Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу «Дискретный анализ»

Студентка: А. Довженко
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2018

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 возведении нуля в нулевую степень, программа должна вывести на экран строку Error.

Список условий:

- Больше (>)
- Меньше (<)
- Равно (=)

1 Описание

Требуется написать реализацию простейших арифметических операций для длинных чисел, таких как сложение, вычитание, умножение, деление, возведение в степень и операции сравнения. Идея реализации длинных чисел заключается в использовании вектора, элементами которого являются числа «короткие» (они играют роль цифр). Число в векторе расположено от младшего разряда к старшему. Размер чисел в векторе ограничен выбранным основанием системы счисления, в нашем случае 10^4.

Сложение осуществляется «столбиком». Пока оба числа не кончились, складываем числа. Начиная с младших разрядов, складываем «цифры», стоящие на данных позициях, прибавляем переносимый со сложения на прошлом шаге остаток. Записываем итоговую «цифру» в результирующий вектор. Если она больше выбранного основания системы счисления, то запоминаем новый остаток, уменьшаем ее на основание системы счисления.

Вычитание осуществляется «столбиком». Аналогично сложению, пока уменьшаемое число не кончилось, вычитаем из него второе число. Начиная с младших разрядов, вычитаем «цифру» и переносимый с вычитания на прошлом шаге «занятый разряд». Записываем итоговую «цифру» в результирующий вектор. Если она меньше 0, то запоминаем, что «заняли разряд», увеличиваем ее на основание системы счисления.

Умножение осуществляется по алгоритму Карацубы [3]. Алгоритм интуитивно понятен, но трудно описывается словами. Суть его состоит в том, что каждое из чисел можно представить в виде суммы их двух частей, половинок длиной $m = n/2$. Тогда
$A_0 = a_{m-1}a_{m-2}...a_0$
$A_1 = a_{n-1}a_{n-2}...a_m$
$A = A_0 + A_1 \cdot BASE^m$
$B_0 = b_{m-1}b_{m-2}...b_0$
$B_1 = b_{n-1}b_{n-2}...b_m$
$B = B_0 + B_1 \cdot BASE^m$
$A \cdot B = (A_0 + A_1 \cdot BASE^m) \cdot (B_0 + B_1 \cdot BASE^m) = A_0 \cdot B_0 + A_0 \cdot B_1 \cdot BASE^m + A_1 \cdot B_0 \cdot BASE^m + A_1 \cdot B_1 \cdot BASE^{2m} = A_0 \cdot B_0 + (A_0 \cdot B_1 + A_1 \cdot B_0) \cdot BASE^m + A_1 \cdot B_1 \cdot BASE^{2m}$
Здесь нужно 4 операции умножения. После несложных преобразований количество операций умножения можно свести к 3-м, заменив два умножения на одно и несколько операций сложения и вычитания, время выполнения которых на порядок меньше:
$A_0 \cdot B_1 + A_1 \cdot B_0 = (A_0 + A_1) \cdot (B_0 + B_1) - A_0 \cdot B_0 - A_1 \cdot B_1$
Окончательный вид выражения:

$$A \cdot B = A_0 \cdot B_0 + ((A_0 + A_1) \cdot (B_0 + B_1) - A_0 \cdot B_0 - A_1 \cdot B_1) \cdot BASE^m + A_1 \cdot B_1 \cdot BASE^{2m}$$

Деление осуществляется «уголком». Количество разрядов у частного от деления не превосходит количества разрядов у делимого. Т.е. начинаем формировать ответ со старшего разряда. На каждой итерации имеем текущее значение, которое пытаемся уменьшить на максимально большое количество раз делимым. Для отыскания этого числа используем бинарный поиск.

Возведение в степень производится быстрее, чем за «умножение * степень». Ускорение достигается благодаря свойствам степени, а именно $x^{2n} = x^{n2}$.

Операции сравнения осуществляются поразрядно.

2 Исходный код

Каждое число представляется вектором его цифр. Для лучшей производительности в качестве основания системы счисления взято 10000. Сравнение реализовано поразрядно. Многие функции опираются на удаление ведущих нулей. В остальном сложение и вычитание осуществляются столбиком, по определению. В первой версии программы умножение выполнялось также стобиком, но в следующей версии был использован алгоритм Карацубы. Возведение в степень опирается на свойство степени, которое обеспечивает сложность возведения в степень пропорционально не линии, а двоичному логарифму степени. Деление осуществляется по алгоритму деления уголком.

BigInt.c	
TBigInt operator+(const TBigInt&);	Сложение двух чисел
TBigInt operator-(const TBigInt&);	Вычитание двух чисел
TBigInt operator*(const TBigInt&) const;	Умножение двух чисел
TBigInt operator/(const TBigInt&);	Деление двух чисел
TBigInt Power(int r);	Возведение числа в степень
vll karatsubaMul(const vll &a, const vll &b) const;	Умножение двух чисел по алгоритму Карацубы
bool operator==(const TBigInt&) const;	Оператор «равно»
bool operator<(const TBigInt&) const;	Оператор «меньше»
bool operator>(const TBigInt&) const;	Оператор «больше»
bool operator<=(const TBigInt&) const;	Оператор «меньше или равно»
friend std::ostream& operator«(std::ostream&, const TBigInt&);	Вывод числа на стандартный поток вывода

```
1  class TBigInt
2  {
3  public:
4      TBigInt() {};
5      TBigInt(std::string&);
6      TBigInt(int n);
7      ~TBigInt() {};
8
9      TBigInt operator+(const TBigInt&);
10     TBigInt operator-(const TBigInt&);
11     TBigInt operator*(const TBigInt&) const;
12     TBigInt operator/(const TBigInt&);
13     TBigInt Power(int r);
14     vll karatsubaMul(const vll &a, const vll &b) const;
```

```
15      bool operator==(const TBigInt&) const;
16      bool operator<(const TBigInt&) const;
17      bool operator>(const TBigInt&) const;
18      bool operator<=(const TBigInt&) const;
19      friend std::ostream& operator<<(std::ostream&, const TBigInt&);
20
21  private:
22      void DeleteZeros();
23      std::vector<int> mData;
24  };
```

3 Консоль

```
karma@karma:~/mai_study/DA/lab6$ make
g++ -std=c++11 -O3 -pedantic -lm -o da6 BigInt.cpp main.cpp
karma@karma:~/mai_study/DA/lab6$ cat test
38943432983521435346436
354353254328383
+
904094384738493247293847 3843
2343543
-
972323
2173937
>
2
3
-
karma@karma:~/mai_study/DA/lab6$ cat test | ./da6
3894343333787468967 4819
904094384738493247293613 0300
false
Error
```

4 Тест производительности

Тест производительности разделим на 2 части: 1 часть – сравнение со встроенной длинной арифметикой в Python, 2 часть – посмотрим, что асимптотически программа работает корректно.

1 часть. Генерирую 3 файла, содержащие 10k, 100k и 375k операций соответственно. Тесты состоят из случайного набора всех возможных операций. Числа лежат в диапазоне от 0 до 10^{35}.

```
karma@karma:~/mai_study/DA/lab6$ cat test.sh
#!/bin/bash

for (( i=0; i<$1; ++i ))
do
python gen.py $2
./da6 <input >resC
diff resC output

python bench.py <input >resPy
diff resPy output
done
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 1.78267 sec.
10001d10000
<Python time: 1.154276 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 18.0493 sec.
100001d100000
<Python time: 12.639469 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 375000
375001d375000
<C++ time: 70.7233 sec.
375001d375000
<Python time: 43.146533 sec.
```

2 часть. Для каждой операции генерирую 3 файла, содержащие 10k, 100k и 375k этих операций. Числа лежат в диапазоне от 0 до 10^{35}. Для операции возведения в степень ограничиваю степень сверху двумя тысячами.

Операция сложения.

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 0.10694 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 1.16086 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 375000
375001d375000
<C++ time: 4.38787 sec.
```

Операция вычитания.

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 0.101635 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 1.12345 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 375000
375001d375000
<C++ time: 4.25694 sec.
```

Операция деления.

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 0.345723 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 3.29334 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 375000
375001d375000
<C++ time: 12.2874 sec.
```

Операция возведения в степень.

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 12.773 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 50000
50001d50000
<C++ time: 65.4794 sec.
```

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 132.038 sec.
```

Операция умножения. Т.к. во второй версии моей программы был реализован алгоритм Карацубы, дополнительно тестирую старую версию. Тестирование производится на одних и тех же тестовых файлах.

Версия программы с алгоритмом Карацубы.

```
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 10000
10001d10000
<C++ time: 0.137207 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 100000
100001d100000
<C++ time: 1.36454 sec.
karma@karma:~/mai_study/DA/lab6$ bash test.sh 1 375000
375001d375000
<C++ time: 5.35588 sec.
```

Версия программы без алгоритма Карацубы.

```
karma@karma:~/mai_study/DA/6first$ bash test.sh 1
10001d10000
<C++ time: 0.154738 sec.
karma@karma:~/mai_study/DA/6first$ bash test.sh 1
100001d100000
<C++ time: 1.20667 sec.
karma@karma:~/mai_study/DA/6first$ bash test.sh 1
375001d375000
<C++ time: 4.67595 sec.
```

Как можно заметить, моя реализация проигрывает встроенной длинной арифметике Python. Это неудивительно, наверняка создатели Python использовали различные оптимизации, которых нет в моей программе. Чтобы в этом убедиться, можно посмотреть исходный код Python (objects/longobject.c). К сожалению, мне мало что удалось в нем понять, но судя по обилию комментариев с описанием кастомных алгоритмов, разрабатывался этот код явно дольше двух недель.

При тестировании во второй части видим, что время работы для сложения и вычитания линейно. Деление и возведение в степень вполне соотносится с $O(n \cdot \log n)$ Отдельно нужно сказать про умножение. Умножение «столбиком» в моей реализации работает быстрее, чем умножение по алгоритму Карацубы. Хотя временная

сложность умножения «в столбик» должна быть $O(n^2)$, а умножения методом Карацубы $O(n^{\log 3})$. Могу предположить, что это связанно с тем, что у Карацубы большая константа и ускорение достигается только на действительно больших числах. В новой версии умножения приходится создавать буферные вектора, элементы которых смогли бы вместить промежуточные вычисления, и перезаписывать результирующий вектор.

5 Выводы

Лабораторная была относительно простой, для ее выполнения хватило знаний, полученных в начальной школе. В частности, выполнение арифметических операций «в столбик». Основная сложность заключалась в том, чтобы аккуратно это запрограммировать. Задача реализации длинной арифметики в простейшем виде тривиальна и неоднократно описана во многих источниках. Тем не менее не всегда очевидные вещи являются таковыми, поэтому не лишним было написать свою библиотеку, содержащую арифметические действия. Думаю, если бы на чекере стояли более строгие временные ограничения, задача сильно бы усложнилась.

Длинная арифметика много где применяется – используется для вычислений, требующих работы с большими числами и высокой точности. В настоящее время во многих языках программирования существуют встроенные модули для работы с длинными числами. Но, т.к. в C++ его нет, иногда требуется писать собственную библиотеку.

Из недостатков моей работы можно отметить отсутствие возможности работы с отрицательными и дробными числами. Но это и не требовалось в задании. В бенчмарке для более честного сравнения можно было бы использовать Boost.Multiprecision. Операция умножения требует значительного улучшения, если мы хотим достигнуть заявленной сложности алгоритма Карацубы.

Список литературы

[1] С. Дасгупта, Х. Пападимитриу, У. Вазирани. *Алгоритмы* — Издательство «МЦНМО», 2014. Перевод с английского: А. С. Куликова под редакцией А. Шеня. — 319 с. (ISBN 978-5-4439-0236-4 (рус.))

[2] *Длинная арифметика — e-maxx.*
URL: `http://e-maxx.ru/algo/big_integer` (дата обращения: 18.02.2018).

[3] *Алгоритм Карацубы — Википедия.*
URL: `https://en.wikipedia.org/wiki/Karatsuba_algorithm` (дата обращения: 28.02.2018).