УЧЕНЫЕ ЗАПИСКИ КАЗАНСКОГО УНИВЕРСИТЕТА.
СЕРИЯ ФИЗИКО-МАТЕМАТИЧЕСКИЕ НАУКИ

2019, Т. 161, кн. 3
С. 393–404

ISSN 2541-7746 (Print)
ISSN 2500-2198 (Online)

УДК 519.688

doi: 10.26907/2541-7746.2019.3.393-404

ЭФФЕКТИВНОЕ УДАЛЕНИЕ ДЕЛИТЕЛЕЙ В k-АРНОМ АЛГОРИТМЕ

Р.Р. Еникеев

Казанский (Приволжский) федеральный университет, г. Казань, 420008, Россия

Аннотация

Один из наиболее эффективных алгоритмов вычисления наибольшего общего делителя при работе с длинными числами – k-арный алгоритм. Вычисление наибольшего общего делителя используется во многих криптографических алгоритмах. Если при нахождении наибольшего общего делителя u, v в k-арном алгоритме бинарные длины чисел u и v близки, то используется шаг редукции $t = |au + bv|/k$, где $0 < a$, $|b| \le \lceil\sqrt{k}\rceil$: $au + bv \equiv 0 \pmod{k}$. Если числа u и v сильно отличаются по длине, то для сокращения u до длины v используется операция dmod, которая определяется по формуле $|u - cv|/2^L$, где $c = uv^{-1} \bmod 2^L$, $L = L(u) - L(v)$, а функция $L(a)$ – бинарная длина числа a. Ускорение k-арного алгоритма проводится путем минимизации количества операций удаления делителей в главном цикле при $k = 2^{2W}$, где W – длина машинного слова. Мы объединяем эту процедуру с операцией деления на 2^{iW}, для которой описана быстрая реализация за $O(1)$ операций, и рассматриваем новый способ вычисления коэффициентов, что в результате позволяет полностью избавиться от удаления делителей перед шагом редукции и позволяет производить удаление делителей перед операцией dmod лишь в 1/3 случаев, тем самым уменьшив общее количество операций, выполняемых над длинными числами. Предложенный нами метод ускоряет главный цикл k-арного алгоритма на 3–16% в зависимости от длины числа.

Ключевые слова: наибольший общий делитель, длинные числа, k-арный алгоритм, удаление делителей

Введение

Длинное число (multi-precision integer) – число, размер которого больше длины машинного слова. Длину машинного слова будем обозначать через W. Операции над длинными числами реализуются программно [1]. Числа, которые не превосходят длины машинного слова, будем обозначать как одинарное число, или число с одинарной точностью (single-precision integer), а числа меньшие 2^{2W} – двойные, или числа с двойной точностью (double-precision integer).

Самым известным алгоритмом вычисления наибольшего общего делителя (НОД) является алгоритм Евклида, известный еще с давних времен и основанный на вычислении остатка от деления. Однако возрастающие требования к компьютерным приложениям, в которых при вычислениях используются длинные числа, привели к необходимости ускорения алгоритма Евклида [2] и создания новых более эффективных методов нахождения НОД, например бинарного алгоритма [3, 4].

Одним из наиболее эффективных методов вычисления НОД является k-арный алгоритм, который был предложен Соренсоном [5]. В качестве k может выступать любое натуральное число, большее единицы, но наиболее эффективным выбором k является степень двойки, $k = 2^m$, в силу того, что числа в памяти компьютера представляются в двоичном виде, что позволяет производить деление на k

очень быстро (по сравнению с делением на произвольное число). Для нахождения НОД $u \geq v > 0$, имеющих почти равную длину в бинарном представлении, в этом алгоритме используется шаг редукции $t = |au + bv|/k$, где $0 < a,\ |b| \leq \lceil\sqrt{k}\rceil$: $au + bv \equiv 0 \pmod{k}$. Данные коэффициенты a, b существуют всегда. Соренсон предлагал использовать предвычисленные таблицы значений a и b, то есть для каждой пары $u \bmod k$ и $v \bmod k$ мы должны хранить соответствующие значения a и b. Данный способ нахождения коэффициентов является неэффективным при больших k, поэтому Вебер [6] и Джебелеан [7] независимо друг от друга предложили способ нахождения a и b, основанный на вычислении обратного элемента по модулю 2^m с последующим применением расширенного алгоритма Евклида, что позволило отказаться от использования предвычисленных таблиц и использовать в качестве $k = 2^{2W}$. При этом $a < 2^W$ и $b < 2^W$, а значит, для хранения a или b достаточно использовать одно машинное слово.

После шага редукции мы получаем число t, которое практически на W бит короче, чем u и v, так как $t = |au + bv|/k < |2^W u + 2^W u|/2^{2W} = 2u/2^W$. Если применить шаг редукции к v и t, мы получим новое число, длина которого меньше длины v на W бит, при этом производится умножение длинного числа на число одинарной точности два раза. Данный шаг можно выполнить эффективнее с помощью операции dmod, которая определяется по формуле $|v - ct|/2^W$, где $c = v/t \bmod 2^W$. Отметим, что число $c < 2^W$, так как c вычисляется по модулю 2^W, следовательно, для хранения c достаточно одинарного числа так же, как для a и b. Операция dmod позволяет сократить v на W бит с использованием всего одного умножения длинного числа на одинарное. Таким образом, в k-арном алгоритме используется поочередное выполнение шага редукции и dmod, после выполнения которых два числа, равных по размеру, уменьшаются на одно машинное слово.

Перед шагом редукции и dmod необходимо удалить все делители двойки из v, потому что во время нахождения коэффициентов a, b или c выполняется операция $u/v \bmod 2^m$, которая вычислима, только если v и 2^m взаимно просты [8].

Операция $t = |au + bv|$ может внести ложные делители (spurious factors) [6]: $\text{НОД}(t, v) = s \cdot \text{НОД}(u, v)$, где s – ложные делители, при этом s является натуральным числом. Пусть после выполнения главного цикла был получен результат $g' = \text{НОД}(u, v) \cdot s$. Для удаления ложных делителей необходимо выполнить операцию $g = \text{АЕ}(u, \text{АЕ}(v, g'))$, где АЕ – алгоритм Евклида, а g – искомый НОД чисел u и v.

Приводимый ниже алгоритм 1 описывает главный цикл k-арного алгоритма нахождения НОД, но без удаления ложных делителей, которое выполняется уже после главного цикла. В алгоритме 1 на входе задаются нечетные числа u и v, так как удаление всех делителей двойки из u и v происходит до главного цикла. В этом алгоритме используется функция $L(a)$, которая вычисляет бинарную длину a. Эта функция имеет сложность $O(1)$ операций. В алгоритме 1 используется также рабочая константа S, благодаря которой происходит поочередное выполнение шага редукции и dmod: если длина u отличается от длины v менее чем на S, то выполняется шаг редукции, иначе выполняется операция dmod. Константу S мы можем задавать сами. Будем обозначать данный алгоритм через ГЦ1.

После выполнения главного цикла в полученный результат могут быть внесены ложные делители. Это дает нам простой способ тестирования правильности реализации алгоритма ГЦ1: необходимо проверить, что значение, найденное в результате выполнения ГЦ1, делится на НОД чисел u и v.

Расширенная версия k-арного алгоритма позволяет найти числа x и y, удовлетворяющие равенству $ux + vy = \text{НОД}(u, v)$ [10, 11]. Расширенные алгоритмы поиска НОД используются в криптографии для нахождения обратного элемента.

Алгоритм 1 Главный цикл k-арного алгоритма

Input: $u > v, u$ и v нечетные
Output: НОД(u, v) с ложными делителями
while v != 0 **do**
  if $L(u) - L(v) < S$ **then**
    Найти a, b: $a * u + b * v = 0 \pmod k$
    $u := |a * u + b * v|/k$ // реализуется через $u := a * u$; u += $b * v$; $u := |u|/k$;
  else
    $c := u/v \bmod 2^W$ // шаг dmod
    $u := |u - c * v|/2^W$ // реализуется через u −= $c * v$; $u := |u|/2^W$;
  Удалить из u общие делители с k
  if $u < v$ **then**
    $swap(u, v)$
return u

1. Представление длинных чисел в библиотеке MPIR

В данном разделе кратко описано внутреннее устройство библиотеки MPIR [1], одной из самых популярных библиотек для работы с длинными целыми числами для C++, чтобы понять, как выполняется удаление делителей двойки, для вычисления сложности алгоритма и его дальнейшей оптимизации. Мы также рассматривали другие библиотеки: TTMath, Arbitrary Precision Math C++ Package, Class Library for Numbers (CLN), Number Theory Library (NTL), Apfloat, C++ Big Integer Library, MAPM, ARPREC, InfInt. Но в документации этих библиотек не описана внутренняя реализация длинных чисел, знание которой позволяет эффективно реализовать k-арный алгоритм.

Лимб (limb) – часть длинного числа, которая умещается в одном машинном слове и обычно имеет размер либо $W = 32$, либо $W = 64$ бита, то есть лимб является одинарным числом. Длинное целое число хранится как массив лимбов и представлено в MPIR как тип mpz_t, являющийся структурой. Опишем наиболее значимые для нас поля этой структуры:

- _mp_d – указатель на массив лимбов;
- _mp_size – количество лимбов в числе, то есть количество лимбов, занимаемых числом в массиве _mp_d. Для удобства будем обозначать _mp_size через N. При этом _mp_d[0] – наименее значащий (младший) лимб, а _mp_d[$N-1$] – наиболее значащий (старший).

Само число равно

$$\sum_{i=0}^{N-1} \text{\_mp\_d}[i] \cdot 2^{iW}.$$

2. Анализ сложности

Оценим сложность каждой операции в алгоритме 1.

- Операции вычисления a, b выполняются за $O(\log_2^2 k) = O(1)$ операций [6], а сложность вычисления c составляет $O(\log 2^W) = O(1)$ операций [9].
- Сложность умножения длинного числа на обычное (код "$a * u$") составляет $O(N)$ операций.
- Шаг редукции использует функцию addmul из библиотеки MPIR (код "u += $b * v$"), а операция dmod – submul (код "u −= $b * v$"), которые представляют собой одну операцию, что позволяет сэкономить память. Функции addmul и submul выполняются за $O(N)$ шагов.

• Деление на 2^{iW} происходит с помощью сдвига всех лимбов на соответствующее количество позиций, i, в массиве _mp_d и имеет сложность $O(N)$ операций.

• Деление на 2^i при $W \nmid i$ (удаление делителей) выполняется с помощью сдвига вправо всех лимбов длинного числа на i бит путем использования флага переноса. Для выполнения этой операции необходимо $O(N)$ шагов.

Очевидно, что два последних пункта выполняются быстрее деления на произвольное число.

Заметим, что хотя вычисление коэффициентов фактически и выполняется за $O(1)$ операций, однако при малых N их нахождение может занимать больше времени, чем все операции над длинными числами.

При оценке сложности мы будем использовать обозначение $l \cdot O(N)$, означающее, что при вычислении алгоритма выполняется l операций, сложность каждой из которых составляет $O(N)$ операций.

Вероятность того, что нам нужно будет удалять общие делители с $k = 2^m$, равняется вероятности того, что после шага редукции $(2W+1)$-й или после операции dmod $(W+1)$-й бит равен 0, и она равняется $1/2$. Этот факт в представлении сложности данной операции мы будем отражать следующим образом: $0.5 \cdot O(N)$.

Шаг редукции и операция dmod выполняются поочередно, поэтому в следующих разделах для удобства анализа главного цикла мы условимся обозначать в качестве сложности одной итерации общую сложность шага редукции и dmod, которая для алгоритма ГЦ1 равняется $3.5 \cdot O(N) + 2.5 \cdot O(N) = 6 \cdot O(N)$.

3. Объединение удаления делителей и деления на 2^W

Заметим, что деление на 2^{iW} и удаление общих делителей вычисляются схожим образом, с помощью сдвига числа вправо, и, что самое главное, выполняются друг за другом, поэтому для оптимизации главного цикла мы можем несколько модифицировать алгоритм 1, объединяя указанные два этапа. Для этого в алгоритме 2 мы пропускаем шаг деления, вычисляя сначала количество делителей двойки, и лишь затем сдвигаем число на соответствующее количество бит. Алгоритм 2 использует функцию ZC, вычисляющую количество делителей двойки или, что то же самое, число наименее значащих битов, равных нулю. Например, если функции ZC задать на входе число 101011000_2, то на выходе получим 3. Функция ZC выполняется за $O(1)$ операций. На шаге редукции мы обращаемся к 3-му лимбу (с помощью кода "v._mp_d[2]"), а после вычисления операции dmod – ко 2-му, так как наименее значащие лимбы перед ними уже равняются нулю. Таким образом, за счет объединения этих двух операций мы получаем, что сложность одной итерации равна $3 \cdot O(N) + 2 \cdot O(N) = 5 \cdot O(N)$. Обозначим этот алгоритм через ГЦ2.

Оценим размер памяти, который необходим для хранения u и v в ГЦ1 и ГЦ2. Алгоритм ГЦ2 объединяет два последовательных этапа сдвига вправо из ГЦ1 в один, поэтому количество памяти, используемое при выполнении этих алгоритмов, равно. Поскольку после шага редукции и выполнения операции dmod происходит сдвиг, то максимальное значение памяти переменных u и v достигается на первой или второй итерации. Во время шага редукции выполнение кода "$a * u$" и "$b * v$" увеличивает размер исходных чисел на 1 машинное слово, а во время сложения (при выполнении кода "$a * u + b * v$"), может появиться бит переноса, поэтому для вычисления шага редукции необходимо дополнительно 2 лимба к длине исходного u. В главном цикле при выполнении операции dmod на первой итерации или при выполнении шага редукции на второй итерации размер v может увеличиться на 1 бит, что в худшем случае приведет к необходимости одного дополнительного лимба.

Алгоритм 2 Модифицированый главный цикл k-арного алгоритма

Input: $u > v, u$ и v нечетные
Output: НОД(u, v) с ложными делителями
while v != 0 **do**
  $d := L(u) - L(v)$
  if $d < S$ **then**
    Найти a, b: $a * u + b * v = 0 \pmod k$
    $u := |a * u + b * v|$
    $m := ZC(v.\_mp\_d[2]) + 2 * W$
  else
    $c := u/v \bmod 2^W$
    $u := |u - c * v|$
    $m := ZC(v.\_mp\_d[1]) + W$
  $u := u \gg m$ // Сдвиг вправо
  if $u < v$ **then**
    $swap(u, v)$
return u

4. Быстрое деление на 2^{iW}

Как описано ранее, деление на 2^{iW} выполняется за $O(N)$ операций, что достаточно медленно из-за внутренней реализации операции сдвига в библиотеке MPIR, потому что операция деления на 2^{iW} не предусмотрена в MPIR, так как в общем случае сдвиг на величину, кратную машинному слову, возникает редко и нет необходимости в оптимизации данной операции (что повлечет изменение всей библиотеки MPIR), поэтому реализовать операцию деления на 2^{iW} нужно нам самим. Очевидным решением для быстрой реализации деления на 2^{iW} является передвигание указателя _mp_d на i-й элемент массива, которое выполняется за $O(1)$ операций, при этом необходимо сохранить значение указателя на исходное начало массива _mp_d.

Для экономии памяти при присваивании нового значения числу, которое мы поделили на 2^{iW} подобным образом, нам необходимо записать результат в исходное начало массива _mp_d. Например, мы сдвинули указатель _mp_d у числа u, тогда результат выполнения кода "$u = a*u$" необходимо записать в исходное начало массива u и нужно передвинуть указатель _mp_d в исходное начало массива, представляющего число u.

Будем обозначать данный метод через ГЦ3. Сложность одной итерации в нем будет равняться $2.5 \cdot O(N) + 1.5 \cdot O(N) = 4 \cdot O(N)$ операций.

Алгоритм ГЦ2 объединяет два последовательных этапа в один, а ГЦ3, в отличие от ГЦ1, использует другую реализацию операции деления, поэтому все три описанных алгоритма будут на каждой итерации получать одинаковые коэффициенты, значение вычисленного числа и, естественно, результат работы, что дает простой критерий проверки корректности (тестирования) реализаций данных алгоритмов во время их тестирования.

За исключением способа удаления делителей и деления на 2^{iW} расширенные версии алгоритмов ГЦ2 и ГЦ3 совпадают с расширенной версией алгоритма ГЦ1, потому что изменения, внесенные в ГЦ1, не отражаются на вычислении x и y.

Найдем количество памяти, необходимое при работе алгоритма ГЦ3, что весьма важно, так как в нем, в отличие от ГЦ1 и ГЦ2, не происходит сдвига сразу, что может повлечь за собой выход за границы массива _mp_d. Рассуждения аналогичны описанному ранее анализу благодаря тому, что мы записываем результат

в исходное начало массива. Только из-за отсутствия сдвига может произойти такая ситуация на второй итерации, когда начальное u во время операции dmod увеличится на 1 бит, поэтому в этом алгоритме необходимо иметь дополнительно 3 лимба к исходным размерам u и v.

Быстрое деление на 2^{iW} за $O(1)$ операций является необходимым условием для работы алгоритма, описанного в следующей главе.

5. Минимизация количества операций удаления делителей

Сначала оценим «рентабельность» удаления делителей. Как отмечалось ранее, появление делителей носит случайный характер. Поэтому мы можем вычислить предполагаемое количество (математическое ожидание) делителей двойки, которое эквивалентно числу нулей наименее значащих бит. Используем описанную ранее функцию ZC, которая вычисляет количество делителей двойки. Сначала найдем вероятность $P(\mathrm{ZC}(u') == i)$ того, что в числе u' имеется ровно i наименее значащих нулей, где u' является лимбом длинного числа и не равно нулю (иначе мы могли бы просто использовать алгоритм из предыдущей главы). Искомая вероятность вычисляется как вероятность появления i последовательных нулей и обязательно следующей за ними единицей

$$P(\mathrm{ZC}(u') == i) = \frac{1}{2^i} \cdot \frac{1}{2} = \frac{1}{2^{i+1}}.$$

Тогда математическое ожидание количества делителей есть

$$M[\mathrm{ZC}(u')] = \sum_{i=0}^{W-1} \frac{i}{2^{i+1}} = \left(\frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^W}\right) + \left(\frac{1}{8} + \cdots + \frac{1}{2^W}\right) + \cdots +$$
$$+ \left(\frac{1}{2^{W-1}} + \frac{1}{2^W}\right) + \frac{1}{2^W} = \sum_{i=1}^{W-1} \frac{2^i - 1}{2^W} = \frac{2^W - (W+1)}{2^W} = 1 - \frac{W+1}{2^W}.$$

При $W = 32$ или $W = 64$ мы получаем, что $M[\mathrm{ZC}(u')] \approx 1$. Таким образом, в среднем появляется лишь один делитель и для его удаления требуется сдвинуть все лимбы длинного числа, что весьма неэффективно.

В следующих разделах для повышения эффективности мы переместили операцию удаления делителей из конца нашего цикла в начало, чтобы перед выполнением шага редукции операции dmod иметь возможность определить необходимость избавления от делителей.

5.1. Удаление делителей перед шагом редукции. Идея алгоритма, описанного в данном подразделе, заключается в том, что для вычисления $d = u/v \bmod k$ вместо v (знаменателя) мы используем то из чисел u, v, которое имеет меньшее количество делителей двойки (наименее значащих бит), а в качестве делителя выбираем оставшееся. При нахождении обратного элемента и вычислении коэффициентов мы сдвигаем не длинные числа, а двойные.

Чтобы не было необходимости в удалении делителей двойки из длинных чисел, как это проводится в алгоритмах 1 и 2, мы изменим функцию нахождения коэффициентов a и b. Модифицированный способ вычисления коэффициентов a и b описан в алгоритме 3. В этом алгоритме определена функция CalculateAB(u, v, m), которая находит коэффициенты $0 < a$, $|b| < \lceil 2^{m/2} \rceil$, удовлетворяющие условию $au + bv \equiv 0 \pmod{2^m}$, где v – нечетное. В работе Вебера [6] эта функция называется ReducedRatMod.

Алгоритм 3 Модифицированый алгоритм нахождения коэффициентов a, b

Input: u, v
Output: Коэффициенты $a, b : |a|, |b| < 2^W$ и $a * u + b * v = 0 \pmod{2^{2W}}$
$(u', v') = (u \bmod 2^{2W}, v \bmod 2^{2W})$ // Берем два последних лимба
if u' и v' четные числа **then**
  $z := min(ZC(u), ZC(v))$
  $(u', v') := (u' \gg z, v' \gg z)$
  $s := 2 * W - z$
else
  $s := 2 * W$
if v' - нечетное число **then**
  return $CalculateAB(u', v', s)$
else
  $(b, a) := CalculateAB(v', u', s)$
  return (a, b)

Заметим, что в алгоритме 3 сдвигаются только числа двойной точности u', v', а не длинные числа u, v.

Покажем корректность данного алгоритма. Пусть либо u, либо v изначально нечетно. Если v нечетно, то нетрудно заметить, что наш алгоритм аналогичен первоначальному, который находит необходимые коэффициенты по модулю 2^{2W}.

Если же u нечетно, тогда мы, не удаляя делителей из v, находим коэффициенты a, b по модулю 2^{2W}. В данном случае мы используем u' для нахождения обратного элемента, а в первоначальном алгоритме для нахождения коэффициентов использовалось вычисление v'^{-1}. В отличие от исходного алгоритма, здесь a может быть отрицательным, но в таком случае b будет положительным, что никак не влияет на сам алгоритм.

И, наконец, рассмотрим случай, когда и u, и v в начале алгоритма оба четные. Сначала мы находим у чисел $u' = u \bmod 2^{2W}$ и $v' = v \bmod 2^{2W}$ общее количество z делителей двойки. Затем вычисляем $u'' = u' \gg z$ (сдвиг вправо на z бит), $v'' = v' \gg z$ и находим коэффициенты a, b: $au'' + bv'' = 0 \pmod{2^{2W-z}}$. Умножая последнее равенство на 2^z, получаем

$$au''2^z + bv''2^z = 0 \pmod{2^{2W}},$$
$$au' + bv' = 0 \pmod{2^{2W}},$$
$$au + bv = 0 \pmod{2^{2W}}.$$

Таким образом, полученные коэффициенты a, b для u'', v'' являются искомыми коэффициентами и для u, v. В итоге при любых исходных значениях u, v мы получаем требуемые коэффициенты a, b и необходимость удаления делителей перед основным преобразованием (после шага dmod) полностью отпадает.

5.2. Удаление делителей перед dmod. Для эффективного удаления делителей перед выполнением операции dmod мы используем метод, описанный в алгоритме 4. Этот модифицированный алгоритм применяется вместо операции dmod. В этом алгоритме мы находим количество z_u и z_v делителей двойки у чисел u и v соответственно. Если $z_u \geq z_v$, то применяется операция dmod, а при вычислении c производится сдвиг одинарных чисел u' и v', которые являются младшими лимбами чисел u и v соответственно. Следовательно, в случае, когда $z_u \geq z_v$, мы

Алгоритм 4 Модифицированый алгоритм dmod

Input: u, v: $L(u) - L(v) \geq W$
Output: Уменьшение размера u на W бит
$(u', v') := (u \bmod 2^W, v \bmod 2^W)$ // Берем последний лимб
$(z_u, z_v) := (ZC(u), ZC(v))$
if $z_v > z_u$ **then**
  $v := v \gg z_v$
  $(v', m) := (v \bmod 2^W, W)$
else
  $(u', v', m) := (u' \gg z_v, v' \gg z_v, W - z_v)$
$c := u'/v' \bmod 2^m$
$u := |u - c * v|/2^W$ // Используется быстрое деление

избавляемся от необходимости удаления делителей двойки у длинных чисел. Если $z_u < z_v$, то вычислить c невозможно, так как v' содержит делители двойки, поэтому не существует v'^{-1}. Следовательно, в случае, когда $z_u < z_v$, необходимо избавиться от делителей двойки из длинного числа v. Таким образом, в алгоритме 4 мы сдвигаем длинное число v, только если количество наименее значащих нулей в v больше, чем в u. Результатом работы алгоритма является уменьшение u на W бит.

Продемонстрируем корректность алгоритма. Если $z_v > z_u$, то алгоритм эквивалентен исходному: мы удаляем сначала все делители двойки из v, а потом вычисляем новое значение u, уменьшая его длину на W бит. В противном случае корректность устанавливается так же, как в п. 5.1, когда u, v оба четные.

Вычислим, как часто нам нужно будет сдвигать v в алгоритме 4. Используя то, что $P(\mathrm{ZC}(u) > i) = 1/2^{i+1}$ (должно быть как минимум $i+1$ подряд идущих нулей), мы получаем, что вероятность того, что $ZC(v) > ZC(u)$ есть

$$P(ZC(v) > ZC(u)) = \sum_{i=0}^{W-1} [P(\mathrm{ZC}(v) > i) \cdot P(\mathrm{ZC}(u) == i)] =$$
$$= \sum_{i=0}^{W-1} \left(\frac{1}{2^{i+1}} \cdot \frac{1}{2^{i+1}} \right) = \frac{1}{4} \sum_{i=0}^{W-1} \frac{1}{4^i} = \frac{1}{4} \cdot \frac{1 - (1/4)^W}{1 - (1/4)} = \frac{4^W - 1}{4^W \cdot 3}.$$

При $W = 32$ или $W = 64$ вероятность сдвига $P(ZC(v) > ZC(u)) \approx 1/3$, откуда вытекает небольшая экономия по сравнению с исходной версией алгоритма, где требовалось удалять делители перед dmod с вероятностью $1/2$.

Алгоритм, который объединяет ГЦ3 и алгоритмы 3, 4, обозначим через ГЦ4. После изменения способа вычисления коэффициентов алгоритм ГЦ4 не будет возвращать тот же результат, что и ГЦ1–ГЦ3. Для проверки правильности реализации (тестирования) ГЦ4 можно использовать критерий, используемый для проверки правильности ГЦ1, то есть нам необходимо проверить, что результат выполнения ГЦ4 делится на истинный НОД чисел u, v. Сложность алгоритма ГЦ4 составляет $2 \cdot O(N) + 1.33 \cdot O(N) = 3.33 \cdot O(N)$ операций.

Быстрое деление в ГЦ4 имеет важное значение, потому что без него использование алгоритмов 3, 4 нецелесообразно, так как мы в любом случае будем сдвигать все наше число во время деления на каждой итерации и можно просто использовать ГЦ2.

Алгоритм 5 Обобщенный алгоритм нахождения коэффициентов a, b

Input: $u, v, k : k \nmid u$
Output: Коэффициенты a, b: $0 < a, |b| < \sqrt{k}$ и $a * u + b * v = 0 \pmod{k}$
$(u', v') := (u \bmod k, v \bmod k)$
$(g_u, g_v) := (\text{НОД}(u', k), \text{НОД}(v', k))$
if $g_u, g_v < \sqrt{k}$ **then**
  $v'' := v'/g_v$
  $c := v''/g_v \bmod k/g_v$
  return Найти коэффициенты a, b из c, g_v по модулю k
else
  return Найти коэффициенты с помощью сдвига

Без удаления делителей расширенная версия алгоритма ГЦ4 претерпевает несущественные изменения, теперь после шага редукции следующие значения коэффициентов Безу вычисляется по формулам $(ax_0 + bx_1)/k$ и $(ay_0 + by_1)/k$ [11]. После выполнении операции dmod, когда происходит сдвиг, эти значения находятся по первоначальным формулам.

6. Удаление делителей для произвольного k

Джебелеан в [7] рассматривает только случай, когда $k = 2^{2W}$, а числа представлены по основанию 2. Вебер в [6] описывает алгоритмы, где используются числа по основанию β. В этой работе он описывает функцию для вычисления коэффициентов для $k = \beta^{2m}$, поэтому при обобщении алгоритмов 3 и 4 будем полагать, что числа u и v представлены в системе счисления β.

Для обобщения алгоритма 4 нам достаточно заменить в этом алгоритме "2" на "β".

Обобщим алгоритм 3 для произвольного k в общем случае в предположении, что операции деления и нахождения остатка от деления на k и все вычисления по модулю k выполняются быстро (так же, как в предыдущих разделах мы описывали быстрое деления на 2^{iW}, а все вычисления выполнялись над числами длины, равной одному или двум машинным словам).

В алгоритме 5 мы сначала вычисляем c и получаем коэффициенты (c, g_v), для которых

$$\begin{aligned} u - cv' &= 0 \pmod{k/g_v}, \\ g_v u - cv' g_v &= 0 \pmod{k}, \\ g_v u - cv &= 0 \pmod{k}. \end{aligned}$$

Таким образом, пара (c, g_v) удовлетворяет лишь половине требований, а именно $g_v u - cv = 0 \pmod{k}$. Чтобы получить коэффициенты a, b (последняя строка внутри тела if в алгоритме 5), которые удовлетворяют также и неравенствам $0 < a$, $|b| < k$, мы применим несколько модифицированный алгоритм ReducedRatMod из [6], в котором зададим $(n_1, d_1) := (k/g_v, 0)$, $(n_2, d_2) := (c, g_v)$. После каждой итерации выполняется равенство $n_1|d_2| + n_2|d_1| = k$, а после окончания цикла будет выполнено условие $n_2 < \sqrt{k} \le n_1$, откуда следует, что $\sqrt{k}|d_2| \le n_1|d_2| < k$. Следовательно, получим условие $|d_2| < \sqrt{k}$. В итоге мы получаем коэффициенты $0 < n_2, |d_2| < \sqrt{k}$, удовлетворяющие всем необходимым критериям.

Для работы алгоритма важно, чтобы $n_2 \nmid n_1$, то есть $n_1 - \lfloor n_1/n_2 \rfloor n_2 > 0$. Согласно модификации алгоритма это условие выполняется на всех итерациях при $g_u, g_v < \sqrt{k}$, так как $\text{НОД}(k, c) \neq 1$ и в худшем случае НОД равен g_u, а значит,

Табл. 1

Относительное время работы алгоритмов в зависимости от длины чисел

N	ГЦ1	ГЦ2	ГЦ3	ГЦ4
10–400	1.02–1.03	1.01	1	1.01
400–600	1.02–1.03	1.01–1.02	1	≈ 1
600–1500	1.03	1.02	≈ 1	1
1500–3000	1.03–1.04	1.02	1.01	1
3000–10000	1.07–1.19	1.05–1.12	1.02–1.05	1

выход из цикла будет осуществлен раньше, чем $n_1 - \lfloor n_1/n_2 \rfloor n_2$ станет равным нулю. Если условие $g_u, g_v < \sqrt{k}$ не выполняется, мы получим коэффициенты, один из которых либо равен нулю, либо больше $\sqrt{k}$. По аналогии с алгоритмом 3, в котором $u \bmod 2^W \neq 0$ и $v \bmod 2^W \neq 0$ (иначе мы просто используем быстрое деление), то есть u, v меньше $\sqrt{k} = 2^W$ и, следовательно, НОД$(u, 2^W) < 2^W$, НОД$(v, 2^W) < 2^W$.

7. Результаты экспериментов

Вышеописанные алгоритмы были реализованы с помощью комплекса программ на C++ с использованием библиотеки MPIR на компьютере с процессором Intel Core 2 Duo, в котором длина машинного слова составляет $W = 64$ бита. В табл. 1 представлены результаты работы описанных ранее алгоритмов – времени выполнения в зависимости от N, где N – количество лимбов, занимаемых в памяти, длинным числом. В данной таблице время указано относительно самого быстрого алгоритма для заданного N. Из анализа результатов, представленных в табл. 1, мы имеем, что $T(\text{ГЦ1}) > T(\text{ГЦ2}) > T(\text{ГЦ3})$ и $T(\text{ГЦ2}) > T(\text{ГЦ4})$ при любом N, где T – время работы соответствующего алгоритма. Алгоритм ГЦ3 выполняется быстрее ГЦ4 до $N = 600$, затем уже время выполнения ГЦ4 становится меньше. Таким образом, можно создать гибридный алгоритм, использующий ГЦ3 до определенной границы (в нашем случае это ≈ 600), после которой уже используется ГЦ4. Для указанной границы будем использовать термин «пороговое значение» (threshold).

Это пороговое значение можно точно вычислить для конкретного компьютера, на котором будет выполняться k-арный алгоритм.

Что же касается количественных показателей, то при $N < 1500$ метод ГЦ1 работает на 2–3% медленнее ГЦ4, а с увеличением N – на 7–19%. Таким образом, с помощью гибридного алгоритма можно получить ускорение от 3% (1/1.03) до 16% (1/1.19).

Литература

1. *Gladman B., Hart W., Moxham J., et al.* MPIR: Multiple Precision Integers and Rationals. Version 2.7.0. – 2015. – URL: http://mpir.org, свободный.

2. *Jebelean T.* A double-digit Lehmer–Euclid algorithm for finding the GCD of long integers // J. Symbol. Computation. – 1995. – V. 19, No 1–3. – P. 145–157. – doi: 10.1006/jsco.1995.1009.

3. *Stein J.* Computational problems associated with Racah algebra // J. Comput. Phys. – 1967. – V. 1, No 3. – P. 397–405. – doi: 10.1016/0021-9991(67)90047-2.

4. *Кнут Д. Э.* Искусство программирования. Т. 2. – М.: Вильямс, 2001. – 832 с.

5. *Sorenson J.* Two fast GCD algorithms // J. Algorithms. – 1994. – V. 16, No 1. – P. 110–144. – doi: 10.1006/jagm.1994.1006.

6. *Weber K.* The accelerated integer GCD algorithm // ACM Transact. Math. Software. – 1995. – V. 21, No 1. – P. 111–122. – doi: 10.1145/200979.201042.

7. *Jebelean T.* A generalization of the binary GCD algorithm // Proc. 1993 Int. Symp. on Symbolic and Algebraic Computation. – N. Y.: ACM, 1993. – P. 111–116. – doi: 10.1145/164081.164102.

8. *Shoup V.* A Computational Introduction to Number Theory and Algebra. – Cambridge: Cambridge Univ. Press, 2008. – 598 p.

9. *Jebelean T.* An algorithm for exact division // J. Symbol. Comput. – 1993. – V. 15, No 2. – P. 169–180. – doi: 10.1006/jsco.1993.1012.

10. *Ишмухаметов Ш.Т., Мубараков Б.Г., Камаль Маад Аль-Анни* Вычисление коэффициентов Безу для k-арного алгоритма нахождения НОД // Изв. вузов. Матем. – 2017. – № 11. – С. 30–38.

11. *Еникеев Р.Р.* Нахождение коэффициентов Безу с помощью расширенного обобщенного бинарного алгоритма // Информационные технологии и математическое моделирование (ИТММ-2017): Материалы XVI Междунар. конф. им. А.Ф. Терпугова. – Томск: Изд-во науч.-техн. лит., 2017. – С. 284–291.

Поступила в редакцию
10.07.18

Еникеев Разиль Радикович, ассистент кафедры системного анализа и информационных технологий

Казанский (Приволжский) федеральный университет
ул. Кремлевская, д. 18, г. Казань, 420008, Россия
E-mail: *renikeev@kpfu.ru*

ISSN 2541-7746 (Print)
ISSN 2500-2198 (Online)

**UCHENYE ZAPISKI KAZANSKOGO UNIVERSITETA.
SERIYA FIZIKO-MATEMATICHESKIE NAUKI
(Proceedings of Kazan University. Physics and Mathematics Series)**

2019, vol. 161, no. 3, pp. 393–404

doi: 10.26907/2541-7746.2019.3.393-404

Efficient Removal of Divisors in the k-ary Algorithm

R.R. Enikeev

Kazan Federal University, Kazan, 420008 Russia
E-mail: *renikeev@kpfu.ru*

Received July 10, 2018

Abstract

The k-ary algorithm is one of the most efficient methods for finding the greatest common divisor (GCD). To find GCD of u and v, we performed the k-ary reduction $t = |au + bv|/k$, where $0 < a,\ |b| \le \lceil\sqrt{k}\rceil$: $au + bv \equiv 0 \pmod k$. The reduction step is used when u and v have almost the same size. Otherwise, we appled the dmod operation $|u - cv|/2^L$, where $c = uv^{-1} \bmod 2^L$, $L = L(u) - L(v)$, $L(a)$ is a function returning the binary length of a.

We considered that k-ary algorithm works with multi-precision integers and $k = 2^{2W}$, where W is the word size. We accelerated this method by minimizing the number of divisors removal operations. We combined this procedure with a division operation by 2^{iW} and described its fast implementation. We proposed a new way to compute coefficients that allows to get rid of divisors removal before the reduction step and to produce that operation before dmod operation in 1/3 of the cases. The proposed method accelerates the main loop of the k-ary algorithm by 3–16%. The results obtained are important in many algorithms in cryptography.

Keywords: greatest common divisor, k-ary algorithm, factors removal, multi-precision integers

References

1. Gladman B., Hart W., Moxham J., et al. *MPIR: Multiple Precision Integers and Rationals. Version 2.7.0*, 2015. Available at: http://mpir.org.
2. Jebelean T. A double-digit Lehmer–Euclid algorithm for finding the GCD of long integers. *J. Symbol. Comput.*, 1995, vol. 19, nos. 1–3, pp. 145–157. doi: 10.1006/jsco.1995.1009.
3. Stein J. Computational problems associated with Racah algebra. *J. Comput. Phys.*, 1967, vol. 1, no. 3, pp. 397–405. doi: 10.1016/0021-9991(67)90047-2.
4. Knuth D.E. *Iskusstvo programmirovaniya* [The Art of Computer Programming]. Vol. 2: Seminumerical algorithms. Moscow, Vil'yams, 2001. 832 p. (In Russian)
5. Sorenson J. Two fast GCD algorithms. *J. Algorithms*, 1994, vol. 16, no. 1, pp. 110–144. doi: 10.1006/jagm.1994.1006.
6. Weber K. The accelerated integer GCD algorithm. *ACM Transact. Math. Software*, 1995, vol. 21, no. 1, pp. 111–122. doi: 10.1145/200979.201042.
7. Jebelean T. A generalization of the binary GCD algorithm. *Proc. 1993 Int. Symp. on Symbolic and Algebraic Computation*. New York, ACM, 1993. pp. 111–116. doi: 10.1145/164081.164102.
8. Shoup V. *A Computational Introduction to Number Theory and Algebra*. Cambridge, Cambridge Univ. Press, 2008. 598 p.
9. Jebelean T. An algorithm for exact division. *J. Symbol. Comput.*, 1993, vol. 15, no. 2, pp. 169–180. doi: 10.1006/jsco.1993.1012.
10. Ishmukhametov Sh.T., Mubarakov B.G., Kamal Al-Anni Maad Calculation of Bezout's coefficients for the k-ary algorithm of finding GCD. *Russ. Math.*, 2017, vol. 61, no. 11, pp. 26–33. doi: 10.3103/S1066369X17110044.
11. Enikeev R.R. Computing Bezout coefficients using extended generalized binary algorithm. *Informatsionnye tekhnologii i matematicheskoe modelirovanie (ITMM-2017): Materialy XVI Mezhdunar. konf. im. A.F. Terpugova* [Information Technology and Mathematical Modeling (ITMM-2017): Proc. XVI Int. Conf. Dedicated to A.F. Terpugov]. Tomsk, Izd. Nauchn.-Tekh. Lit., 2017, pp. 284–291. (In Russian)

⟨ ***Для цитирования:*** *Еникеев Р.Р.* Эффективное удаление делителей в k-арном алгоритме // Учен. зап. Казан. ун-та. Сер. Физ.-матем. науки. – 2019. – Т. 161, кн. 3. – С. 393–404. – doi: 10.26907/2541-7746.2019.3.393-404. ⟩

⟨ ***For citation:*** Enikeev R.R. Efficient removal of divisors in the k-ary algorithm. *Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, 2019, vol. 161, no. 3, pp. 393–404. doi: 10.26907/2541-7746.2019.3.393-404. (In Russian) ⟩