Большие числа и операции с ними

Представление больших чисел

Для множества приложений предоставляемых процессором базовых типов вполне хватает. Однако, встречается много задач, исходные данные которых слишком велики. Число из 1000 цифр не поместится ни в один регистр. Поэтому комьютерное представление таких чисел и операции над ними приходится реализовывать самостоятельно.
При этом время выполнения внешнего алгоритма, использующего такие числа, очень сильно зависит от эффективности их реализации. Например, если оценка времени определяется $O(n^2)$ умножениями, то использование для этой операции в два раза более быстрого алгоритма дает ускорение в 4 раза. Поэтому в этой главе мы будем, пожалуй, наиболее серьезно заинтересованы не просто в правильных, но возможно более эффективных алгоритмах, которые при необходимости можно реализовать на приличном уровне.

Обычно, неотрицательное целое число N длины n представляется в виде

$$N = a_0 + a_1*BASE + a_2*BASE^2 + ... + a_{n-1} BASE^{n-1},$$

где BASE – основание системы счисления, все коэффициенты $0 \le a_i < BASE$.

Например, число 12345_{10} в этой интерпретации будет иметь вид

$$12345_{10} = 5 + 4*10 + 3*10^2 + 2*10^3 + 1*10^4 \ (BASE=10).$$

Длинное число хранится в массиве, где i-й элемент соответствует коэффициенту числа при $BASE^i$.
В качестве примера, рассмотрим массив для 12345_{10}: (5, 4, 3, 2, 1).
Как видно, цифры хранятся в обратном порядке. Это – некая "заготовка на будущее": дело в том, что реализации алгоритмов при этом имеют более естественный вид.

Такое представление N является частным случаем многочлена n-й степени $P(x) = a_0 + a_1*x + a_2*x^2 + ... + a_{n-1} x^{n-1}$, который также удобно хранить в виде массива коэффициентов. Поэтому большинство основных операций над числами при соответствующем упрощении алгоритмов работают для произвольных многочленов(сложение, умножение и т.п).

Знак числа, как и место десятичной точки, можно запомнить в отдельной переменной и учитывать при выполнении операций. Далее мы будем оперировать лишь с целыми неотрицательными числами.

Основание системы счисления BASE обычно зависит от максимального размера базового типа данных на компьютере, и выбирается, исходя из следующих соображений:

1. Основание должно подходить под один из базовых типов данных
2. BASE должно быть как можно больше, чтобы уменьшить размер представления длинного числа и увеличить скорость операций с ними, но достаточно малого размера, чтобы все операции с коэффициентами использовали базовый тип данных.
3. Для удобства можно выбрать BASE как степень 10 (вывод информации, отладка). BASE - степень двойки позволяет проводить быстрые операции на низком уровне.

В качестве разумного компромисса положим

```
#define BASE 10000
```

Это позволит нам понять функции в общем виде, не вдаваясь в тонкости битовых операций.

Число 20! = 243,2902,0081,7664,0000 представляется по этому основанию как
$20! = 0 + 7664*BASE + 81*BASE^2 + 2902*BASE^3 + 243*BASE^4$,
соответствующий массив коэффициентов:

Номер элемента массива	0	1	2	3	4
Значение	0	7664	81	2902	243

В примерах же для краткости изложения будем использовать десятичную систему, BASE=10.

Объявим класс длинного числа с простейшими операциями:

```
class BigInt {
public:
	// к этим членам можно закрыть доступ
	ulong Size, SizeMax;            // Size – текущая длина
	                                // SizeMax – максимальная длина
	short *Coef;                    // Массив коэффициентов

	// в этом случае здесь также должны быть описаны дружественные функции
	// операций над большими числами, которые будут разобраны ниже.
	BigInt();
	BigInt(ulong);
	BigInt(const BigInt&);
	virtual ~BigInt();

	void zero();                    // Обнулить число
	void update();

	BigInt& operator=(const BigInt&);
	operator long();                // Оператор преобразования BigInt к типу long
};
```

При объявления длинного числа используется один из трех конструкторов, уничтожение происходит через деструктор, с освобождением занимаемой памяти.

```
BigInt::BigInt() {
	SizeMax = Size = 0;             // Объявление вида BigInt A;
	Coef = NULL;                    // Создается полностью пустое число
}

BigInt::BigInt(ulong MaxLen) {
	Coef = new short[MaxLen];       // Объявление вида BigInt A(10);
	SizeMax = MaxLen;               // Выделяет память под MaxLen цифр
	Size = 0;
}


BigInt::BigInt(const BigInt &A) {       // Конструктор копирования
	SizeMax = A.SizeMax;            // Создает B, равное A
	Size = A.Size;
	Coef = new short[SizeMax];
	for(ulong i=0; i<SizeMax; i++) Coef[i] = A.Coef[i];
}

BigInt::~BigInt() {
	delete Coef;
}


void BigInt::zero() {                   // A.zero() – обнулить число
	for(ulong i=0; i<SizeMax; i++) Coef[i]=0;
	Size=1;
}
```

Оператор long вычисляет число в “обычном виде”

$$N = Coef[0] + Coef[1]BASE + \dots + Coef[n-1]\, BASE^{n-1},$$

он может быть весьма полезен при отладке, когда BASE = 10, а числа небольшие.

```
BigInt::operator long() {
	long tmp=0;  // при вычислениях может произойти переполнение
	for(ushort i=0; i<Size; i++) tmp += Coef[i]*(long)pow( BASE, (real)i);
```

```
	return tmp;
}
```

Несколько более интересен оператор присваивания. Для его правильной работы необходимо разобрать случай A=A и при необходимости выделить дополнительную память под коэффициенты.

```
inline BigInt& BigInt::operator=(const BigInt &A) {
	const short *a = A.Coef;
	if (this == &A) return *this;	// Если присваивание вида A=A - выйти
	if ( SizeMax < A.Size ) {	// Если размера не хватает – переинициализация
		if (Coef) delete Coef;
		Coef = new short[A.Size];
		SizeMax = Size = A.Size;
	} else Size = A.Size;

	for(ulong l=0; l<Size; l++)
		Coef[l] = a[l];

	return *this;
}
```

Возвращение this необходимо, чтобы работали присваивания вида A=B=C (они интерпретируются как A=(B=C)).

Операции над большими числами

Прежде всего определимся с интерфейсом. Можно определить операторы как части класса, и это будет достаточно удобно в использовании.

```
BigInt& operator+(BigInt&);
```

Однако, рассмотрим, что произойдет при вычислении C=A+B.
Оператор при всем желании не сможет получить доступ к числу C, поэтому придется создавать временное число Temp=A+B, которое затем будет скопировано в C оператором присваивания. Лишних операций и использования дополнительной памяти можно избежать, если записывать результат в C напрямую.

Поэтому в качестве интерфейса выберем внешние функции, которые принимают в качестве параметров исходные данные и число для записи результата. Если удобство стоит на первом месте – замена их на операторы не должна составить большого труда.

Сложение и вычитание

Практически каждый умеет складывать длинные числа, используя для этого листок бумаги и карандаш. Чем мы теперь займемся – это перенесем имеющееся у нас понимание на язык, понимаемый компьютером.

Схема для сложения очень проста: складываем цифры слева направо(цифры хранятся в обратном порядке). Если зафиксировано переполнение (т.е при сложении получена цифра, большая максимально возможной в данной системе счисления), то происходит “перенос”(carry) в следующий разряд.

```
// Вычисление C = A+B, работает вызов вида Add (A, B, A).
// максимальный размер C должен быть достаточен для хранения суммы
void Add(const BigInt &A, const BigInt &B, BigInt &C) {
	ulong i;
	long temp;	// temp здесь и далее играет роль “временной” цифры,
			// до выполнения переноса. Возможно, temp > BASE.

	// Здесь и в дальнейших примерах для быстрого доступа к коэффициентам
```

```
        // объявляются временные указатели a,b,c.
        const short *a=A.Coef, *b=B.Coef;
        short *c=C.Coef, carry = 0;
        // Добиваемся B.Size ≤ A.Size.
        if ( A.Size < B.Size ) {
                Add(B,A,C);
                return;
        }

        // Теперь B.Size ≤ A.Size.
        // Складываем два числа, i - номер текущей цифры
        for (i=0; i<B.Size; i++) {
                temp = a[i] + b[i] + carry;
                if (temp >= BASE) {              // переполнение. Перенести единицу.
                        c[i] = temp - BASE;
                        carry = 1;
                } else {
                        c[i] = temp;
                        carry = 0;
                }
        }

        // Меньшее число кончилось
        for (; i < A.Size; i++) {
                temp = a[i] + carry;
                if (temp >= BASE) {
                        c[i] = temp - BASE;
                        carry = 1;
                } else {
                        c[i] = temp;
                        carry = 0;
                }
        }
        // Если остался перенос – добавить его в дополнительный разряд
        if (carry) {
                c[i] = carry;
                C.Size = A.Size+1;
        }
        else C.Size = A.Size;
}
```

Вычитание осуществляется аналогично, с той лишь разницей, что осуществляется перенос “заимствования”. Мы работаем с положительными числами, поэтому если вычитаемое большое по размеру – происходит выход.
Если длины одинаковы, но одно больше другого - это приведет к тому, что в конце процесса останется неиспользованное заимствование, а результат будет дополнением до $BASE^{B.Size}$.
Например, при обычном вычитании 35 – 46 = -11, при нашем 35 – 46 = 89, так как выполняется равенство 89 = 100 – 11(89 дополнение 11 до 100).

```
// C = A-B, должно быть A.Size >= B.Size. Работает вызов Sub(A, B, A).
// Если длины равны, но A<B: возвращается -1, результат будет дополнением.
int Sub (const BigInt& A, const BigInt& B, BigInt& C) {
        const short *a=A.Coef, *b=B.Coef;
        short *c=C.Coef;
        ulong i;
        long temp, carry=0;
        if ( A.Size < B.Size ) error ("BigSub: size error");

        for (i=0; i<B.Size; i++) {
                temp = a[i] - b[i] + carry;
                if (temp < 0) {
                        c[i] = temp + BASE;
                        carry = -1;
                } else {
                        c[i] = temp;
                        carry = 0;
                }
        }
```

```
        for (; i<A.Size; i++) {
                temp = a[i] + carry;
                if (temp < 0) {
                        c[i] = temp + BASE;
                        carry = -1;
                } else {
                        c[i] = temp;
                        carry = 0;
                }
        }

        // Размер результата может быть гораздо меньше, чем у исходного числа
        // Устанавливаем его по первому положительному разряду
        i = A.Size-1;
        while ( (i>0) && (c[i]==0) ) i--;
        C.Size = i+1;
        return carry;
}
```

Умножение

В качестве “разминки” напишем функцию в случае, если один из множителей – обычное, короткое число. В этом случае происходит простое умножение на него каждой цифры.
При этом, конечно же, может произойти переполнение, поэтому необходимо вычислять переносы, которые при умножении могут принимать значения от 0 до BASE-1.

```
// C = A*B, работает вызов Smul (A, B, A).
void SMul(const BigInt &A, const short B, BigInt &C) {
        ulong i, temp;
        const short *a=A.Coef;
        short *c=C.Coef, carry=0;

        for (i=0; i<A.Size;i++) {
                temp = a[i]*B + carry;
                carry = temp / BASE;
                c[i] = temp - carry*BASE;              // c[i] = temp % BASE
        }

        if (carry) {
                // Число удлинилось за счет переноса нового разряда
                c[i] = carry;
                C.Size = A.Size+1;
        }
        else C.Size = A.Size;

}
```

Можно заметить, что для нахождения правильного значения цифры остаток по модулю BASE берется без использования оператора ‘%’. Причина в том, что процессор выполняет взятие остатка во много раз медленнее, чем вычитание с умножением. Если уже есть частное – ни к чему загружать его лишней работой в главном цикле.

Как быть, если оба числа – длинные ? Используемая нами в этом случае “бумажная” система нуждается в некоторой модификации.

```
      8 9 2 4
        5 6 7
  ___________
    6 2 4 6 8
  5 3 5 4 4
4 4 6 2 0
  ___________
5 0 5 9 9 0 8
```

Обычно мы умножаем число последовательно на одну цифру за другой, сохраняя временные результаты. Потом временные значения суммируются с учетом разряда.

Однако, можно ничего не хранить, а сразу прибавлять к окончательному результату.

УМНОЖЕНИЕ (A, B, результат C) {

1. Обнулить C
2. i = 0
3. Вычислить временный результат, соответствующий умножению i-ой цифры A на число B, в процессе вычисления сразу прибавлять его к C, начиная с i-ой позиции.
   Если получившаяся цифра C больше BASE – сделать перенос.
4. Если A не кончилось, i++ и идти на шаг 3

}

```
// C = A*B, не работает вызов Mul(A, B, A)
void Mul(const BigInt &A, const BigInt &B, BigInt &C) {
	ulong i, j;
	const short *a=A.Coef, *b=B.Coef;
	short *c=C.Coef;
	ulong temp, carry;

	// Обнулить необходимую для работы часть C
	for ( i=0; i <= A.Size+B.Size; i++ ) c[i]=0;

	for ( i = 0; i < A.Size; i++) {
		carry = 0;

		// вычисление временного результата с одновременным прибавлением
		// его к c[i+j] (делаются переносы)
		for (j = 0; j < B.Size; j++) {
			temp = a[i] * b[j] + c[i+j] + carry;
			carry = temp/BASE;
			c[i+j] = temp - carry*BASE;
		}
		c[i+j] = carry;
	}

	// Установить размер по первой ненулевой цифре
	i = A.Size + B.Size - 1;
	if ( c[i] == 0 ) i--;
	C.Size = i+1;
}
```

Алгоритм состоит A.Size циклов по всем цифрам B, значит, время работы оценивается произведением длин чисел – Θ(A.Size*B.Size).

Деление

Эта операция довольно часто является “камнем преткновения” для начинающего программиста. Однако, как мы увидим, ее выполнение следует “школьной” системе еще больше, чем умножение.

Пусть делитель – обычное число базового типа.

В этом случае деление выполняется очень просто: проходим по цифрам делимого, начиная со старшего разряда по направлению к младшему.
Для каждой пройденной цифры: разделить ее на B, целая часть результата добавляется в конец общего частного, остаток переносится и принимает участие в обработке следующей цифры…. И так до конца делимого.

Последний перенесенный остаток является остатком от всего деления.

```
_ 6 8 9 7 | 5
  5       |--------
  -       | 1 3 7 9
_ 1 8
  1 5
  ---
  _ 3 9
    3 5
    ---
      4 7
      4 5
      ---
        2
  ↑ ↑ ↑ ↑
i = 3 2 1 0
```

деление A=6897 на B=5

```
// Q = A/B , R = A%B,     A, Q - длинные числа, B, R - базового типа
void SDiv(const BigInt &A, const short B, BigInt &Q, short &R) {
       short r=0, *q=Q.Coef;
       const short *a=A.Coef;
       long i, temp;

       for ( i=A.Size-1; i>=0; i--) {   // идти по А, начиная от старшего разряда
             temp = r*BASE + a[i];      // r - остаток от предыдущего деления
                                        // вначале r=0, temp - текущая цифра A с
                                        // учетом перенесенного остатка

             q[i] = temp / B;           // i-я цифра частного

             r = temp - q[i]*B;         // остаток примет участие в вычислении
                                        // следующей цифры частного
       }

       R = r;

       // Размер частного меньше, либо равен размера делимого
       i = A.Size-1;
       while ( (i>0) && (q[i]==0) ) i--;
       Q.Size = i+1;

}
```

Что делать, если оба числа – длинные ? В этом случае школьный алгоритм несколько усложняется. Обозначим делитель $B=(b_0, b_1, \ldots, b_{n-1})$, делимое $A=(a_0, a_1, \ldots, a_{n+m-1})$, числа имеют длины n и $n+m$ соответственно.

Общий цикл остается почти таким же: алгоритм состоит из последовательно выполняемых шагов, где шаг представляет собой деление ($n+1$)-разрядной части А на n-разрядный делитель В(исключение может составлять первый шаг, но это укладывается в общую схему, которая будет обсуждаться ниже).

При этом i-й шаг состоит из двух действий:

1. Угадать i-ю цифру частного q[i]
2. Не создавая временных чисел, вычесть "сдвинутое" произведения B*q[i] из A.
   При этом сдвиг B относительно A на каждом шаге уменьшается.

Рассмотрим действия, совершаемые при делении A=68971 на B=513. Здесь n=5, m=2

```
 _ 6 8 9 7 1    | 5 1 3
   5 1 3        |--------
   -----          1 3 4
 _ 1 7 6 7
   1 5 3 9
   -------
   _ 2 2 8 1
     2 0 5 2
     -------
       2 2 9
```

горизонтальные линии проводятся между шагами

1. i = (индекс старшего коэффициента A) = 4
   a. Угадываем q[0] = 1
   b. Вычитаем сдвинутое B*1 = 513 из A
      (на бумаге мы при этом пишем только значимую для следующего шага часть A)
2. i = 3
   a. Угадываем q[1] = 3
   b. Вычитаем сдвинутое B*3 = 1539 из A
3. i = 2
   a. Угадываем q[3] = 4
   b. Вычитаем сдвинутое B*4 = 2052 из A
4. i < m = 2, процесс закончен. То, что осталось от A после вычитаний является остатком деления.

Вроде бы, все очевидно, за исключением одного "творческого" шага – угадывания.

Как заставить компьютер генерировать правильное частное *q* ? Или, хотя бы, достаточно близкое к нему число ?

Довольно интересный способ состоит в высказывании догадки qGuess по первым цифрам делителя и делимого. Понятно, что этих нескольких цифр недостаточно для гарантированно правильного результата, однако неплохое приближение все же получится.

Пусть очередной шаг представляет собой деление некоторого $U = (u_0, u_1, \ldots, u_n)$ на $B=(b_0, b_1, \ldots, b_{n-1})$. Если $b_{n-1} \geq$ BASE/2, то можно доказать следующие факты[1].

1. Если положить qGuess = (u_n*BASE + u_{n-1}) / b_{n-1} , то qGuess-2 ≤ q ≤ qGuess.
   Иначе говоря, вычисленная таким способом "догадка" будет не меньше искомого частного, но может быть больше на 1 или 2.

2. Если же дополнительно выполняется неравенство
   qGuess*b_{n-2} > BASE*r + u_{n-2} , где *r* – остаток при нахождении qGuess и qGuess ≠ BASE, (2)
   то qGuess-1 ≤ q ≤ qGuess, причем вероятность события qGuess = q+1 приблизительно равна 2/BASE.

Таким образом, если $b_{n-1} \geq$ BASE/2, то можно вычислить qGuess = (u_n*BASE + u_{n-1}) / b_{n-1} и уменьшать на единицу до тех пор, пока не станут выполняться условия (2). Получившееся значение будет либо правильным частным q, либо, с вероятностью 2/BASE, на единицу большим числом.

Что делать, если b_{n-1} слишком мало, чтобы пользоваться таким способом ?
Например, можно домножить делитель и делимое на одно и то же число scale = BASE / (b_{n-1} +1).
При этом несколько изменится способ вычисления остатка, а частное останется прежним. Такое домножение иногда называют *нормализацией* числа.

На тот случай, если qGuess получилось все же на единицу большим q, будем использовать в пункте (b) вычитание, аналогичное функции Sub, которое вместо отрицательного числа даст дополнение до следующей степени основания.
Если такое произошло, то последний перенос будет равен borrow = -1. Это сигнал, что необходимо прибавить одно B назад. Заметим, что в конце сложения будет лишний перенос carry=1, о котором нужно забыть (он компенсирует borrow=-1).

Например, делим A=505 на B=50. Угаданное частное qGuess=11.
505-550 = 955(=1000-45). Последний перенос borrow=-1, рельное частное q=qGuess-1=10.
Необходимо прибавить одно B: 955+50 = 5.
При сложении произошел перенос carry=1 из старшего разряда. Должно было получится 1005, но об этом переносе "забыли", так как он компенсировал borrow=-1.

```
// Q = A/B, R=A%B
void Div(const BigInt &A, BigInt &B, BigInt &Q, BigInt &R) {
	// Вырожденный случай 1. Делитель больше делимого.
	if ( A.Size < B.Size ) {
		Q.zero();
		R=A;
		return;
	}
	// Вырожденный случай 2. Делитель – число базового типа.
	if ( B.Size == 1) {
		SDiv ( A, B.Coef[0], Q, R.Coef[0]);
		R.Size = 1;
		return;
	}
```

[1] Доказательства не очень сложны, и если они интересны – они присутствуют в книге Д. Кнута [3].

```
// Создать временный массив U, равный A
// Максимальный размер U на цифру больше A, с учетом
// возможного удлинения A при нормализации
BigInt U(A.Size+1); U = A; U.Coef[A.Size]=0;

// Указатели для быстрого доступа
short *b=B.Coef, *u=U.Coef, *q=Q.Coef;

long n=B.Size, m=U.Size-B.Size;

long uJ, vJ, i;
long temp1, temp2, temp;

short scale;             // коэффициент нормализации

short qGuess, r;         // догадка для частного и соответствующий остаток
short borrow, carry;     // переносы

// Нормализация
scale = BASE / ( b[n-1] + 1 );
if (scale > 1) {
      SMul (U, scale, U);
      SMul (B, scale, B);
}

// Главный цикл шагов деления. Каждая итерация дает очередную цифру частного.
// vJ - текущий сдвиг B относительно U, используемый при вычитании,
// по совместительству - индекс очередной цифры частного.
// uJ – индекс текущей цифры U
for (vJ = m, uJ=n+vJ; vJ>=0; --vJ, --uJ) {

      qGuess = (u[uJ]*BASE + u[uJ-1]) / b[n-1];
      r = (u[uJ]*BASE + u[uJ-1]) % b[n-1];

      // Пока не будут выполнены условия (2) уменьшать частное.
      while ( r < BASE) {
            temp2 = b[n-2]*qGuess;
            temp1 = r*BASE + u[uJ-2];

            if ( (temp2 > temp1) || (qGuess==BASE) ) {
                  // условия не выполнены, уменьшить qGuess
                  // и досчитать новый остаток
                  --qGuess;
                  r += b[n-1];
            } else break;
      }

      // Теперь qGuess - правильное частное или на единицу больше q
      // Вычесть делитель B, умноженный на qGuess из делимого U,
      // начиная с позиции vJ+i

      carry = 0; borrow = 0;
      short *uShift = u + vJ;

      // цикл по цифрам B
      for (i=0; i<n;i++) {
            // получить в temp цифру произведения B*qGuess
            temp1 = b[i]*qGuess + carry;
            carry = temp1 / BASE;
            temp1 -= carry*BASE;

            // Сразу же вычесть из U
            temp2 = uShift[i] - temp1 + borrow;
            if (temp2 < 0) {
                  uShift[i] = temp2 + BASE;
                  borrow = -1;
            } else {
                  uShift[i] = temp2;
                  borrow = 0;
            }
      }
```

```
            // возможно, умноженое на qGuess число B удлинилось.
            // Если это так, то после умножения остался
            // неиспользованный перенос carry. Вычесть и его тоже.
            temp2 = uShift[i] - carry + borrow;
            if (temp2 < 0) {
                  uShift[i] = temp2 + BASE;
                  borrow = -1;
            } else {
                  uShift[i] = temp2;
                  borrow = 0;
            }


            // Прошло ли вычитание нормально ?
            if (borrow == 0) {          // Да, частное угадано правильно
                  q[vJ] = qGuess;

            } else {      // Нет, последний перенос при вычитании borrow = -1,
                          // значит, qGuess на единицу больше истинного частного
                  q[vJ] = qGuess-1;

                  // добавить одно, вычтенное сверх необходимого B к U
                  carry = 0;
                  for (i=0; i<n; i++) {
                        temp = uShift[i] + b[i] + carry;
                        if (temp >= BASE) {
                              uShift[i] = temp - BASE;
                              carry = 1;
                        } else {
                              uShift[i] = temp;
                              carry = 0;
                        }
                  }
                  uShift[i] = uShift[i] + carry - BASE;
            }

            // Обновим размер U, который после вычитания мог уменьшиться
            i = U.Size-1;
            while ( (i>0) && (u[i]==0) ) i--;
            U.Size = i+1;

      }
      // Деление завершено !

      // Размер частного равен m+1, но, возможно, первая цифра - ноль.
      while ( (m>0) && (q[m]==0) ) m--;
      Q.Size = m+1;

      // Если происходило домножение на нормализующий множитель –
      // разделить на него. То, что осталось от U – остаток.
      if (scale > 1) {
            short junk;         // почему-то остаток junk всегда будет равен нулю…
            SDiv ( B, scale, B, junk);
            SDiv ( U, scale, R, junk);
      } else R=U;
}
```

При делении числа длины *n* на число длины *m* производится Θ(n-m) проходов цикла, каждый из которых делает несколько операций стоимостью Θ (m). Таким образом, общая оценка будет квадратичной, как и у умножения: Θ (nm).

Печать длинного числа

Зачем нужны все расчеты, если результат нельзя напечатать ? С++ предоставляет удобную возможность перегрузки операторов ввода-вывода, которой мы сейчас и воспользуемся.
Для вывода при помощи этой функции необходимо, чтобы основание BASE было степенью 10, например, BASE = 10000 = 10^4. Тогда количество цифр в одном знаке числа будет равно этой степени, а реализация алгоритма – тривиальной.

Внешний цикл проходит коэффициенты один за другим, начиная со старшего. Внутренний цикл делит коэффициент на цифры и выдает их на печать.

Для удобства зададим количество цифр в коэффициенте заранее константой BASE_DIG.

```
#define BASE_DIG 4   // полагая BASE = 10000


ostream& operator<<(ostream& os, const BigInt& A) {
       long j, Digit=0;
       short Pow, Dec, Coef;

       os << A.Coef[A.Size-1];

       for (long i=A.Size-2; i>=0; i--) {      // Цикл вывода коэффициентов
              Pow = BASE/10;
              Coef = A.Coef[i];
              for (j=0; j<BASE_DIG; j++) {     // Цикл, выводящий каждый коэффициент
                     Dec = Coef/Pow;
                     Coef -= Dec*Pow;
                     Pow /= 10;          // Очередная цифра получается делением
                     os << Dec;          // коэффициента на 10^j
                     Digit++;
                     // Каждые 1000 цифр сопровождаются переходом строки
                     if (Digit%1000==0) os << "\n\n";
                     else if (Digit%50==0) os << "\t: " << Digit << "\n";
              }
       }
       return os;
}
```

Умножение с использованием БПФ

Сверткой векторов a и b называется вектор c с координатами $c_i = \sum_{k+l=i} a_k b_l$. Суммируются все произведения элементов, индексы которых в сумме равны i.

У коэффициентов свертки есть практический смысл – они дают результат умножения многочлена $a_0+a_1x^1+\ldots+a_{n-1}x^{n-1}$ на многочлен $b_0 + b_1x^1 + \ldots + b_{m-1}x^{m-1}$, где степени m, n – произвольные натуральные числа
.

$$(a_0+a_1x^1+\ldots+a_{n-1}x^{n-1})(b_0 + b_1x^1 + \ldots + b_{m-1}x^{m-1}) = c_0 + c_1x^1 + \ldots + c_{n+m-1}x^{n+m-1}$$

Числа в записи по основанию BASE являются многочленами, где в роли x выступает само основание, поэтому свертка может быть проинтерпретирована как результат умножения числа A= $a_0 + a_1$*BASE + a_2*BASE^2 + ... + a_{n-1} BASE^{n-1} на число B= $b_0 + b_1$*BASE + b_2*BASE^2 + ... + b_{m-1} BASE^{m-1} без вычисления переносов:

$$A \otimes B = c_0 + c_1\mathrm{BASE}^1 + \ldots + c_{n+m-1}\mathrm{BASE}^{n+m-1}$$

Конструкцию, которую образуют коэффициенты c_i , иногда называют *пирамидой умножения,* так как длина выражения для коэффициентов сначала растет, достигая максимума в середине вычислений, а затем падает, в конце обращаясь в ноль.
$c_0 = a_0 * b_0$,

$c_1 = a_0*b_1 + a_1*b_0$,
$c_2 = a_0*b_2 + a_1*b_1 + a_2*b_0$,
……………………………
$$c_i = \sum_{k+l=i} a_k b_l$$
……………………………
$c_{n+m-3} = a_{n-1}*b_{m-2} + a_{n-2}*b_{m-1}$
$c_{n+m-2} = a_{n-1}*b_{m-1}$
$c_{n+m-1} = 0$

При этом используемый базовый тип должен иметь достаточный объем для хранения коэффициентов порядка (n+m-1)*$BASE^2$, возникающих на вершине пирамиды.

Из всего вышесказанного следует, что для умножения больших чисел достаточно
1. Вычислить коэффициенты свертки c_i, i=0…n+m-1.
2. Сделать переносы, чтобы все коэффициенты были меньше BASE.

Шаг 2 выполняется довольно просто. Все переносы можно вычислить за O(n+m) шагов, двигаясь от младшего разряда к старшему.
```
for ( i=0; i ≤ n+m; i++) {
       carry = c[i] / BASE;       // Лишний разряд
       c[i+1] += carry;           // прибавить к следующей цифре
       c[i] -= carry * BASE;      // в c[i] оставить c[i]%BASE, что всегда меньше BASE
}
```

Таким образом, вся сложность умножения заключается в вычислении коэффициентов свертки на шаге 1. Для этого можно с успехом применить быстрое преобразование Фурье и быстрое преобразование Хартли.

Применение БПФ для вычисления свертки $a \otimes b$

Одно из главных свойств ДПФ состоит в выполнении *теоремы о свертке:* преобразование Фурье от свертки двух векторов есть скалярное произведение Фурье-образов этих векторов:

$$c = a \otimes b \iff \text{ДПФ}(c) = \text{ДПФ}(a) * \text{ДПФ}(b)$$

Отсюда следует, что $c = \text{ДПФ}^{-1}(\ \text{ДПФ}(a) * \text{ДПФ}(b)\ )$. Таким образом, весь алгоритм вычисления свертки состоит из 3 шагов:
1. Вычислить БПФ(a) и БПФ(b)
2. Скалярно перемножить получившиеся вектора
3. Вычислить обратное БПФ от скалярного произведения.

Перемножение многочленов сводится к скалярному произведению соответствующих векторов!

Предполагается, что на каждом шаге размеры векторов одинаковы и равны N: нельзя скалярно перемножить короткий вектор с более длинным, так что общее время имеет порядок O(N logN). Причем, наилучшего быстродействия достигают лишь варианты БПФ, работающие на векторах размера $N=2^k$, поэтому векторы по необходимости следует дополнить нулями.

В качестве примера возьмем A=(3,4), B=(5,4,3,2,1). Они хранятся “задом наперед”: старший разряд идет последним. Тогда псевдокод алгоритма будет таков:

БЫСТРОЕ_УМНОЖЕНИЕ {
1. Найти наименьшее число Len – степень двойки: Len ≥ A.Size + B.Size. Для рассматриваемых чисел Len=8.
2. Дополнить A и B ведущими незначащими нулями до Len. А нашем примере A=(3,4,0,0,0,0,0,0), B=(5,4,3,2,1,0,0,0).

3. Вычислить БПФ действительных векторов на обоих массивах цифр.
4. Скалярно перемножить преобразованные вектора, получив вектор размера Len.
5. Применить обратное преобразование Фурье, результатом которого будет свертка.
6. Преобразовать свертку в массив целых чисел, сделать переносы.

}

Цифры больших чисел хранятся в целочисленном формате. Поэтому для БПФ их необходимо скопировать во временные массивы типа с плавающей точкой. Если предполагается получать результат максимальной длины N, то необходимо выделить для них память как минимум размера MaxLen=2^k, где MaxLen – минимальная степень двойки, большая N. Например, если максимальный результат будет состоять из 1000 цифр по основанию BASE, то минимальный объем памяти MaxLen=1024, так как именно такой длины БПФ будет вычисляться.

```
real *LongNum1=NULL, *LongNum2=NULL;

// Инициализацию можно делать только 1 раз за всю программу.
void FastMulInit(ulong Len) {
       ulong MaxLen;
       if ((Len & -Len) == Len)   // Len = степень двойки
              MaxLen = Len;
       else {                     // иначе вычислить MaxLen – наименьшую степень 2,
              MaxLen = 2;         // такую что 2^MaxLen ≥ Len
              do MaxLen*=2; while (MaxLen < Len);
       }
       LongNum1 = new real[MaxLen];
       LongNum2 = new real[MaxLen];
}

// Деинициализация
void FastMulDeInit() {
       delete LongNum1;
       delete LongNum2;
}
```

Разобранная в соответствующем разделе функция RealFFT() производит преобразование “на месте”, возвращая результирующие векторы в сжатом виде.

a[0]	a[N/2]	a[1]	a[2]	…..	a[N/2-1]

b[0]	b[N/2]	b[1]	b[2]	…..	b[N/2-1]

Соответственно, функция скалярного произведения должна учитывать такой формат.

```
// Скалярное умножение комлексных векторов в сжатом виде: LongNum1 = LongNum1*LongNum2
void RealFFTScalar(real *LongNum1, const real *LongNum2, ulong Len) {
       Complex *CF1=(Complex*)LongNum1;
       const Complex *CF2=(Complex*)LongNum2;

       // первые два элемента- сгруппированные в одно комплексное действительные числа
       LongNum1[0] = LongNum1[0] * LongNum2[0];
       LongNum1[1] = LongNum1[1] * LongNum2[1];

       for (ulong x = 1; x < Len/2; x++)       // остальные – комплексные, как им и
              CF1[x] = CF1[x]*CF2[x];          // следует быть после ДПФ
}
```

Сделаем более подробный разбор последнего шага.
Все вычисления происходят в формате чисел с плавающей точкой, используют иррациональные числа, поэтому результат будет не набором целых чисел, а, скорее, приближением к нему. Для получения ответа каждую координату свертки необходимо округлить до ближайшего целого числа.

Проблема скрывается в том, что если точность вычислений недостаточно высока, то округление может произойти не к тому числу. В результате алгоритм благополучно завершится, но ответ будет неверен.
Часть вопросов, связанных с точностью, была решена в обсуждении БПФ. Однако даже при абсолютно точной тригонометрии будут накапливаться ошибки вычислений, так как операции арифметические операции не могут производиться с абсолютной точностью. Поэтому размер используемого типа данных должн быть достаточно большим, чтобы ошибка на любом знаке была меньше 0.5.

Например, при использовании типа данных размера 1, дробь 1/3 представляется в виде 0.3. При сложении трех дробей получаем
1/3 + 1/3 + 1/3 = (в формате чисел с плавающей точкой) 0.3 + 0.3 + 0.3 = 0.9
Если же размер – две цифры, то 1/3 = 0.33,
1/3 + 1/3 + 1/3 = 0.33 + 0.33 + 0.33 = 0.99. Точность вычислений сильно возросла.

Вообще говоря, путей увеличения точности два. Один из них связан с увеличением длины используемого типа: От float к double, далее к long double, потом к double double[2]…
Другой подход более гибок. Он предполагает при фиксированном типе уменьшать длину основания BASE. Таким образом число станет длиннее, будет занимать больше памяти, но за счет этого увеличивается точность.

Ограничения БПФ-умножения

Интересную оценку для ошибок дал Колин Персиваль [2].
Пусть требуется перемножить векторы из 2^n координат с использованием БПФ для векторов с действительными координатами. Тогда из его результатов следует, что погрешность err умножения x на y оценивается сверху выражением

$$err < 2^n \, BASE^2 \left(\varepsilon * 3n + \varepsilon\sqrt{5}\,(3n+4) + \beta(3n+3) \right) \qquad (2)$$

где ε - точность сложения/умножения, β – точность тригонометрических вычислений,
Отсюда при заданных ε, β не составляет труда найти минимально возможное основание BASE.
Например, при используемом типе double(53 бита), $\varepsilon=2^{-53}$. Ошибки тригонометрии ограничены величиной $\beta=\varepsilon/\sqrt{2}$.
Оценим верхнюю границу ошибок (2) числом ½. Приблизительно посчитав константы, получаем

$$2^n \, BASE^2 \, 2^{-53} \, (11.83\, n + 11.07) < ½$$

Для чисел длины 2^{20} это ведет к неравенству BASE < 4100. Такова оценка худшего случая, обоснованная математически.
На практике, однако, хорошим выбором будет BASE=10000. БПФ-умножение при таком основании может работать даже для много бОльших чисел. Однако, при этом не будет математических гарантий правильного результата.
При округлении следует смотреть на разницу между округляемым значением и результатом округления. Если она меньше 0.2, то умножение, скорее всего, дает правильный результат, если больше – рекомендуется уменьшить BASE или воспользоваться другим базовым типом для хранения коэффициентов.

После выполнения шага 5 нет готового произведения, а есть лишь свертка – результат без переносов. Как уже говорилось в при рассмотрении пирамиды умножения, значения коэффициентов свертки могут быть много больше основания, достигая $2N*BASE^2$. Если дополнительно вспомнить, что при обратном преобразовании Фурье происходит деление результатов выполнения функции RealFFT() на N , то максимальный размер цифры становится равен $2N^2*BASE^2$, поэтому следует позаботиться, чтобы не произошло переполнения. В частности, не следует объявлять BASE длиннее 4х десятичных цифр.

[2] последние два типа поддерживают далеко не все процессоры

В качестве резюме к вышесказанному, заметим, что проблемы всего три:

1. Точность тригонометрии
2. Точность при вычислении БПФ
3. Переполнение базового типа.

Первая проблема решена при обсуждении БПФ. Вторая и третья решаются путем уменьшения BASE, либо увеличения базового типа. При этом эффективность алгоритма падает, так как меньшее основание означает удлинение количества цифр, а больший базовый тип не всегда доступен.

Следующая функция преобразует свертку Convolution длины Len в большое число C, делая округления и выполняя переносы. В конце выполнения переменная MaxError будет содержать максимальную ошибку округления.
RealFFT() не производит нормализацию результата, поэтому ее необходимо сделать здесь же.

```
real MaxError;

void CarryNormalize(real *Convolution, ulong Len, BigInt &C) {
	real inv = 1.0 / (Len/2);         // коэффициент нормализации
	                                  // ДПФ проводилось над "комплексным" вектором
	                                  // в 2 раза меньшей длины

	real RawPyramid, Pyramid, PyramidError, Carry = 0;
	short *c = C.Coef;
	ulong x;

	// В C поместим только ту часть результата, которая туда влезает
	// ДПФ имеет длину, равную 2^k, но вектор коэффициентов
	// мог быть инициализован на меньшее количество элементов, не на степень 2.
	if ( Len > C.SizeMax ) Len=C.SizeMax;

	MaxError = 0;

	for (x = 0; x < Len; x++) {
		RawPyramid = Convolution[x] * inv + Carry;

		// Прибавляем 0.5, чтобы округление произошло к ближайшему целому
		Pyramid = floor(RawPyramid + 0.5);
		PyramidError = fabs(RawPyramid - Pyramid);

		if (PyramidError > MaxError)
			MaxError = PyramidError;

		Carry = floor(Pyramid / BASE);              // вычисляем переносы
		c[x] = (short)(Pyramid - Carry * BASE);
	}


	// Все готово, осталось лишь установить размер C, по первому
	// ненулевому коэффициенту.
	do { x--; } while (c[x]==0);
	C.Size = x+1;
}
```

Теперь можно реализовать алгоритм целиком.

```
// Вычислить C = A*B, работает вызов FastMul(A, B, A)
void FastMul(const BigInt &A,const BigInt &B, BigInt &C) {
	ulong x;
	const short *a=A.Coef, *b=B.Coef;

	if (!LongNum1 || !LongNum2) error("FastMul not initalized.");

	// Шаг 1
	ulong Len = 2;
	while ( Len <  A.Size + B.Size ) Len *=2;
	if ( Len < 8 ) Len = 8;    // FFT работает
```

```
        // Шаг 2. Переписываем число во временный массив и дополняем ведущими нулями
        //        Порядок цифр обратный, поэтому ведущие нули будут в конце
        for (x = 0; x < A.Size; x++)     LongNum1[x] = a[x];
        for (; x < Len; x++)             LongNum1[x] = 0.;

        // Шаги 3,4.
        RealFFT(LongNum1, Len, 1);

        if (&A == &B) {
               // Если имеем дело с возведением в квадрат - можно сэкономить одно БПФ
               RealFFTScalar ( LongNum1, LongNum1, Len );
        } else {
               // Иначе обработать и второе число
               for (x = 0; x < B.Size; x++)     LongNum2[x] = B.Coef[x];
               for (; x < Len; x++)             LongNum2[x] = 0.;

               RealFFT(LongNum2, Len, 1);

               RealFFTScalar ( LongNum1, LongNum2, Len );
        }

        RealFFT(LongNum1, Len, -1);             // Шаг 5
        CarryNormalize( LongNum1, Len, C);      // Шаг 6
}
```

По окончании умножения в MaxError хранится наибольшая ошибка округления. Для правильности результата ей лучше быть меньше 2.

Применение БПХ для вычисления свертки*

Умножение оперирует с действительными числами. Как мы говорили, в этом случае лучшие результаты может дать преобразование Хартли. Сделаем более совершенный вариант быстрого умножения на его основе.

Для коэффициентов ДПХ($a \otimes b$) теорема о свертке имеет вид:

$$ДПХ(a \otimes b)_k = \frac{1}{2}\left(c_k(d_k + d_{N-k}) + c_{N-k}(d_k - d_{N-k})\right), \text{ где } c = ДПХ(a),\ d = ДПХ(b),\ k=0..N-1 \quad \textbf{(3)}$$

Индексы считаются по mod N, то есть, вместо элементов с индексом N нужно брать элементы с нулевым индексом. Все шаги алгоритма БЫСТРОЕ_УМНОЖЕНИЕ остаются такими же, кроме шага 4, где координаты вектора следует вычислять по формуле (3).

Существование двух стилей БПХ и БПФ(разбиения по частоте и по времени) позволяет сделать одно существенное улучшение алгоритма. А именно, полностью избежать вызова FFTReOrder(), который занимает около 10% времени.
Действительно, БПХ_В требует на вход реорганизованный вектор и дает - обычный, в то время как БПХ_Ч работает наоборот: на входе – обычный, а в конце работы – реорганизованный. Поэтому можно изменить шаги следующим образом:

Векторы хранятся с обычным порядком индексов.

3. Вычислить БПХ_Ч на обоих массивах цифр.

   Вектор на выходе в реорганизованном виде.

4. Вычислить коэффициенты ДПХ($a \otimes b$) по формуле (3), учитывая изменения в порядке индексов.

   Вектор все еще в реорганизованном виде.

5. Вычислить БПХ_В, результатом которого будет ненормализованная свертка

   Окончательный вектор имеет обычный порядок индексов.

Все необходимые для реализации кирпичики уже есть. Единственно, нужно научиться применять формулу (3) к векторам, “перетасованным” функцией FFTReOrdeк().
В качестве примера рассмотрим вычисление ДПХ($a \otimes b$) для a=*БПХ_Ч*($a_1, a_2, \ldots$) , b=*БПХ_Ч*($b_1, b_2, \ldots$) на месте вектора a.

Элементы с индексами 0 и N/2 можно обработать отдельно. Для них формула приобретает особенно простой вид: $a_0 = a_0 b_0$, $a_{N/2} = a_{N/2} b_{N/2}$.

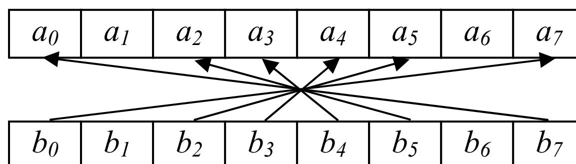
Для остальных элементов стоит учесть, что на вход формулы поступают числа с двух позиций, а выход записывается на одну. Чтобы не произошло лишнее затирание, можно вычислять по два значения одновременно, используя симметричность выражения.

$$a_k = \frac{1}{2}\left(a_k (b_k + b_{N-k}) + a_{N-k}(b_k - b_{N-k})\right)$$

$$a_{N-k} = \frac{1}{2}\left(a_{N-k}(b_{N-k} + b_k) - a_k (b_k - b_{N-k})\right)$$

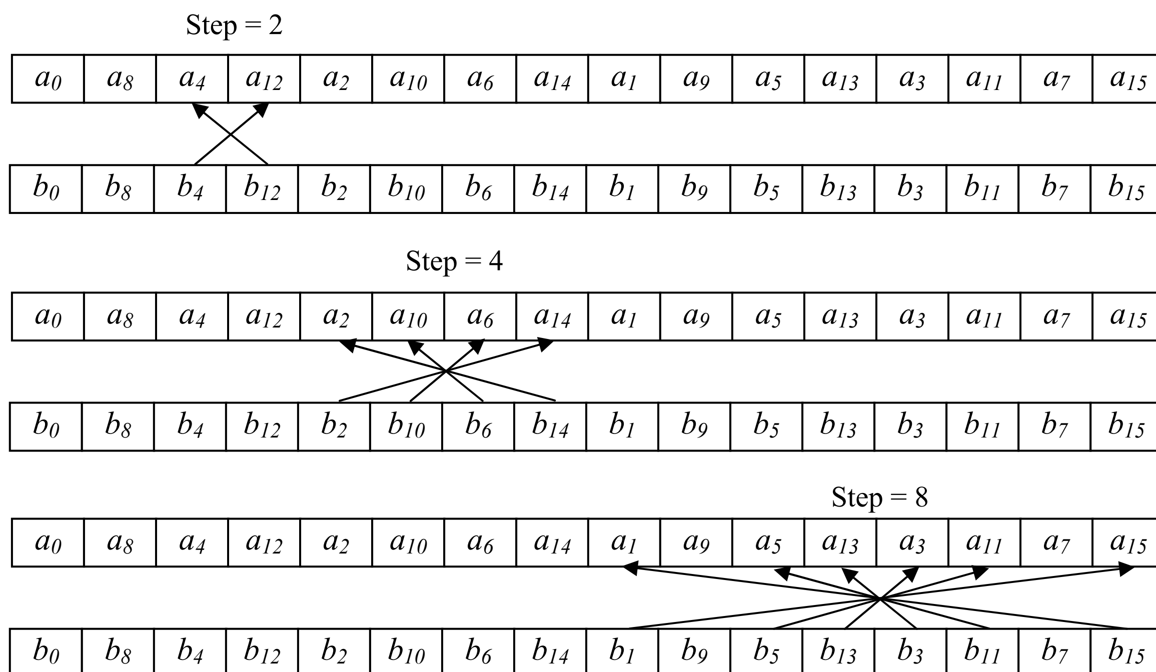
Теперь элементы вектора a с суммой индексов N будут преобразовываться попарно и “на месте”. Образуется “бабочка свертки”, выполнение которой для k=1…N/2-1 дает необходимый результат.

Если порядок индексов обычный, то вычисления производятся непосредственно по формулам. Для 8-элементных векторов:



Вычисление ДПХ свертки через ДПХ векторов. Индексы идут в обычном порядке, N=8

Интереснее с реорганизованными векторами. Чтобы учесть обратный битовый порядок, можно выполнять сначала бабочки для 2 элементов, затем для 4, для 8.. И так далее с шагом *2. Первые два элемента имеют индексы 0, N/2 и их следует обработать отдельно.



Вычисление ДПХ свертки через ДПХ векторов. Индексы в обратном битовом порядке, N=16

```
// Вычисление ДПХ свертки на месте LongNum1
// Не производится умножение на ½ в формуле (3), поэтому элементы
// получаются в 2 раза больше, чем должны быть.
```

```
void FHTConvolution(real *LongNum1, const real *LongNum2, ulong Len) {
	ulong Step=2, Step2=Step*2;
	ulong x,y;

	// Индексы 0, N/2. Как и все остальное, по формуле (3) без ½
	LongNum1[0]   = LongNum1[0] * 2.0 * LongNum2[0];
	LongNum1[1]   = LongNum1[1] * 2.0 * LongNum2[1];

	while (Step < Len) {       // Step – текущий шаг вычисления бабочек

		for (x=Step,y=Step2-1;x<Step2;x+=2,y-=2) {
			// Преобразовать элементы с индексами x, y по формуле (3) без ½
			real h1p,h1m,h2p,h2m;
			real s1,d1;
			h1p=LongNum1[x];
			h1m=LongNum1[y];
			s1=h1p+h1m;
			d1=h1p-h1m;
			h2p=LongNum2[x];
			h2m=LongNum2[y];
			LongNum1[x]=(h2p*s1+h2m*d1);
			LongNum1[y]=(h2m*s1-h2p*d1);
		}
		Step*=2;
		Step2*=2;
	}
}
```

Вычисление пары коэффициентов ДПХ свертки таким способом требует 4 сложений и 4 умножений, в то время как для преобразования Фурье эти цифры составляют 2 комлексных умножения , то есть 8 действительных умножений и 4 сложения. Однако, комплексный вектор в 2 раза короче, поэтому ДПХ требует на 2 сложения больше для пары элементов – всего на N сложений больше для всего вектора.

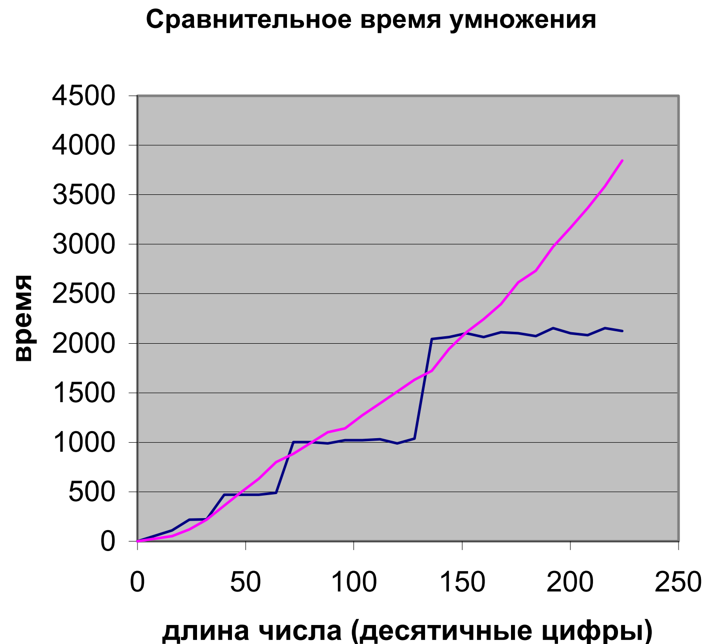
Теперь уже можно организовать быстрое умножение с использованием функций FHT_F(), FHT_T(). Для этого нужно добавить в конец FastMulInit() вызов CreateSineTable(MaxLen) и изменить коэффициент нормализации в CarryNormalize() на inv = 0.5 / Len: длина преобразования Len, а 0.5 появляется из-за того, что элементы свертки в два раза больше, чем надо.

```
// Быстрое умножение с совместным использованием разбиения по времени и по частоте
// Работает только для чисел, результат умножения которых имеет 8 или более разрядов
// Ограничение – следствие базиса рекурсии в 8 элементов для FFT_T.
// В любом случае, для таких маленьких чисел обычное умножение намного лучше.
void FastMul(const BigInt &A,const BigInt &B, BigInt &C) {
	ulong Len = 2;
	while ( Len <  A.Size + B.Size ) Len *=2;
	// (**)

	ulong x;
	const short *a=A.Coef, *b=B.Coef;

	for (x = 0; x < A.Size; x++)      LongNum1[x] = a[x];
	for (; x < Len; x++)              LongNum1[x] = 0.0;

	FHT_F(LongNum1,Len);

	if (a == b) {
		FHTConvolution(LongNum1,LongNum1,Len);
	} else {
		for (x = 0; x < B.Size; x++)      LongNum2[x] = b[x];
		for (; x < Len; x++)              LongNum2[x] = 0.0;
		FHT_F(LongNum2,Len);
		FHTConvolution(LongNum1,LongNum2,Len);
	 }

	FHT_T(LongNum1,Len);

	CarryNormalize(LongNum1, Len, C);
```

```
}
```

Сравнение "быстрого" и "школьного" умножения

Итак, алгоритм разобран и запрограммирован. Теперь самое время сравнить два метода.
Для сравнения возьмем БПХ-умножение как метод, оптимизированный нами гораздо лучше БПФ, и функцию Mul(), реализованную при создании класса BigInt. Умножаются два числа одинакового размера.



Как видно, поведение методов принципиально различается. Скачок в БПХ-методе возникает при переходе через степень двойки. Например, при перемножении 128-значных чисел вычисляется ДПХ 128-координатных векторов, а уже для умножения 129-значных векторов приходится работать с наименьшей степенью двух, большей 129 – с 256-значными векторами.
Исходя из тестовых данных, начиная со 150-300 десятичных цифр(в зависимости от компилятора эффективность кода разная) БПХ-алгоритм становится быстрее.

Естественным следствием из такого поведения алгоритмов является использование обычного умножения для чисел длины менее 40(возможно, 80) разрядов и применение "быстрого" метода при больших размерах.

```
// вставить вместо (**) в функцию FastMul
if ( Len < 40 ) {
       BigInt Atmp(A), Btmp(B);
       Mul(Atmp,Btmp,C);
       return;
}
```

Временные числа Atmp, Btmp создаются, чтобы сохранить возможность запуска FastMul(A,B,A), так как Mul(A, B, A) всегда обнуляет место для результата(здесь – число A) перед процедурой.

Конечно, рассмотренные реализации алгоритмов не являются наиболее оптимальными. Однако, судя по многочисленным тестам, хорошо запрограммированное "быстрое" умножение опережает школьный алгоритм того же качества в районе 250 цифр.

Точность вычислений и ее улучшения.

Как уже говорилось, в отличие от школьного алгоритма, “быстрое” умножение может ошибаться. Очень плохими векторами в этом смысле являются (BASE-1, BASE-1, …, BASE-1). При BASE=10000 БПФ-умножения позволяло умножать числа до 4 миллионов цифр. Оценка БПХ несколько лучше, здесь верхняя граница достигала 16 миллионов.

Очень сильно помочь может “балансировка” исходных векторов. Координаты входных векторов лежат в от 0 до BASE-1. Перед умножением преобразуем их к интервалу [-BASE/2…BASE/2-1], а после - вернем цифры в обычный диапазон.
Не важно, используется ли БПХ или БПФ, опыт показывает, что при таком подходе ошибка на случайном векторе уменьшается во много раз. Если умножение работает на пределе точности, то рекомендуется использовать этот способ.
Математически доказано, что максимальная длина чисел в этом случае увеличивается в 4 раза. То есть, если из формулы (2) следует, что для $N=2^{20}$ и типа double необходимое основание BASE < 4100, то при балансировке будет BASE < 16400. Практически, с основанием 10000 можно перемножать числа даже до 2^{30}, однако это выходит за рамки математических гарантий правильности результата.

Альтернативные методы умножения.

Существуют рекурсивные алгоритмы, основанные на подходе “разделяй-и-властвуй”. В качестве примера, можно упомянуть методы Карацубы, Тома-Кука. Они позволяют свести умножение большого числа к умножениям двух-трех в 2 раза меньших и нескольким сложениям, уменьшая тем самым асимптотику приблизительно до $O(n^{1.5})$. Эти алгоритмы были весьма популярны, когда операции с плавающей точкой осуществлялись очень медленно. Сейчас ситуация изменилась, и лучшие результаты показывают методы, использующие БПФ/БПХ.

Преобразования Фурье и Хартли можно делать и не только в комплексных числах. Можно, например, перейти в кольцо целых чисел Zp. Если заменить ω на первообразный корень из единицы в таком кольце. то получится *теоретико-числовое преобразование* (NTT, Number Theoretic Transform). Похожим образом работает метод Шенхаге-Штрассена.

При таком подходе избегаются ошибки округления – это позволяет умножать числа, состоящие из десятков и сотен миллиардов цифр, что сопряжено с большими проблемами, если использовать числа с плавающей точкой. В частности, такие алгоритмы всегда дают правильный ответ, как и школьное умножение, в то время как использование БПФ/БПХ далеко за границами формулы (2), может привести к непредсказуемому переполнению, когда вместо 999.1 получится 998.1 и округление “незаметно” призойдет не к тому числу. Вычислительная практика показывает, что такое при надлежащем контроле точности(ошибка округления менее 0.2) не происходит, но никаких гарантий, в отличие от NTT, нет.
Теоретическая база подобных алгоритмов довольно проста, однако качественная реализация требует выполнения большинства операций на низком уровне и быстрой модулярной арифметики. Из-за того, что умножение по модулю гораздо медленнее, чем умножение чисел с плавающей точкой, даже профессионально написанное NTT уступает по времени БПФ/БПХ. Поэтому подобные методы используют лишь по достижении границ точности, когда базового типа не хватает для правильных вычислительных операций, а увеличивать основание уже невыгодно из-за роста количества цифр и связанных с этим затрат времени/памяти.

У БПФ(БПХ)-умножения и NTT есть одно важное преимущество перед другими методами. Если одно число умножается на несколько других – его преобразование можно вычислить лишь один раз и запомнить. Все следующие умножения будут требовать 2 преобразования вместо трех. Школьное умножение, методы “разделяй-и-властвуй” и алгоритм Шенхаге-Штрассена не позволяют сделать ничего подобного.

Принципиально другого подхода требуют *разреженные* числа и многочлены, где почти все коэффициенты равны нулю. Они возникают во множестве практических задач и рассмотренные

методы для них будут малоэффективными. Существуют специальные алгоритмы, которые учитывают разреженность при хранении таких объектов и операциях с ними. Они, однако, выходят за пределы данного обзора.

(с) Кантор Илья, 2002
Отзывы и предложения по адресу algolist@manual.ru