Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу «Дискретный анализ»

Студент: И. С. Глушатов
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
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 ($\wedge$).
- 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.

1 Описание

Длинная арифметика - это набор арифметических операций, такие как: сложение, вычитание, умножение, деление и т.д., выполняющихся над числами, разрядность которых превышает длину машинного слова. Эти операции выполняются программно с использованием стандартных аппаратных средств работы с числами меньших разрядов.

Для реализации длинных чисел я создал класс BigInt, полем которого является вектор, каждый элемент которого является разрядом в 10000 системе счисления. Конструктор принимает строку, которая в последствии парсит символьное представление числа в вектор разрядов.

```
1  namespace Calculater {
2      class BigInt {
3
4          static const int32_t BASE = 10000;
5          static const int32_t RADIX = 4;
6
7      private:
8          std::vector<int32_t> _data;
9
10     public:
11
12         BigInt() = default;
13
14         BigInt(const std::string& str) {
15             for (size_t i = str.size(); i > 0; i -= BigInt::RADIX) {
16                 if (i < BigInt::RADIX) {
17                     _data.push_back(atoi(str.substr(0, i).c_str()));
18                     break;
19                 } else {
20                     _data.push_back(atoi(str.substr(i - BigInt::RADIX, BigInt::RADIX).
                           c_str()));
21                 }
22             }
23
24             RemoveZeros();
25         }
26
27         BigInt(const int32_t num) {
28             _data.push_back(num % BigInt::BASE);
29             _data.push_back((num / BigInt::BASE) % BigInt::BASE);
30             _data.push_back((num / BigInt::BASE) / BigInt::BASE);
31
32             RemoveZeros();
33         }
34
35         void RemoveZeros() {
36             while (_data.size() > 1 && _data.back() == 0) {
37                 _data.pop_back();
38             }
39         }
40     }
41 }
```

Сложение и вычитание реализуются на подобии алгоритмов выполнения этих операций в столбик и имеют сложность $O(max\{n, m\})$, где n и m - число разрядов чисел.

```
BigInt operator+(const BigInt& a) const {
            BigInt res;
            int32_t carry = 0;
            size_t n = std::max(a._data.size(), _data.size());
            for (size_t i = 0; i < n; ++i) {
                int32_t sum = carry;
                if (i < a._data.size()) {
                    sum += a._data[i];
                }
                if (i < _data.size()) {
                    sum += _data[i];
                }
                carry = sum / BigInt::BASE;
                res._data.push_back(sum % BigInt::BASE);
            }
            if (carry != 0) {
                res._data.push_back(1);
            }
            res.RemoveZeros();
            return res;
        }

BigInt operator-(const BigInt& a) const {
            BigInt res;
            int32_t p, carry = 0;
            size_t n = std::max(a._data.size(), _data.size());
            for (size_t i = 0; i < n; ++i) {
                p = _data[i] - carry;
                carry = 0;

                if (i >= a._data.size()) {
                    if (p < 0) {
                        carry = 1;
                        p += BigInt::BASE;
                    }
                    res._data.push_back(p);
                    continue;
                }

                if (p < a._data[i]) {
                    carry = 1;
                    p += BigInt::BASE;
                }

                res._data.push_back(p - a._data[i]);
            }

            res.RemoveZeros();
            return res;
        }
```

Умножение тоже похоже на умножение в столбик, однако мы не будем складывать в конце все числа, полученные перемножением по разрядам, а будем сразу считать результат с учетом сдвигов при переполнении разрядов. Так же важно упомянуть, что для 10000 разрядности умножение двух разрядов не выйдет за пределы 32-битного типа int. Сложность наивного алгоритма умножения $O(n * m)$, что не очень хорошо, когда количество разрядов числа слишком велико, поэтому для более сложных случаев применяются алгоритмы Карацубы или Шёнхаге-Штрассена.

```
BigInt operator*(const BigInt& a) const {
          BigInt res;
          size_t n = a._data.size() + _data.size();
          res._data.resize(n);

          int32_t k = 0;
          int32_t r = 0;
          for (size_t i = 0; i < _data.size(); i++) {
              for (size_t j = 0; j < a._data.size(); j++) {
                  k = _data[i] * a._data[j] + res._data[i + j];
                  r = k / BigInt::BASE;
                  res._data[i + j + 1] = res._data[i + j + 1] + r;
                  res._data[i + j] = k % BigInt::BASE;
              }
          }

          res.RemoveZeros();
          return res;
      }

BigInt operator*(const int32_t& a) const {
          BigInt c(a);
          return c * (*this);
      }
```

Быстрое возведение в степень - алгоритм, учитывающий четность степени, позволяет возводить число со сложностью $O(logn)$, где n - количество перемножений, которые надо совершить.

```
bool Odd(const BigInt& a) const {
          return a._data[0] & 1;
      }

BigInt Pow(BigInt a, BigInt b) const {
          BigInt res(1);
          while (b > BigInt(0)) {
              if (Odd(b)) {
                  res = res * a;
              }
              a = a * a;
              b = b / BigInt(2);
          }
          return res;
      }
```

Деление является, пожалуй, самым сложным алгоритмом. Я реализовал деление столбиком. Его примерная сложность $O((n-m) * logBASE * (m))$, так как по реализации цикл проходит $n-m$ раз, внутри цикла бинарным поиском подбирается нужное значение разряда в пределах от 1 до 10000, и при каждом подборе мы производим умножение m на разряд, т.е. сложность умножения в данном случае будет $O(m)$.

```
BigInt operator/(const BigInt& a) const {
          BigInt res;

          if (*this == a) return BigInt(1);
          if (*this < a) return BigInt(0);

          if (_data.size() == a._data.size()) {
              return BigInt(FindFactor(*this, a));
          } else {
              BigInt cur;
              cur._data = {_data.begin() + (_data.size() - a._data.size()), _data.end
                  ()};
              int32_t inc = _data.size() - a._data.size();

              if (cur < a) {
                  cur._data.insert(cur._data.begin(), _data[inc - 1]);
                  inc--;
              }
              do {
                  int32_t num = FindFactor(cur, a);
                  res._data.insert(res._data.begin(), num);
                  cur = cur - (a * num);
                  if (inc == 0) {
                      break;
                  } else {
                      inc--;
                      cur._data.insert(cur._data.begin(), _data[inc]);
                  }
                  cur.RemoveZeros();
              } while(true);
          }

          return res;
      }
```

Операции сравнения реализуются очень просто, сначала сравниваются длины векторов, и в случае если они равны, то поразрядно. Поэтому их код я приводить не буду.

2 Тест производительности

Для тестов я использвал утилиту gnuplot для построения графиков зависимости времени операций от количества разрядности входных чисел. Так же для сравнения использовал библиотеку int_width для 128-битных чисел и chrono для замера времени.

```
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab6$ ./a.out
a = 753275733897352885583252657455685685,b = 723587383839

Сложение:
753275733897352885583253381043069524
Моя реализация: 2.2498e-05
Библиотека <int_width>: 0.00179459

Вычитание:
753275733897352885583251933868301846
Моя реализация: 4.602e-06
Библиотека <int_width>: 0.000723548

Деление:
104102939150326398948834 9
Моя реализация: 1.76e-05
Библиотека <int_width>: 0.000360422
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab6$
```

Из приведенных тестов видно, что операции сложения, вычитания и деления проходят быстрее. Далее я приведу графики зависимости времени выполнения операций от количества разрядов числа.

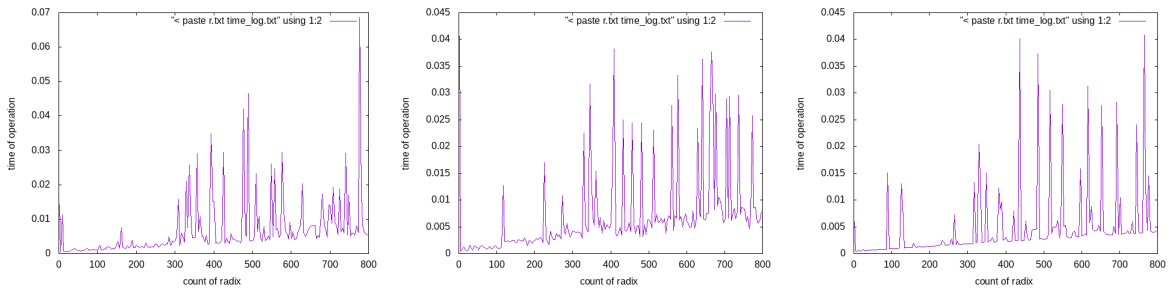

Рис. 1: Сложение, вычитание и сравнение

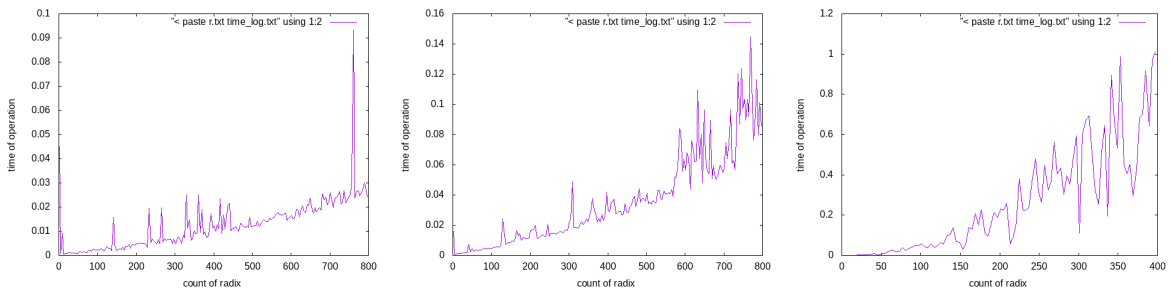

Рис. 2: Умножение, деление и возведение в степень

3 Выводы

В ходе шестой лабораторной работы я познакомился с длинной арифметикой. Реализовал класс BigInt и операции для работы с ним. Самым сложным алгоритмом оказалось деление. Сначала пришлось расписать весь код на листочке, попутно разбирая множество примеров и лишь затем программировать, зато таким образом я потратил очень мало времени на дебаг.

Список литературы

[1] *Поисковик - Google.*
URL: `https://www.google.com/`

[2] *Сайт с подробной документацией библиотек C++*
URL: `https://en.cppreference.com/`

[3] *Про длинную арифметику*
URL: `https://e-maxx.ru/algo/big_integer/`