx) % RADIX != 0) {
22 |         ++digitsSize;
23 |     }
24 |     digits.resize(digitsSize);
25 |
26 |     size_t digitsCount = 0;
27 |     for (long long i = str.size() - 1; i >= startIdx; i -= RADIX) {
28 |         long long currDigit = 0;
29 |         long long digitStart = i - RADIX + 1;
30 |         if (digitStart < 0 || digitStart <= startIdx) {
31 |             digitStart = 0;
32 |         }
33 |         for (long long j = digitStart; j <= i; ++j) {
34 |             currDigit = currDigit * 10 + str[j] - '0';
35 |         }
36 |
37 |         digits[digitsCount] = currDigit;
38 |         ++digitsCount;
39 |     }
40 | }

```

```

41 |
42 | size_t TBiggestInt::Size() const {
43 |     return digits.size();
44 | }
45 |
46 | TBiggestInt& TBiggestInt::operator=(const TBiggestInt & rhs) {
47 |     digits.resize(rhs.Size());
48 |     for (size_t i = 0; i < Size(); ++i) {
49 |         digits[i] = rhs.digits[i];
50 |     }
51 |     return *this;
52 | }
53 |
54 | void TBiggestInt::DeleteLeadingZeros() {
55 |     while (Size() > 1 && digits.back() == 0) {
56 |         digits.pop_back();
57 |     }
58 | }

```

Далее, реализуем перегруженные операторы ввода и вывода, а также метод получения строкового представления (пригодился при написании юнит тестов на корректное внутреннее представление числа).

```

1 | std::istream& operator>>(std::istream & is, TBiggestInt & rhs) {
2 |     std::string str;
3 |     is >> str;
4 |     rhs.Initialize(str);
5 |     return is;
6 | }
7 |
8 | std::ostream& operator<<(std::ostream & os, const TBiggestInt & rhs) {
9 |     os << rhs.digits[rhs.Size()- 1];
10 |     for (long long i = rhs.Size() - 2; i >= 0; --i) {
11 |         os << std::setfill('0') << std::setw(TBiggestInt::RADIX) << rhs.digits[i];
12 |     }
13 |     return os;
14 | }
15 |
16 | std::string TBiggestInt::GetString() const {
17 |     std::stringstream ss;
18 |     ss << *this;
19 |     return ss.str();
20 | }

```

Перейдём к реализации основных арифметических операций. Сложение и вычитание производим столбиком, отслеживая сколько должны прибавить или вычесть из следующего разряда.

```

1 | TBiggestInt TBiggestInt::operator+(const TBiggestInt & rhs) const {
2 |     size_t resSize = std::max(rhs.Size(), Size());

```

```

3   TBiggestInt res(resSize);
4   long long carry = 0;
5
6   for (size_t i = 0; i < resSize; ++i) {
7       long long sum = carry;
8       if (i < rhs.Size()) {
9           sum += rhs.digits[i];
10      }
11      if (i < Size()) {
12          sum += digits[i];
13      }
14      carry = sum / BASE;
15      res.digits[i] = sum % BASE;
16  }
17  if (carry != 0) {
18      res.digits.push_back(carry);
19  }
20  res.DeleteLeadingZeros();
21  return res;
22 }
23
24 TBiggestInt TBiggestInt::operator-(const TBiggestInt & rhs) const {
25     if (*this < rhs) {
26         throw std::logic_error("Error: trying to subtract bigger number from smaller");
27     }
28
29     size_t resSize = std::max(rhs.Size(), Size());
30     long long carry = 0;
31     TBiggestInt res(resSize);
32
33     for (size_t i = 0; i < resSize; ++i) {
34         long long diff = digits[i] - carry;
35         if (i < rhs.Size()) {
36             diff -= rhs.digits[i];
37         }
38
39         if (diff < 0) {
40             carry = 1;
41             diff += BASE;
42         } else {
43             carry = 0;
44         }
45         res.digits[i] = diff % BASE;
46     }
47     res.DeleteLeadingZeros();
48     return res;
49 }

```


Напишем методы умножения и деления. Умножение производим в столбик, не забывая о переполнении разряда. Делим тоже в столбик, используя бинарный поиск для поиска промежуточного результата на каждой итерации.

```

1 TBiggestInt TBiggestInt::operator*(const TBiggestInt & rhs) const {
2     TBiggestInt res(Size() + rhs.Size());
3     for (size_t i = 0; i < Size(); ++i) {
4         long long carry = 0;
5         for (size_t j = 0; j < rhs.Size() || carry > 0; ++j) {
6             long long current = res.digits[i + j] + carry;
7             if (j < rhs.Size()) {
8                 current += digits[i] * rhs.digits[j];
9             }
10            res.digits[i + j] = current % BASE;
11            carry = current / BASE;
12        }
13    }
14    res.DeleteLeadingZeros();
15    return res;
16 }
17
18 TBiggestInt TBiggestInt::operator/(const TBiggestInt & rhs) const {
19     if (rhs == 0) {
20         throw std::logic_error("Error: Trying to divide by zero");
21     }
22
23     TBiggestInt res;
24     TBiggestInt curr;
25     for (size_t i = Size() - 1; i < Size(); --i) {
26         curr.digits.insert(std::begin(curr.digits), digits[i]);
27         curr.DeleteLeadingZeros();
28
29         long long currResDigit = 0;
30         long long leftBound = 0;
31         long long rightBound = BASE;
32         while (leftBound <= rightBound) {
33             long long middle = (leftBound + rightBound) / 2;
34             TBiggestInt tmp = rhs * middle;
35             if (tmp <= curr) {
36                 currResDigit = middle;
37                 leftBound = middle + 1;
38             }
39             else {
40                 rightBound = middle - 1;
41             }
42         }
43
44         res.digits.insert(std::begin(res.digits), currResDigit);
45         curr = curr - rhs * currResDigit;
46     }

```

```

47 |
48 |     res.DeleteLeadingZeros();
49 |     return res;
50 |
51 | }

```

Реализуем алгоритм умножения Карацубы. Для этого, реализуем функцию сдвига числа влево. Затем, делаем числа одинаковой чётной длины, путём добавления ведущих нулей, разделяем их на 2 части и производим рекурсивное вычисления произведений с последующими сдвигами, складывая получаем результат.

```

1 | void TBiggestInt::Shift(const long long degree) {
2 |     if (*this == 0) {
3 |         return;
4 |     }
5 |
6 |     size_t oldSize = digits.size();
7 |     digits.resize(digits.size() + degree);
8 |
9 |     for (long long i = oldSize - 1; i >= 0; --i) {
10 |         digits[i + degree] = digits[i];
11 |     }
12 |     for (long long i = 0; i < degree; ++i) {
13 |         digits[i] = 0;
14 |     }
15 | }
16 |
17 | TBiggestInt TBiggestInt::KaratsubaMultiplication(TBiggestInt && lhs, TBiggestInt &&
18 |     rhs) {
19 |     size_t length = std::max(lhs.Size(), rhs.Size());
20 |     if (length == 1) {
21 |         return lhs * rhs.digits.back();
22 |     }
23 |     if (length % 2 != 0) {
24 |         ++length;
25 |     }
26 |
27 |     lhs.digits.resize(length);
28 |     rhs.digits.resize(length);
29 |
30 |     TBiggestInt leftHalfLhs;
31 |     TBiggestInt rightHalfLhs;
32 |     std::copy(std::begin(lhs.digits), std::begin(lhs.digits) + (length / 2), std:::
33 |         back_inserter(rightHalfLhs.digits));
34 |     std::copy(std::begin(lhs.digits) + (length / 2), std::end(lhs.digits) , std:::
35 |         back_inserter(leftHalfLhs.digits));
36 |
37 |     TBiggestInt leftHalfRhs;
38 |     TBiggestInt rightHalfRhs;

```

```

37 | std::copy(std::begin(rhs.digits), std::begin(rhs.digits) + (length / 2), std::
    | back_inserter(rightHalfRhs.digits));
38 | std::copy(std::begin(rhs.digits) + (length / 2), std::end(rhs.digits) , std::
    | back_inserter(leftHalfRhs.digits));
39 |
40 | // res = BASE^n * Prod1 + BASE^(n/2)*(Prod2 - Prod1 - Prod3) + Prod 3
41 | TBiggestInt Term1 = KaratsubaMultiplication(std::move(leftHalfLhs), std::move(
    | leftHalfRhs));
42 | TBiggestInt Term2 = KaratsubaMultiplication(leftHalfLhs + rightHalfLhs, leftHalfRhs
    | + rightHalfRhs);
43 | TBiggestInt Term3 = KaratsubaMultiplication(std::move(rightHalfLhs), std::move(
    | rightHalfRhs));
44 |
45 | Term2 = Term2 - Term1 - Term3;
46 | Term1.Shift(length);
47 | Term2.Shift(length / 2);
48 |
49 | TBiggestInt result = Term1 + Term2 + Term3;
50 |
51 | long long carry = 0;
52 | for (long long i = 0; i < result.Size(); ++i) {
53 |     result.digits[i] += carry;
54 |     carry = result.digits[i] / BASE;
55 |     result.digits[i] %= BASE;
56 | }
57 |
58 | if (carry != 0) {
59 |     result.digits.push_back(carry);
60 | }
61 |
62 | return result;
63 | }

```

Затем, реализуем бинарное возведение длинного в степень длинного числа.

```

1 | TBiggestInt TBiggestInt::Pow(const TBiggestInt & degree) const {
2 |     if (*this == 0 && degree == 0) {
3 |         throw std::logic_error("Error: 0^0 is uncertain");
4 |     }
5 |
6 |     TBiggestInt res("1");
7 |     if (degree == 0) {
8 |         return res;
9 |     }
10 |
11 |     TBiggestInt curr = *this;
12 |     TBiggestInt currDegree = degree;
13 |     while (currDegree > 0) {
14 |         if (currDegree.digits.back() % 2 != 0) {
15 |             res = res * curr;

```

```

16     }
17
18     curr = curr * curr;
19     currDegree = currDegree / 2;
20 }
21 return res;
22 }

```

При возведении в степень и делении длинных чисел использовались операции над длинным и коротким числами. Реализуем соответствующие перегрузки операторов. Алгоритмы те же самые, только отсутствуют циклы по разрядам второго числа, поскольку оно является коротким. Операторы деления и взятия остатком отличаются только возвращаемым значением.

```

1 BiggestInt TBiggestInt::operator-(const long long rhs) const {
2     if (Size() == 1 && digits[0] < rhs) {
3         throw std::logic_error("Error: trying to subtract bigger number from smaller");
4     }
5
6     TBiggestInt res = *this;
7     int idx = 0;
8     res.digits[0] -= rhs;
9     while (res.digits[idx] < 0) {
10         --res.digits[idx + 1];
11         res.digits[idx++] += BASE;
12     }
13     res.DeleteLeadingZeros();
14     return res;
15 }
16
17 TBiggestInt TBiggestInt::operator*(const long long rhs) const {
18     TBiggestInt res(Size());
19     long long carry = 0;
20     for (size_t i = 0; i < Size() || carry > 0; ++i) {
21         long long currDigit = carry;
22         if (i == Size()) {
23             res.digits.push_back(0);
24         } else {
25             currDigit += digits[i] * rhs;
26         }
27         res.digits[i] = currDigit % BASE;
28         carry = currDigit / BASE;
29     }
30     res.DeleteLeadingZeros();
31     return res;
32 }
33
34 TBiggestInt TBiggestInt::operator/(const long long rhs) const {
35     TBiggestInt res(Size());

```

```

36     long long carry = 0;
37     for (size_t i = Size() - 1; i < Size(); --i) {
38         long long currDigit = carry * BASE + digits[i];
39         res.digits[i] = currDigit / rhs;
40         carry = currDigit % rhs;
41     }
42     res.DeleteLeadingZeros();
43     return res;
44 }
45
46 long long TBiggestInt::operator%(const long long rhs) const {
47     long long carry = 0;
48     for (size_t i = Size() - 1; i < Size(); --i) {
49         carry = ( carry * BASE + digits[i] ) % rhs;
50     }
51     return carry;
52 }

```

Реализуем операторы сравнения длинных чисел. Сначала сравниваем их длины, затем значения в разрядах, начиная со старшего.

```

1  bool TBiggestInt::operator< (const TBiggestInt & rhs) const {
2      if (Size() != rhs.Size()) {
3          return Size() < rhs.Size();
4      }
5
6      for (size_t i = Size() - 1; i < Size(); --i) {
7          if (digits[i] != rhs.digits[i]) {
8              return digits[i] < rhs.digits[i];
9          }
10     }
11
12     return false;
13 }
14
15 bool TBiggestInt::operator<= (const TBiggestInt & rhs) const {
16     return !(*this > rhs);
17 }
18
19 bool TBiggestInt::operator> (const TBiggestInt & rhs) const {
20     if (Size() != rhs.Size()) {
21         return Size() > rhs.Size();
22     }
23
24     for (size_t i = Size() - 1; i < Size(); --i) {
25         if (digits[i] != rhs.digits[i]) {
26             return digits[i] > rhs.digits[i];
27         }
28     }
29

```

```

30     return false;
31 }
32
33 bool TBiggestInt::operator==(const TBiggestInt & rhs) const {
34     if (Size() != rhs.Size()) {
35         return false;
36     }
37
38     for (size_t i = Size() - 1; i < Size(); --i) {
39         if (digits[i] != rhs.digits[i]) {
40             return false;
41         }
42     }
43
44     return true;
45 }

```

Также реализуем методы сравнения с коротким числом, чтобы избежать дублирование кода в других операциях.

```

1 bool TBiggestInt::operator==(const long long rhs) const {
2     if (Size() != 1) {
3         return false;
4     }
5     return digits.back() == rhs;
6 }
7
8 bool TBiggestInt::operator> (const long long rhs) const {
9     if (Size() > 1) {
10         return true;
11     }
12     return digits.back() > rhs;
13 }

```

3 Консоль

```
MacBook-Pro:da_lab_06 mr-ilin$ make clean
rm -f *.o solution debug.out benchmark unit_test
MacBook-Pro:da_lab_06 mr-ilin$ make
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -O3 -c main.cpp -o main.o
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -O3 -c biggest_int.cpp
-o biggest_int.o
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -O3 main.o biggest_int.o
-o solution
MacBook-Pro:da_lab_06 mr-ilin$ cat test.t
0
0
+
100145
4
*
11234
9876
>
MacBook-Pro:da_lab_06 mr-ilin$ ./solution <test.t
0
400580
true
MacBook-Pro:da_lab_06 mr-ilin$
```

4 Тест производительности

Тест производительности представляет из себя сравнение производительности написанного класса *TBiggestInt* с библиотекой *GMP*. С помощью *test_generator.py* происходит 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.

```
MacBook-Pro:da_lab_06 mr-ilin$ ./wrapper.sh 100 benchmark
[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/01.t...
=====
===== 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
```

```
MacBook-Pro:da_lab_06 mr-ilin$ ./wrapper.sh 1000 benchmark
[2021-03-31 21:20:16] [INFO] Compiling...
make: 'benchmark' is up to date.
[2021-03-31 21:20:16] [INFO] Generating tests (examples for each test=[1000])...
```



```
[2021-03-31 21:21:27] [INFO] Executing tests/01.t...
```

```
=====
```

```
===== ADDITION =====
```

```
TBiggestInt Time = 222 ms
```

```
GMP Time = 160 ms
```

```
===== SUBTRACTION =====
```

```
TBiggestInt Time = 100 ms
```

```
GMP Time = 89 ms
```

```
===== MULTIPLICATION =====
```

```
TBiggestInt Time = 342 ms
```

```
Karatsuba Time = 100439 ms
```

```
GMP Time = 228 ms
```

```
===== DIVISION =====
```

```
TBiggestInt Time = 28649 ms
```

```
GMP Time = 160 ms
```

```
===== POWER =====
```

```
TBiggestInt Time = 372708199 ms
```

```
GMP Time = 139997 ms
```

```
===== LESS THEN =====
```

```
TBiggestInt Time = 20 ms
```

```
GMP Time = 26 ms
```

```
===== EQUAL TO =====
```

```
TBiggestInt Time = 27 ms
```

```
GMP Time = 29 ms
```

```
=====
```

```
[2021-03-31 21:27:40] [INFO] No failed tests, hooray
```

Как видно, почти во всех случаях *GMP* справляется быстрее, кроме операций сравнения.

5 Выводы

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

Список литературы

- [1] *Длинная арифметика — e-maxx*.
URL: https://e-maxx.ru/algo/big_integer (дата обращения: 15.03.2021).
- [2] *Длинная арифметика — Алгоритмы на C++ (олимпиадный подход)*.
URL: <http://cppalgo.blogspot.com/2010/05/blog-post.html> (дата обращения: 15.03.2021).
- [3] *C++ reference*
URL: <https://en.cppreference.com> (дата обращения: 15.03.2021).