УДК: 519.61

ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ АППРОКСИМИРУЮЩЕГО K-АРНОГО АЛГОРИТМА ВЫЧИСЛЕНИЯ НОД В БИБЛИОТЕКЕ MPIR

АМЕР И.,
АЛЬХАЛИДИ А.М.,
ИШМУХАМЕТОВ Ш.Т.
Аспиранты
Казанский федеральный университет

Аннотация: В этой статье мы рассмотрим процедуру эффективного программирования аппроксимирующего k-арного алгоритма вычисления НОД натуральных чисел на языке программирования C++ в среде программирования Visual Studio с установленной библиотекой длинных чисел MPIR. Также приведены некоторые теоретические расчеты, обосновывающие применения тех или решений.

Аппроксимирующий k-арный алгоритм был разработан в 2016 году Ш.Т. Ишмухаметовым [1] и предназначен для вычисления НОД для пары длинных натуральных чисел. В работе [2] Аркан и Ишмухаметов улучшили алгоритм путем сочетания двух стратегий построения по заданным числам A и B нового числа $C = xA + yB)/k$, такого, что пара (B, C) обладает тем же значением НОД или кратным НОД исходной пары (A, B) такого, что коэффициент редукции $\rho = B/C$ равен или превышает k. Параметр k может принимать произвольное значение, однако более эффективным является выбор k равным степени двойки с наиболее удачным значением $k = 1024$ и $k = 4096$.

Библиотека длинных чисел MPIR [3] является портированием под операционную систему Windows известной библиотеки длинных чисел GMP, разработанной первоначально для ОС Linux. Эта библиотека содержит тип длинных положительных чисел mpz_t и набор основных операций с этим форматом чисел. Описание переменной типа mpz_t выполняется так же, как и обычных типов, то есть командой

mpz_t список переменных, разделенных запятой ;

после чего переменные, перечисленные в этом списке должны быть инициализированы командой mpz_init. Переменная типа mpz_t является указателем на блок памяти, выделенной переменной этого типа. После использования переменной память, занимаемая переменной, может быть освобождена командой mpz_clear.

Приведем пример программирования алгоритма Евклида вычисления НОД для пары длинных чисел A и $B, A > B$:

```
mpz_t Euclid (mpz_t A,  mpz_t B){
        mpz_t  C;        mpz_init(C);
        size_t  L=mpz_size(B);
        while (L>0){
        mpz_fdiv_r(C, A, B);
        mpz_set(A, B); mpz_set(B, C); L=mpz_size(B);
```

```
}
return A;
mpz_clear(C;
}
```

В этой процедуре команда mpz_set присваивает значение второго операнда первому, а команда mpz_fdiv_r вычисляет остаток от деления второго операнда на третий и заносит полученное значение в первую переменную.

Длина переменной типа mpz_t определяется функцией mpz_size и вычисляется в лимбах, где лимб – это длина регистра процессора компьютера, в котором установлена библиотека, т.е. лимб равен 32 или 64 битам. Если число не превышает 2^{64} для 64-битового процессора, то длина такого числа равна 1. Условие $mpz\_size(B) > 0$ эквивалентно условию $B > 0$, но выполняется быстрее.

Перейдем теперь к программированию аппроксимирующего алгоритма. Примеры вычисления НОД по этому алгоритму и расширение алгоритма для вычисления обратных элементов по модулю заданного числа можно найти в [4]-[7].

```
mpz_t AKA(mpz_t A,  mpz_t B, int k){
        mpz_t AA, BB;  int i=0;
        mpz_init_set(AA, A); mpz_init_set(BB, B); // copy A, B to AA, BB
        size_t  L1=mpz_size(A),   L2=mpz_size(B);
        while(L2>0){
                i++;
                … основной цикл
}
return A;
}
```

Основной цикл представляет собой одну итерацию, в которой по паре нечетных чисел $A, B, A > B$, вычисляются целые числа x, y, такие что

$$0 < x \le \sqrt{k}, \qquad y < 0, \qquad Ax + By \equiv 0 \; mod \; k, \qquad Ax \approx -By \; . \; (1)$$

После этого вычисляется число $C = |(Ax + By)/k|$, C проверяется на четность и если четно, то сокращается на 2 до тех пор, пока не станет нечетным. После этого выполняется присвоение значений $A = B$, $B = C$, и выполняется переход к следующей итерации. Рассмотрим, как выполняется программирование отдельной итерации аппроксимирующего алгоритма.

Программирование основного шага процедуры AKA

На входе итерации подаются два числа A и $B, A > B > 0$, типа mpz_t. Процедура работает следующим образом:

1. Вычисляем длины чисел A и B:

size_t l1 = mpz_size(A); size_t l2 = mpz_size(B);

2. Находим приближенное значение отношения $rr = A/B$ с точностью до 1 (или до константы $\varepsilon < 1$).

Для этого отсекаем от чисел A и B одинаковое число L битовых разрядов справа. Обозначим через A_1 и B_1 оставшуюся часть, а через A_2 и B_2 – отсеченную часть.

$$A = A_1 \cdot 2^L + A_2, \qquad B = B_1 \cdot 2^L + B_2, \qquad \text{и } A_2 < 2^L, B_2 < 2^L.$$

Вычислим модуль разности между A/B и A_1/B_1 (по нашему алгоритму, $k \le A/B < k^2$, и значит, $A_1/B_1 < k^2$):

$$\left|\frac{A_1}{B_1}-\frac{A}{B}\right| = \left|\frac{A_1}{B_1}-\frac{A_1\cdot 2^L+A_2}{B_1\cdot 2^L+B_2}\right| = \left|\frac{A_1\cdot B_2 - A_2\cdot B_1}{(B_1\cdot 2^L+B_2)B_1}\right| < \left|\frac{A_1\cdot B_2}{B_1^2\cdot 2^L}\right| < \left|\frac{k^2}{2^L}\right|$$

Значит, чтобы обеспечить значение этой разности меньше 1, достаточно, чтобы длина отсекаемой части L была не меньше двух длин параметра k. В нашей программе мы рассматриваем значения $k \le 2^{16}$, поэтому для вычисления отношения A/B достаточно взять длину B_1 равно 32. Поэтому мы разбиваем все вычисление на 2 цикла. В первом рассматриваются пары чисел с длиной большего числа в лимбах больше 1, а во втором цикле, пары у которых длина равна одному лимбу (т.е. меньше 64 бит), и тогда эти числа умещаются в формате insigned long long.

Рассмотрим вычисление отношения A/B для случая, когда длины *A и B* в лимбах больше 1.

Вычисление отношения A/B для случая, когда длины *A и B* в лимбах больше 1.

1. Сначала необходимо найти старшие разряды B длины 2^{32}. Для этого считаем старший лимб и узнаем его длину в битах:

$f2 = A{-}{>}mp\_d[l1 - 1];$
$ll2 = blen(f2);$

Функция blen(long f) запрограммирована нами и вычисляет длину в битах целого положительного f. Она выполняется значительно быстрее стандартной функции двоичного логарифма, входящей в стандартную библиотеку языка C:

```
int blen(ull f) {
        int l=1, h2,h = (f < 4294967296); ull b;
        if (h == 1) {h2 = (f < 65536);
                if (h2 == 1) {l = 16;       b = 32768; //b=2^15
                while (f < b) {    b /= 2; l--;}}
        else {l = 32;      b = 65536;       while (f < b) {b /= 2; l--;  }}
        }
        else {
                h2 = (f < 4295032832);
                if (h2 == 1) { l = 48; b = 2147516416;// b=2^31 + 2^15
                        while (f < b) { b /= 2; l--;}}
                else {l = 64;      b = 9223372036854775808;// 2^63
                        while (f < b) { b /= 2; l--;}}
        }
        return l;
}
```

Если длина l старшего лимба B меньше 32, то надо увеличить старший сегмент $f2$, добавив к нему старшую часть второго лимба длины $dd = 32 - ll2$:

```
if (ll2 < 32) {dd = 32 - ll2;
                        f3 = (B->_mp_d[l1 - 2])>>(64-dd);
                        f2 = (f2 << dd);
                        f2+= f3;f4= (A->_mp_d[l1 - 2]) >> (64 - dd);
                        f1= (f1 << dd) + f4;
                }
```

При этом надо также увеличить f1, расширяя его на ту же длину dd:

```
f4= (A->_mp_d[l1 - 2]) >> (64 - dd);
        f1= (f1 << dd) + f4;
```

2. Следующим шагом является вычисление параметров $rr = A/B, q = AB^{-1}\ (mod\ k)$ и $\alpha = (rr - q)/k$ (обозначен через alp). Напомним, что согласно аппроксимирующему алгоритму выполняется соотношение

$$C = B\left|\left(\frac{rr - q}{k}\right)x + s\right| = |\alpha x + s|$$

Параметр α может принимать как положительное, так и отрицательное значение. Для объединения этих случаев мы ввели целый параметр $h = sign(\alpha)$ (знак α), заменили α на $h(r + ss)$ где ss – целая часть $|\alpha|$ и $0 < r < 1$. Вычисление q выполняется с помощью заранее сформированного массива обратных по модулю k элементов. Это позволяет добиться значительного ускорения алгоритма. Если значение k становится больше, то размер таблицы обратных элементов сильно увеличивается, поэтому делать k слишком большим – не выгодно. При $k > 2^{16}$ накладные расходы съедают выгоды от увеличения k.

Если длина B в лимбах равна 1, то заимствование дополнительной части B в младшем разряде становится невозможным, поэтому в этом случае выполняется полное деление A на B без их сокращений. Если же значение B окажется сравнимым или меньше значения k, то выполнение процедуры аппроксимирующего алгоритма является невыгодным и должно завершиться классической процедурой Евклида.

Построение дроби Фарея для параметра r

Одним из важных и ресурсоемких процедур аппроксимирующего алгоритма является поиск дроби Фарея, то есть правильной рациональной дроби, приближающей значение параметра r. В результате выполнения процедуры Farey(float r, int k, int &m, int &n) по действительному параметру $r, 0 < r < 1$, мы находим два целых числа m и n такие, чтобы выражение εn было минимальным, где

$$\varepsilon = \left|r - \frac{m}{n}\right|, n < k.$$

Поиск подходящей дроби выполняется в два этапа. На первом этапе ищем интервал, заключающий значение r:

$$\frac{m_1}{n_1} < r < \frac{m_2}{n_2},$$

такой, что $n_1 + n_2 \geq k$, $m_2 n_1 - m_1 n_2 = 1$. Последнее условие обеспечивает условие наилучшего приближения к r дробей m_1/n_1 и m_2/n_2. Поиск подходящей дроби выполняется в виде итерационной процедуры:

1. Ищем наибольшее число $n < k$ такое, что выполняется неравенство

$$\frac{1}{n+1} < \beta < \frac{1}{n}$$

2. Предположим, что на шаге i построены две дроби $r_1 = m_1/n_1$ и $r_2 = m_2/n_2$, такие что

$$r_1 < \beta < r_2, \quad \text{и} \quad n_1 + n_2 < k.$$

3. Определим медиану дробей r_1 и r_2 по формуле $r = (m_1 + m_2)/(n_1 + n_2)$.

4. Проверяем условие $\beta < r$. Если оно выполняется, то определим новое значение r_2, равным r, иначе определим $r_1 = r$. Если сумма знаменателей $n_1 + n_2 \geq k$, то перейдем к шагу 5, иначе, вернемся к шагу 2 с новыми значениями r_1 и r_2.

На втором этапе выбираем из двух дробей ту, которая дает меньшее значение разности $|rn_i - m_i|, i = 1,2$. Для этого вычисляем действительную переменную $x = (m_1 + m_2)/(n_1 + n_2)$ и сравниваем ее значение с r. Если $x < r$, то выходом процедуры Фарей является меньшая дробь m_1/n_1, иначе, большая дробь m_2/n_2.

Зам***ечание.*** Процедура Фарей съедает немалую часть времени вычисления НОД по аппроксимирующему алгоритму. Поэтому ускорение ее выполнения позволит существенно уменьшить общее время работы алгоритма. Для ускорения выполнения процедуры Фарей можно использовать предвычисление массива точек

$$float\ XMas = \left\{\frac{i}{j}, 1 \leq i < k - 1, i < j < k\right\}$$

с его последующей сортировкой. Тогда при поиске подходящей дроби достаточно найти два соседних числа $x_i, x_j,\ x_i < r < x_{i+1}$ и по этим числам выбрать подходящую дробь m_i/n_i.

Вычисление параметров x и y, удовлетворяющих условию (1)

После того как найдена подходящая дробь Фарея m/n необходимо вычислить значение параметров x и y по формулам:

$$x = n, y = -qn - (m + ss \cdot n)k$$

и число $C = |(Ax + By)/k.|$ После вычисления C необходимо проверить, не равно ли оно 0. Если $C \neq 0$, то проверяем C на четность, и при необходимости сокращать на 2, пока C не станет нечетным. После этого выполним присваивание $A = B,\ B = C$ и переходим к новой итерации.

Если выполнится случай $C = 0,$ переходим к завершению всей процедуры:

1. Вычисляем общий знаменатель текущих значений A и B

$$d = \frac{A}{y} = \frac{B}{x}.$$

Вычисляя $d,$ необходимо помнить, что НОД текущих значений A и B не обязан быть равен НОД исходной пары A и B, а является его кратным. Поэтому после вычисления d следует найти его НОД с исходными значениями A и B:

2. $d_1 = GCD(A, d),\ d_2 = GCD(B, d_1)$, где значения A и B равны исходным значениям, а НОД вычисляется по стандартной схеме алгоритма Евклида.

Заключение

В этой статье нами были рассмотрены основные этапы программирования аппроксимирующего ***k***-арного алгоритма. Этот алгоритм является улучшением ***k***-арного алгоритма Соренсона [8], [9] и позволяет добиться ускорения классического алгоритма Евклида в два и более раз.

Список литературы

1. Ishmukhametov S.T. An Approximating k-ary GCD Algorithm// S.T. Ishmukhametov. / Lobachevskii Journal of Mathematics, 2016, Volume 37, Issue 6, pp. 723-729

2. Аркан Аль Халиди Мухаммед. Эффективное программирование процедуры вычисления НОД натуральных чисел// М. Аркан, Ш.Т.Ишмухаметов/ Известия вузов. Математика (в печати).

3. Библиотека длинных чисел MPIR http://www.mpir.org/

4. Ишмухаметов Ш.Т. Вычисление коэффициентов Безу для k-арного алгоритма нахождения НОД// Ш.Т.Ишмухаметов, Б.Г. Мубараков, Маад Камаль Аль-Анни/ Изв. ВУЗов.Математика, 2017, №11, с.30-38

5. Амер И. Об ускорении k-арного алгоритма вычисления НОД натуральных чисел// И. Амер, Ш.Т. Ишмухаметов/Уч. записки Казан. унив. (сер. физ.-мат. науки), 2019. Том 161 Книга 1, с.110-119

6. Amer I. On acceleration of the k-ary GCD Algorithm// I.Amer, S.T.Ishmukhametov/ International Conference on Innovations in Engineering, Technology and Sciences" (ICIETS), IEEE series N 45147,Sept.2018

7. Antonov N.A. A Study Of Euclidean K-Ary Gcd Algorithm// N.A.Antonov, S.T. Ishmukhametov, Arkan Al Halidi M., M.E. Maiorova,/ REVISTA PUBLICANDO. - 2017. - Vol.4, Is.13. - P.1108-1125

8. Sorenson J. The k-ary GCD Algorithm//J.Sorenson/ Universitet of Wisconsin-Madison, Tecn.Report (1990), pp.1–20

9. Sorrenson J. Two fast GCD Algorithms// J.Sorenson/ J.Alg. 16, №1, 110–144 (1994).