НАУЧНО-ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ ЖУРНАЛ
ШИРОКОГО ПРОФИЛЯ

**Учредитель**
Московский государственный индустриальный университет

Журнал зарегистрирован 30 декабря 2004 г. Федеральной службой по надзору за соблюдением законодательства в сфере массовых коммуникаций и охране культурного наследия. Свидетельство о регистрации ПИ № ФС 77-19295

# ИЗВЕСТИЯ МГИУ
# № 1 (6) '2007
## Информационные технологии

*Журнал выходит 4 раза в год*
ISSN 1992-5492



# В НОМЕРЕ



**Вниманию подписчиков!**
Подписка на журнал «Известия МГИУ» проводится в издательстве МГИУ
**Тел.: (495) 674-62-50. E-mail:** izdat@msiu.ru

# О пользе решения олимпиадных задач

УДК 004.021

Е.А. Роганов
roganov@msiu.ru

Бóльшая часть программ, рассматриваемых в данной работе, написана на языке Ruby, названным так в честь драгоценного камня рубина, — одном из самых молодых современных языков. Первая версия интерпретатора была обнародована создателем языка, японским программистом Юкихиро Мацумото (Yukihiro Matsumoto) в 1995 году. Официальный сайт, посвящённый языку Ruby, размещён по адресу `http://www.ruby-lang.org`, а много дополнительной полезной и интересной информации можно найти в Википедии — свободной Интернет-энциклопедии [1]. Ruby — это чрезвычайно мощный, динамический, чисто объектно-ориентированный язык, при разработке которого основное внимание было уделено удобству программирования на нём. Многие удачные идеи, использованные ранее в таких языках, как Perl, Python, Smalltalk, LISP и некоторых других, в Ruby удалось гармонично объединить. Благодаря этому язык легко изучать, на нём очень легко и приятно писать программы, а в уже написанные программы легко вносить необходимые изменения.

Интерпретатор этого языка работает достаточно медленно — в некоторых случаях Ruby-программа может выполняться в 100 или даже 200 раз дольше, чем эквивалентная откомпилированная C-программа. Может показаться, что это обстоятельство должно заставлять отказываться от использования языка Ruby при решении олимпиадных задач с жёсткими временны́ми ограничениями, но на самом деле это не так. Использование Ruby вынуждает программиста сосредоточиться на поиске *оптимального алгоритма* решения задачи, что особенно важно для роста профессионализма. Дополнительный эффект от использования Ruby в качестве языка программирования — освобождение от необходимости борьбы с чисто техническими проблемами. Например, при работе с целыми числами не нужно заботиться об их величине или реализовывать «длинную арифметику».

В работе рассматриваются три задачи, взятые из следующих источников:

- `http://www.spoj.pl` (Sphere Online Judge) — польский веб-сайт, предоставляющий возможность круглосуточной сдачи на любом из более чем 30 языков программирования почти 1200 (на момент написания данной статьи) различных задач;
- `http://acm.msu.ru` (ACM-ICPC, Moscow Subregion) — сайт Московских студенческих олимпиад по программированию, проводимых в рамках чемпионата мира Ассоциации вычислительной техники (Association for Computing Machinery),
- `http://rubyquiz.com` (Ruby Quiz) — сайт еженедельно проводимых соревнований по решению задач на языке Ruby.

При рассмотрении задач особое внимание уделяется собственно процессу поиска их решения и тому, как этот процесс способствует повышению квалификации специалиста. Обсуждение полученных при этом результатов, имеющих самостоятельное значение, приведено в заключении.

## 1. Задача о сумме квадратов

Первая из задач, которую мы рассмотрим[1], взята с сайта Sphere Online Judge [2]. Этот веб-сайт выделяется среди ряда подобных тем, что позволяет писать программы не только на таких традиционных языках, как C, C++, Java или Pascal, но и на реже используемых при решении олимпиадных задач: Python, Ruby, Lisp, Haskell. Некоторые из задач можно успешно сдать, применяя экзотические языки типа Whitespace [3], на котором любая программа состоит лишь из пробельных символов. По этой причине в МГИУ именно данный сайт пользуется особой популярностью у преподавателей и студентов — ведь решать на нём задачи можно в процессе знакомства с любым из изучаемых языков.

**Постановка задачи.** Требуется выяснить, может ли заданное неотрицательное целое число $n$ быть представлено в виде суммы квадратов двух целых чисел. Более точно, нужно написать программу, которой на стандартный вход даётся следующая информация: первая строка содержит натуральное число $c$ — количество тестовых случаев, а на каждой из последующих $c$ строк записано ровно по одному неотрицательному целому числу, не превосходящему $10^{12}$. Программа должна для каждого из тестовых случаев напечатать на стандартный выход `Yes`, если данное число может быть представлено в виде суммы квадратов двух целых чисел и `No` в противном случае. Дополнительные ограничения таковы: объём программного кода не должен превышать 50000 байт, а время работы программы — 2 секунд (см. [4]).

**«Наивный» подход.** Решение, использующее полный перебор всех возможных вариантов (с учётом того, что из условия $m^2 + n^2 = k$ вытекает $m \leqslant \sqrt{k}$ и $n \leqslant \sqrt{k}$), конечно, не может удовлетворить заданному временно́му ограничению. Приведённая программа прежде всего иллюстрирует синтаксис и возможности языка Ruby. Функция `two_sqrs?` (в Ruby имя функции или метода может заканчиваться на вопросительный знак) содержит два вложенных цикла, в которых величины $m$ и $n$ меняются от 0 до $\sqrt{k}$. Если представление числа $k$ в виде суммы двух квадратов найти удаётся, то возвращается `true`, иначе — `false`. Метод `gets` читает строку со стандартного входа, метод `to_i` преобразует строку в целое число, итератор `times` обеспечивает выполнение блока, заключённого между `do` и `end`, необходимое число раз, а `puts` печатает на стандартный вывод строку `Yes`, если метод `two_sqrs?` возвращает истину, и `No` в противном случае.

```
def two_sqrs?(k)
  lim = Math.sqrt(k).to_i
  for m in 0 .. lim
    for n in 0 .. lim
      return true if m*m+n*n == k
    end
  end
  false
end

gets.to_i.times do
  puts two_sqrs?(gets.to_i) ? "Yes" : "No"
end
```

**Начинаем думать.** Легко заметить, что чётное число $2k$ представимо в виде суммы двух квадратов тогда и только тогда, когда представимо число $k$. Это вытекает из соотношения $2(m^2 + n^2) = (m + n)^2 + (m - n)^2$, поэтому из исходного числа можно удалить все множители 2. Нечётное число может представляться нужным образом только в том

[1] Эта задача привлекла внимание автора тем, что успешных её сдач на Ruby к 17 апреля 2006 г. не было.

случае, когда одно из возводимых в квадрат слагаемых является чётным, а другое — нет. При этом $(2k)^2 + (2l+1)^2 = 4k^2 + 4l^2 + 4l + 1$. Следовательно, нечётные числа вида $4n+3$ заведомо не представимы в виде суммы квадратов двух чисел.

Мы можем модифицировать метод `two_sqrs?`, используя сделанные наблюдения. Для оценки эффективности программы включим в неё сто вызовов модифицированного метода для чисел, чуть больших миллиона. На компьютере с тактовой частотой процессора 2.4 Ghz программа работает около 30 секунд. И это для относительно небольших чисел! Ясно, что нужны какие-то дополнительные идеи, но перед этим — некоторые комментарии по языковым конструкциям, использованным в программе.

```
def two_sqrs?(k)
  return true if k < 3
  k >>= 1 while k&1 == 0
  return false if k&3 == 3
  lim = Math.sqrt(k).to_i
  for m in 0 .. lim
    for n in 0 .. lim
      return true if m*m+n*n == k
      end
  end
  false
end

1_000_100.upto(1_000_199) do |i|
  puts "#{i} #{two_sqrs?(i)}"
end
```

В языке Ruby, кроме привычных операторов `if` и `while`, можно использовать так называемые модификаторы, что и сделано в данной версии программы. Во второй строке мы возвращаем `true` для чисел 1 и 2. В третьей строке аналогичным образом записан цикл `while`, делящий число $k$ пополам до тех пор, пока оно является чётным. Четвёртая строка «обслуживает» числа вида $4n+3$, а метод `upto` класса `Integer` обеспечивает выполнение блока для всех целых чисел от 1000100 до 1000199. Блок в данном случае состоит из единственного оператора печати строки, в которой дважды используется конструкция `#{expr}`, выполняющая подстановку текущего значения величины `i` и результата вычисления выражения `two_sqrs?(i)`.

**Интернет помогает.** Автору кажется, что многие всё ещё недооценивают полезность общемировой сети Интернет в качестве источника информации. В данном случае речь идёт не о поиске какого-либо расписания или магазина с оптимальной ценой на искомый товар, а об информации, представляющей реальную научную ценность. Эффективность принципа «Если Вы что-то ищите, то попробуйте найти это в сети» будет неоднократно демонстрироваться ниже, а сейчас хочется обсудить причины весьма скептического отношения к общемировой сети.

Во-первых, Интернет быстро меняется. За последние пять лет объём открыто опубликованной, качественной и быстро находимой в сети информации по различным вопросам, имеющим отношение к математике и информатике, возрос чрезвычайно. Многие этого просто не знают. Во-вторых, основной язык сети — английский. Более того, на русском языке «полезной» в указанном выше смысле информации весьма и весьма немного. В-третьих, необходимо *уметь искать* нужную информацию. Тут нужны и опыт, и определённое знание предметной области поиска.

Итак, запустим браузер и, воспользовавшись Google (`www.google.ru`), поищем фразу `"sum of two squares"`. Среди первых нескольких ссылок — ссылка [5] на один из самых известных сетевых ресурсов по математике — Wolfram MathWorld [6]. Там решена более общая задача о количестве различных представлений заданного натурального числа $n$ в виде суммы $k$ квадратов целых чисел. В частности, $n$ представимо в виде суммы двух квадратов целых чисел тогда и только тогда, когда возможно следующее разложение $n$ на простые множители: $n = 2^{a_0} p_1^{2a_1} \dots p_n^{2a_n} q_1^{b_1} \dots q_r^{b_r}$, где $p_i$ — простые числа вида $4k+3$, а $q_i$ — простые вида $4k+1$. Иначе говоря, все множители вида $4k+3$ должны

входить в разложение числа на простые множители в чётных степенях[2].

Усовершенствуем метод `two_sqrs?`, используя найденный критерий. Для этого будем делить число $k$ (из которого предварительно удалены все множители 2) на все его нечётные делители, начиная с тройки, и вычислять степень вхождения каждого такого делителя в $k$. Данный алгоритм гарантирует простоту рассматриваемых делителей, поэтому для применения критерия представимости числа в виде суммы двух квадратов достаточно проверить, что все делители вида $4k+3$ входят в чётных степенях.

```
def two_sqrs?(k)
  return true if k < 3
  k >>= 1 while k&1 == 0
  return false if k&3 == 3
  m = Math.sqrt(k).to_i
  3.step(m,2) do |i|
    s = 0
    while k%i == 0
      k /= i
      s += 1
    end
    return false if i&3 == 3 and s&1 == 1
  end
  true
end
```

Программа, вызывающая обновлённый метод `two_sqrs?` для 100 чисел от 1000100 до 1000199 работает теперь доли секунды. По условиям задачи, однако, числа могут быть заметно больше. Заменим 1000100 на 100000000100, а 1000199 — на 100000000199. Эксперимент показывает, что время выполнения программы увеличивается до пяти с половиной секунд, что почти втрое превышает допустимую границу.

Именно теперь начинается самое интересное, потому что в случае использования, например, языка C задача уже была бы решена, а Ruby вынуждает нас продолжить искать более эффективный подход!

**Вспомним классику.** Нужно совсем немного времени для того, чтобы обнаружить основную причину недостаточно быстрой работы алгоритма, использованного в последней из реализаций метода `two_sqrs?`. Основной цикл в нём выполняется для *всех нечётных* чисел от 3 до $\sqrt{k}$, а желательно было бы делать это только для *простых чисел вида* $4k+3$.

Среди математиков-программистов «решето Эратосфена» известно гораздо лучше, чем критерий представимости числа в виде суммы квадратов, ну а тем, кто забыл этот классический способ нахождения простых чисел, быстро поможет Интернет. Приведённая программа находит и печатает массив всех простых чисел вида $4k+3$, не превосходящих величины `max`, равной в данном случае 100.

```
max = 10**2
sieve = (0 .. max).to_a
for i in 2 .. Math.sqrt(max)
  next unless sieve[i]
  (i*i).step(max, i) do |j|
    sieve[j] = nil
  end
end
p sieve.compact.select{|x| (x&3) == 3}
```

Поясним некоторые впервые встретившиеся языковые конструкции. Две звёздочки в Ruby обозначают возведение в степень, метод `to_a` превращает диапазон `(0 .. max)` в массив длины `max+1`, инициализированный числами от `0` до `max`, метод `select` выбирает из массива `sieve` числа, имеющие остаток 3 при делении на 4, а `compact` отсеивает все «вычеркнутые» элементы, которым

[2]Отметим, что этот результат легко найти и в русскоязычной части сети (см., например, [7]).

в процессе работы основной части алгоритма было присвоено значение `nil`. Результат работы программы таков: `[3, 7, 11, 19, 23, 31, 43, 47, 59, 67, 71, 79, 83]`.

Увеличим значение величины `max` до миллиона (квадратный корень из максимально возможного по условиям задачи числа) и внесём соответствующие изменения в метод `two_sqrs?`, который теперь будет использовать массив `primes`, найденный с помощью «решета Эратосфена». С целью повышения эффективности в приведённой программе используется бинарный поиск для определения нужной вырезки `primes[0..j]` из массива всех простых чисел, не превосходящих миллиона. Эксперимент показывает, что эта программа работает быстрее, но совсем не так быстро, как нам нужно!

```
def two_sqrs?(k, primes)
  return true if k < 3
  k >>= 1 while k&1 == 0
  return false if k&3 == 3
  i,j,m = 0,primes.size-1,Math.sqrt(k).to_i
  while j-i > 1
    n = (i+j)/2
    primes[n] <= m ? i = n : j = n
  end
  primes[0..j].each do |i|
    while k%i == 0
      k /= i
      return false if k%i != 0
      k /= i
    end
  end
  true
end

max = 10**6
sieve = (0 .. max).to_a
for i in 2 .. Math.sqrt(max)
  next unless sieve[i]
  (i*i).step(max, i) do |j|
    sieve[j] = nil
  end
end
primes = sieve.compact.select{|x| (x&3) == 3}

100_000_000_100.upto(100_000_000_199) do |i|
  puts "#{i} #{two_sqrs?(i,primes)}"
end
```

Небольшое умственное усилие — и мы понимаем, что надо делать. Нахождение массива `primes` занимает, как легко проверить, более двух секунд. Поэтому, если вычислить его заранее и просто включить в текст программы, то мы явно уложимся в заданный лимит времени. Итак, заменим всё, что связано с нахождением данного массива, на один единственный оператор присваивания: `primes = [3, 7, 11, 19, 23,...`

Реализовав данную идею, обнаруживаем, что программа работает всего полсекунды. Задача решена? Увы, нет! Ведь есть ещё одно ограничение — на объём исходного кода. Эта величина не должна превосходить по условию задачи 50000 байт, а у нас получилось около 300 килобайт.

Поскольку длина массива `primes` равна 39322, то на запись каждого из его элементов можно позволить себе тратить только один байт. А ведь числа в этом массиве, в основном, не маленькие. Похоже, что разместить вычисленный заранее массив в тексте программы, к сожалению, невозможно!

**Эффективные методы факторизации.** Как известно, математика — лучший друг программиста. И если однажды нам уже удалось её применить, воспользовавшись критерием представимости числа в виде суммы квадратов, то почему бы не заменить весьма древнее «решето Эратосфена» на что-либо более современное и быстрое? Легко видеть, что нам достаточно научиться всего лишь быстро находить разложение на простые множители заданного числа — факторизовать его.

Поиск слова `factorization` в сети Интернет немедленно проводит нас к ссылкам [8]

и [9], описывающим различные алгоритмы факторизации целых чисел. Автор данной статьи даже «выкачал» исходные тексты программы, реализующей один из самых эффективных алгоритмов факторизации, использующий эллиптические кривые, и убедился, что она действительно быстро разлагает на простые множители числа, десятичная запись которых состоит из сотен цифр.

Эффективные алгоритмы факторизации, однако, сложны как с теоретической точки зрения, так и с точки зрения их реализации. Более того, для «маленьких» чисел (в нашей задаче в числе не более 13 цифр) они просто проигрывают более простым методам. Из пушки по воробьям не стреляют! Итак, мы опять в тупике.

**Кульминация.** Кажется, что у нас есть только два варианта решить задачу: либо найти «хороший» способ определения для заданного числа всех его простых делителей вида $4k+3$, либо суметь достаточно плотно упаковать заранее найденный список всех простых чисел такого вида, не превосходящих миллиона. В случае успеха второй вариант даст более эффективную программу, поэтому займёмся именно им.

Итак, можно ли задать большой массив чисел, используя малое количество информации? Иногда можно. Простейший случай — массив, состоящий из одинаковых чисел. Легко воссоздать и массив чисел, представляющих собой, например, арифметическую прогрессию. Но у нас-то ситуация не такая: простые числа вида $4k + 3$ отстоят друг от друга на различные расстояния. Интересно, а каковы эти разности?

Небольшая программа, написание которой на языке Ruby требует всего нескольких секунд, позволяет предположить, что все они не превосходят 255. Ещё несколько секунд на модификацию программы и мы *доказываем*, что разности $d_i$ между соседними членами массива простых чисел вида $4k + 3$ удовлетворяют неравенству $4 \leqslant d_i \leqslant 200$. Следовательно, для хранения данных, необходимых для восстановления исходного массива (например, первое простое число и все разности), нам нужно не более 40000 байт!

Неужели задача решена? Не вполне, ибо надо ещё понять, как разместить в тексте Ruby-программы массив из чисел $d_i$ так, чтобы каждое число занимало только один байт. Ясно, что это должно быть возможно, но как? Естественный способ, когда числа просто записываются через запятую, требует до четырёх байт на один элемент. Поэтому изучаем стандартную библиотеку (лучше всего с помощью великолепной книги [10] или сетевой версии её первого издания [11]) и получаем окончательное решение, основанное на использовании директивы `__END__` и методов `pack` и `unpack`. Первая вспомогательная программа будет строить массив простых чисел, находить массив их разностей, упаковывать каждую из разностей в байт и печатать получающуюся строку на стандартный вывод.

```
max = 10**6
sieve = (0 .. max).to_a
for i in 2 .. Math.sqrt(max)
  next unless sieve[i]
  (i*i).step(max, i) do |j|
    sieve[j] = nil
  end
end
list = sieve.compact.select{|x| (x&3) == 3}
diff = []
list.inject(0){|s,x| diff << x-s; x}
puts diff.pack("C"*diff.size)
```

Вторая из вспомогательных программ, приведённая на следующей странице, заканчивается директивой `__END__`. Любой текст, размещённый в Ruby-программе после этой директивы, рассматривается как данные и может быть прочтён из потока ввода/вывода `DATA`. Перенаправив стандартный вывод программы генерации и упаковки разностей, допишем эти данные в конец файла, содержащего вторую из программ, и по-

лучим итоговый результат — программу, которую уже можно пытаться сдать на сайте[3], так как размер этой программы менее 40000 байт и работает она достаточно быстро.

```
GC.disable
str = DATA.gets
diff = str.unpack("C"*(str.size-1))
primes=[]
diff.inject(0){|s,y| primes <<s+y; s+y}

def two_sqrs?(k, primes)
  return true if k < 3
  k >>= 1 while k&1 == 0
  return false if k&3 == 3
  i,j,m = 0,primes.size-1,Math.sqrt(k).to_i
  while j-i > 1
    n = (i+j)/2
    primes[n] <= m ? i = n : j = n
  end
  primes[0..j].each do |i|
    while k%i == 0
      k /= i
      return false if k%i != 0
      k /= i
    end
  end
  true
end

gets.to_i.times do
  puts two_sqrs?(gets.to_i,primes) ? "Yes" : "No"
end

__END__
```

**Ставим рекорды.** После того как нам удалось найти решение задачи на языке Ruby, возникает естественное желание реализовать эти же идеи на языке C для того, чтобы получить самое быстрое решение данной задачи.

Первая возникающая проблема — отсутствие в языке C директивы, аналогичной директиве `__END__` языка Ruby. Эта проблема, конечно, легко решается: нужно упаковывать данные в C-строку. Поскольку все разности между последовательными простыми числами положительны, то нулевого байта мы никогда не получим, и всё, кажется, должно быть хорошо.

Пишем C-программу, эквивалентную имеющемуся Ruby-варианту, содержащую строку вида `char* diff = "..."`, а затем в текстовом редакторе просто заменяем многоточие на строку упакованных разностей, получаемую с помощью уже имеющейся у нас программы на языке Ruby.

Проверяем работоспособность получившейся программы и видим, что что-то не так. Небольшое исследование показывает: массив простых чисел почему-то восстанавливается не правильно. Легко найти и причину — в C-строках целый ряд символов трактуется специальным образом, поэтому их надо экранировать. Следовательно, необходимо внести изменения в Ruby-программу, которая создаёт упакованную строку разностей.

Даже на этом, казалось бы, совершенно тривиальном шаге нас ждут новые «приключения». Например, естественный для языка Ruby способ заменить в строке все вхождения какого-либо одного образца на другой — применить метод `gsub` класса `String`. Только почему-то вызов `gsub('\\','\\\\')` не заменяет каждый бэкслэш на два...

В конце концов программу, конечно, удаётся написать и сдать. И не просто с рекордным временем, а со временем, в несколько раз меньшим наилучшего. Целесообразно ли было тратить два дня (около 10 часов «чистого» времени) на решение этой задачи? Автор уверен в положительном ответе.

[3]Автор данной статьи был 18 апреля 2006 года первым, кто сдал на сайте Sphere Online Judge эту задачу на языке Ruby (правда, под чужим именем).

## 2. Задача о наибольшем общем делителе

Следующая задача — с сайта Московских студенческих олимпиад по программированию [12]. Этот веб-сайт заметно отличается от рассмотренного выше сайта Sphere Online Judge. Во-первых, виртуальные и интернет-турниры проводятся только несколько раз в год, а в остальное время на сайте можно лишь познакомиться с условиями задач и результатами их решения участниками турниров. Во-вторых, решения задач могут быть представлены только на языках Pascal, C/C++ и Java (правилами проведения финальных соревнований ACM-ICPC язык Pascal уже не допустим).

**Постановка задачи.** Для заданного натурального $n \leqslant 10^{250}$ найти наибольший общий делитель чисел, получаемых всевозможными перестановками цифр исходного числа (лидирующие нули допустимы). Время работы программы не должно превышать 3 секунд, а объём используемой памяти — 16 мегабайт (см. `Problem D` из [13]).

Несмотря на языковые ограничения, накладываемые правилами проведения соревнований ACM, в качестве языка программирования вы вновь возьмём Ruby. После того как задача будет решена (а нам удастся найти оптимальное решение даже несколько более общей задачи), программу легко можно будет переписать, например, на C.

**Два случая.** Ясно, что попытка решения данной задачи «в лоб» обречена на провал. Хотя алгоритм Евклида вычисления наибольшего общего делителя двух чисел является достаточно быстрым, число всевозможных перестановок цифр числа $n$ слишком велико, когда количество цифр в нём превышает 200.

Таким образом, необходимо находить какие-то математические факты, способные облегчить вычисление искомой величины. Нужно совсем немного времени для обнаружения двух принципиально различных случаев.

Если все цифры исходного числа $n$ одинаковы, то искомая функция $f(n)$ (наибольший общий делитель чисел, получаемых всевозможными перестановками цифр числа $n$), очевидно, совпадает с $n$. В противном случае в десятичной записи $n$ имеются две различные цифры $p$ и $q$. Рассмотрим перестановки цифр исходного числа, в которых $p$ и $q$ являются двумя последними цифрами. Разность этих двух чисел $n_1$ и $n_2$ равна

$$(100s + 10p + q) - (100s + 10q + p) = 10(p - q) + (q - p) = 9(p - q), \qquad (1)$$

откуда по свойствам наибольшего общего делителя следует, что $f(n) \leqslant 81$, и, более того,

$$f(n) = 2^{0..3} \cdot 3^{0..4} \cdot 5^{0..1} \cdot 7^{0..1}. \qquad (2)$$

Эта запись означает, что множитель 2 может входить в $f(n)$ в степени 0, 1, 2 или 3, множитель 3 — в степени от 0 до 4, а множители 5 и 7 — в нулевой или первой степени. Найдём необходимые и достаточные условия для этого. Напоминаем, что далее мы рассматриваем случай, когда в записи числа $n$ имеются хотя бы две различные цифры.

**Множитель $2^k$.** Если среди цифр числа $n$ есть хотя бы одна нечётная, то $f(n)$ не может делиться на 2. Если же все цифры являются чётными, то $f(n)$ на 2 делится.

Наличие среди цифр числа $n$ одной из цифр 2 или 6 делает невозможным делимость $f(n)$ на 4. Если же запись числа $n$ не содержит цифр, отличных от нулей, четвёрок и восьмёрок, то $f(n)$, очевидно, делится на 4.

Наконец, для делимости $f(n)$ на 8 необходимо и достаточно, чтобы запись числа $n$ не содержала цифр, отличных от нулей и восьмёрок.

**Множитель $3^k$.** Из признака делимости на 3 следует, что $f(n)$ делится на 3, если $s(n)$ — сумма цифр числа $n$ — делится на 3. Аналогично, из признака делимости на 9 заключаем, что $f(n)$ делится на 9, если $s(n)$ делится на 9.

Разность $9(p-q)$ чисел $n_1$ и $n_2$ не может делиться на 81, если цифры $p$ и $q$ не равны девятке и нулю. Если же запись числа $n$ не содержит цифр, отличных от девяток и нулей, то $f(n)$ делится на 81 тогда и только тогда, когда количество девяток кратно 9. Это следует из признака делимости на девять (цифры 0 и 9 в следующем равенстве могут располагаться в любом порядке):

$$n = \underbrace{9\ldots9}_{\alpha\text{ цифр}}\underbrace{0\ldots0}_{\beta\text{ цифр}} = 9\cdot\underbrace{1\ldots1}_{\alpha\text{ цифр}}\underbrace{0\ldots0}_{\beta\text{ цифр}}.$$

Таким образом, $f(n)$ делится на 81 тогда и только тогда, когда запись числа $n$ не содержит цифр, отличных от нулей и девяток, а сумма всех его цифр $s(n)$ делится на 81.

Найдём необходимые и достаточные условия делимости $f(n)$ на 27. Разность $9(p-q)$ чисел $n_1$ и $n_2$ может делиться на 27 лишь в одном из следующих трёх случаев: a) запись числа $n$ не содержит цифр, отличных от нулей, троек, шестёрок и девяток; b) она не содержит цифр, отличных от единиц, четвёрок и семёрок; c) запись не содержит цифр, отличных от двоек, пятёрок и восьмёрок.

В первом из этих случаем имеем (порядок цифр роли не играет):

$$n = \underbrace{9\ldots9}_{\alpha\text{ цифр}}\underbrace{6\ldots6}_{\beta\text{ цифр}}\underbrace{3\ldots3}_{\gamma\text{ цифр}}\underbrace{0\ldots0}_{\delta\text{ цифр}} = 3\cdot10^{\delta}\cdot\left(\underbrace{3\ldots3}_{\alpha\text{ цифр}}\underbrace{2\ldots2}_{\beta\text{ цифр}}\underbrace{1\ldots1}_{\gamma\text{ цифр}}\right). \tag{3}$$

Из признака делимости на 9 следует, что в этом случае $n$ делится на 27 тогда и только только, когда $3\alpha+2\beta+\gamma$ делится на 9, что эквивалентно тому, что сумма цифр числа $n$ делится на 27. Поскольку любая перестановка цифр сохраняет делимость числа на 9, то в этом случае и $f(n)$ делится на 27.

Во втором случае $n$ имеет следующий вид:

$$n = \underbrace{7\ldots7}_{\alpha\text{ цифр}}\underbrace{4\ldots4}_{\beta\text{ цифр}}\underbrace{1\ldots1}_{\gamma\text{ цифр}}. \tag{4}$$

Вынесем множитель 3 из разрядов, содержащих четвёрки и семёрки:

$$n = 3\cdot\left(\underbrace{2\ldots2}_{\alpha\text{ цифр}}\underbrace{1\ldots1}_{\beta\text{ цифр}}\underbrace{0\ldots0}_{\gamma\text{ цифр}}\right) + \underbrace{1\ldots1}_{\alpha+\beta+\gamma}.$$

Заметим, что если $\alpha+\beta+\gamma$ не кратно трём, то $n$ не делится даже на 3. Если же $\alpha+\beta+\gamma=3k$, то

$$n = 3\cdot\left(\underbrace{2\ldots2}_{\alpha\text{ цифр}}\underbrace{1\ldots1}_{\beta\text{ цифр}}\underbrace{0\ldots0}_{\gamma\text{ цифр}} + \underbrace{037037\ldots037037}_{k\text{ групп «037»}}\right).$$

Для делимости числа $n$ (и всех чисел, получаемых из него перестановками цифр) на 27 необходимо и достаточно, чтобы выражение, стоящее в скобках, делилось на 9. Поскольку при сложении двух находящихся там чисел переносов из разряда в разряд не происходит, то это означает, что сумма всех их цифр должна делиться на 9:

$$2\alpha+\beta+10k = 0 \mod 9.$$

Преобразуем полученное условие:

$$\begin{aligned} 2\alpha+\beta+10k &= 0 \mod 9;\\ 2\alpha+\beta+k &= 0 \mod 9;\\ 6\alpha+3\beta+(\alpha+\beta+\gamma) &= 0 \mod 27;\\ 7\alpha+4\beta+\gamma &= 0 \mod 27. \end{aligned}$$

Мы получили, что для $n$, задаваемых соотношением 4, необходимым условием делимости $f(n)$ на 27 является делимость на 27 суммы всех цифр $s(n)$ числа $n$.

В оставшемся третьем случае всё аналогично:

$$n = \underbrace{8\ldots8}_{\alpha\text{ цифр}}\underbrace{5\ldots5}_{\beta\text{ цифр}}\underbrace{2\ldots2}_{\gamma\text{ цифр}} = 3\cdot\left(\underbrace{2\ldots2}_{\alpha\text{ цифр}}\underbrace{1\ldots1}_{\beta\text{ цифр}}\underbrace{0\ldots0}_{\gamma\text{ цифр}} + \underbrace{074074\ldots074074}_{k\text{ групп «074»}}\right). \qquad (5)$$

Необходимое условие делимости числа $n$ (и всех чисел, получаемых из него перестановками цифр) на 27 в этом случае выглядит так:

$$\begin{aligned} 2\alpha+\beta+11k &= 0 \mod 9;\\ 2\alpha+\beta+2k &= 0 \mod 9;\\ 6\alpha+3\beta+2(\alpha+\beta+\gamma) &= 0 \mod 27;\\ 8\alpha+5\beta+2\gamma &= 0 \mod 27. \end{aligned}$$

Таким образом, для чисел $n$, задаваемых соотношением 5, необходимым условием делимости $f(n)$ на 27 так же, как и для двух ранее рассмотренных случаев, является делимость на 27 суммы всех цифр $s(n)$ числа $n$.

Достаточность этого условия для первого случая (когда число не содержит цифр, отличных от нулей, троек, шестёрок и девяток) следует из формулы 3. Для двух других случаев достаточность вытекает формул 4 и 5 и импликаций

$$\begin{aligned} 7\alpha+4\beta+\gamma &= 0 \mod 27 \quad\Longrightarrow\quad \alpha+\beta+\gamma = 0 \mod 3;\\ 8\alpha+5\beta+2\gamma &= 0 \mod 27 \quad\Longrightarrow\quad \alpha+\beta+\gamma = 0 \mod 3. \end{aligned}$$

**Множитель 5.** Из признака делимости на 5 вытекает, что $f(n)$ делится на 5 тогда и только тогда, когда запись числа $n$ не содержит цифр, отличных от нулей и пятёрок.

**Множитель 7.** Формула 1 показывает, что необходимым условием делимости $f(n)$ на 7 является выполнение одного из трёх следующих условий: a) запись числа $n$ не содержит цифр, отличных от нулей и семёрок; b) она не содержит цифр, отличных от единиц и восьмёрок; c) запись не содержит цифр, отличных от двоек и девяток.

Очевидно, что в первом случае $f(n)$ делится на 7, так как при этом

$$n = \underbrace{7\ldots7}_{\alpha\text{ цифр}}\underbrace{0\ldots0}_{\beta\text{ цифр}} = 7\cdot\underbrace{1\ldots1}_{\alpha\text{ цифр}}\underbrace{0\ldots0}_{\beta\text{ цифр}},$$

причём аналогичное рассуждение справедливо и для всех чисел, получаемых из $n$ произвольными перестановками цифр.

Во втором случае имеем:

$$n = \underbrace{8\ldots8}_{\alpha\text{ цифр}}\underbrace{1\ldots1}_{\beta\text{ цифр}} = \underbrace{7\ldots7}_{\alpha\text{ цифр}}\cdot 10^{\beta} + \underbrace{1\ldots1}_{\alpha+\beta\text{ цифр}}.$$

Для любого числа $m$, получаемого из $n$ перестановкой его цифр, также справедливо представление $m = 7k + r$, где запись числа $r$ содержит только единицы.

В третьем случае получаем:

$$n = \underbrace{9\ldots9}_{\alpha \text{ цифр}}\underbrace{2\ldots2}_{\beta \text{ цифр}} = \underbrace{7\ldots7}_{\alpha \text{ цифр}} \cdot 10^{\beta} + 2 \cdot \underbrace{1\ldots1}_{\alpha+\beta \text{ цифр}} .$$

Для делимости $n$ (и всех чисел, получаемых из него перестановками цифр) на 7 здесь тоже необходима и достаточна делимость на 7 числа, запись которого содержит только единицы. Выведем нужный нам признак делимости.

**Лемма.** *Число $m$, десятичная запись которого содержит только единицы, делится на 7 тогда и только тогда, когда количество цифр в $m$ кратно* 6.

**Доказательство.** Пусть $m = \underbrace{1\ldots1}_{k \text{ цифр}}$ и $\beta_i = 10^i \mod 7$, то есть $\beta_0 = 1$, $\beta_1 = 3$, $\beta_2 = 2$, $\beta_3 = 6$, $\beta_4 = 4$, $\beta_5 = 5$, $\beta_6 = 1$, $\beta_7 = 3$, $\beta_8 = 2$, $\beta_9 = 6$, $\beta_{10} = 4$, ...

Из равенства

$$m = \sum_{i=0}^{k-1} 10^i = \sum_{i=0}^{k-1} \beta_i \mod 7$$

следует, что $m$ делится на 7 только при тех значениях величины $k$, когда сумма $k$ первых элементов последовательности $\beta_k$ делится на 7. Как легко проверить, это бывает только при $k$, кратном 6.

Таким образом, $f(n)$ делится на 7 если и только если выполнено одно из следующих трёх условий: a) запись числа $n$ не содержит цифр, отличных от нулей и семёрок; b) она не содержит цифр, отличных от единиц и восьмёрок, а количество цифр в записи числа кратно 6; c) запись не содержит цифр, отличных от двоек и девяток, а количество цифр в ней кратно 6.

**Новая постановка задачи.** Проведённое исследование показывает, что нашу задачу можно обобщить следующим образом: требуется найти эффективный способ вычисления функции $f\colon \{0, 1, \ldots, 9\}_1^* \longrightarrow \mathbb{N}$, заданной на последовательностях десятичных цифр. Значение функции на последовательности (цепочке) цифр $\omega$, представляющей собой десятичную запись числа $n$, совпадает с рассматриваемой выше функцией $f(n)$.

Для решения этой задачи применима теория индуктивных функций, изложение которой содержится в книгах [14] и [15]. Напомним, что функция $f\colon X^* \to Y$ называется индуктивной, если $f(\omega \circ x)$ можно вычислить, зная $f(\omega)$ и $x$, то есть если $\exists G\colon Y \times X \to Y$ такое, что $\forall \omega \in X^*\ \forall x \in X\ f(\omega \circ x) = G(f(\omega), x)$.

Функция $F\colon X^* \to \widetilde{Y}$ называется индуктивным расширением функции $f\colon X^* \to Y$, если $F$ индуктивна и $\exists \pi\colon \widetilde{Y} \to Y$ такое, что $\forall \omega \in X^*\ f(\omega) = \pi(F(\omega))$. Основной результат теории индуктивных функций утверждает, что для любой функции на пространстве последовательностей может быть построено минимальное индуктивное расширение, что эквивалентно существованию минимального по памяти однопроходного алгоритма вычисления этой функции.

Рассмотрим простейший пример, иллюстрирующий введённые понятия. Определённая на непустых цепочках действительных чисел функция $f\colon \mathbb{R}_1^* \to \mathbb{R}$ *среднее арифметическое элементов последовательности*, очевидно, не является индуктивной. В самом деле, зная лишь её значение на некоторой цепочке и новый элемент, нельзя вычислить значение этой функции на удлинённой цепочке.

Легко видеть, что функция $F\colon \mathbb{R}_1^* \to \mathbb{R} \times \mathbb{N}$, определённая соотношением $F(\omega) = ((s(\omega), n(\omega)))$, где $\omega = a_1 a_2 \ldots a_n$, $s(\omega) = \sum_{i=1}^{n} a_i$, а $n(\omega) = |\omega|$, является индуктивным расширением исходной функции $f$, и $\pi(s, n) = s/n$. Программа, реализующая вычисление функции $f$, должна после ввода очередного элемента последовательности $x$ пересчитывать величины $s$ и $n$, увеличивая их соответственно на $x$ и на 1, а в момент достижения конца цепочки — просто печатать значение $\pi(s, n) = s/n$. На пустой цепочке $\varepsilon$ функция $F$ равна, очевидно, $(0, 0)$. Поэтому программа, реализующая вычисление значения этой функции, должна начинаться с инициализации величин $s$ и $n$ нулями.

Из формулы 2 и полученных необходимых и достаточных условий делимости $f(n)$ на числа 2, 3, 4, 5, 7, 8, 9, 27 и 81 следует, что мы фактически построили индуктивное расширение $F$ функции $f$:

$$F\colon \{0, 1, \ldots, 9\}^* \longrightarrow \{T, F\}^{10} \times \mathbb{Z}_+ \times \mathbb{Z}_+, \quad F(\omega) = (m, n, s),$$

где $m$ — битовая маска, наличие единицы в $k$-м разряде которой означает наличие цифры $k$ в последовательности $\omega$, а $n$ и $s$ — количество и сумма элементов последовательности. При этом мы можем считать, что $F$ на пустой цепочке $\varepsilon$ имеет нулевые значения ($F(\varepsilon) = (0, 0, 0)$), а отображение $G$ задаётся равенством

$$G((m, n, s), x) = (\texttt{m|1<<x}, n + 1, s + x),$$

где установка соответствующего бита маски $m$ записана «в программистском стиле».

Вычислению значения $f(w)$ по известной $F(w)$ было посвящено всё предыдущее исследование, которое показало следующее. Если в маске $m$ установлен только один бит, то $f(\omega)$ записывается, как $n$ одинаковых цифр, равных $s/n$. В противном случае мы получаем значение $f(\omega)$, перемножая несколько величин. Начинаем мы с 9, 3 или 1 в зависимости от того, делится ли $s$ на 9, 3 или лишь 1. Если в маске $m$ нет единиц в разрядах, соответствующих нечётным числам, то добавляем множитель 2. Приведённая таблица описывает условия, при которых появляются дополнительные множители 7, 5, 4, 2, 9 и 3. Этот алгоритм позволяет написать приведённую на следующей странице программу.

| **Биты маски $m$** | **Дополнительное условие** | **Множитель** |
|---|---|---|
| 0 и 7 | — | **7** |
| 1 и 8 или 2 и 9 | `n%6 == 0` | **7** |
| 0 и 5 | — | **5** |
| 0 и 8 | — | **4** |
| 0, 4; 4, 8; 0, 4, 8 | — | **2** |
| 0 и 9 | `s%81 == 0` | **9** |
| 0 и 9 | `s%81 != 0 && s%27 == 0` | **3** |
| 1 и 4; 1 и 7; 4 и 7; 1, 4, 7; 2 и 5; 2 и 8; 5 и 8; 2, 5, 8; 0 и 3; 0 и 6; 3 и 6; 3 и 9; 6 и 9; 0, 3, 6; 0, 3, 9; 0, 6, 9; 3, 6, 9; 0, 3, 6, 9; | `s%27 == 0` | **3** |

Заметим, что в МГИУ студенты-программисты знакомятся с индуктивными функциями уже на первом курсе.

```
def pi(m, n, s)
  f = s%9==0 ? 9 : s%3==0 ? 3 : 1;  f *= 2 if m&0b1010101010 == 0
  case m
  when 1, 2, 4, 8, 16, 32, 64, 128, 256, 512;    f = ((s/n).to_s)*n
  when 129;                                      f *= 7
  when 258, 516;                                 f *= 7 if n%6 == 0
  when 33;                                       f *= 5
  when 257;                                      f *= 4
  when 17, 272, 273;                             f *= 2
  when 513;                                      f *= s%81==0?9:s%27==0?3:1
  when 18, 130, 144, 146, 36, 260, 288, 292, 9,
       65, 72, 520, 576, 73, 521, 577, 584, 585; f *= 3 if s%27 == 0
  end
  f.to_s
end

m, n, s = 0, 0, 0
begin
  while true
    print "x -> "; x = readline.to_i; m, n, s = m|(1<<x), n+1, s+x
  end
rescue EOFError
  puts "\ngcd = #{pi(m, n, s)}"
end
```

Эта программа читает последовательность цифр и печатает результат, когда последовательность заканчивается (при обработке исключительной ситуации `EOFError`).

### 3. Задача о «странных» числах

Следующая задача взята с веб-сайта Ruby Quiz [16], еженедельно публикующим какую-либо проблему, решения которой на языке Ruby от всех желающих сначала принимаются, а затем тщательно анализируются. Основной целью организаторов сайта является обсуждение возможностей и особенностей языка Ruby, позволяющих получать красивые и эффективные решения различных задач.

**Постановка задачи.** «Странным» (weird) числом называется натуральное число $n$ такое, что сумма всех его делителей (исключая само $n$) больше, чем $n$, но при этом не существует такого подмножества его делителей, сумма которых равна ровно $n$. Первыми тремя такими числами являются 70, 836 и 4030. Требуется написать программу, находящую все «странные» числа, не превосходящие заданного натурального числа (см. [17]).

Мы не будем решать эту задачу. Вместо этого мы обсудим, как её можно было бы пытаться начать решать. Затем проанализируем присланные различными людьми варианты решений, отметим лучшие из них и вычленим интересную подзадачу — задачу поиска делителей числа. Далее мы рассмотрим лучшие решения этой подзадачи и сформулируем близкую к ней задачу нахождения массива списков (или массивов) делителей всех натуральных чисел, не превосходящих заданного.

**Анализ различных решений.** Начинать решение, как всегда, нужно с поиска тех фактов, которые могут помочь справиться с проблемой. В данном случае на русском языке в сети найти практически ничего полезного не удаётся, но если искать англоязычную ин-

формацию, то её более чем достаточно. Вот некоторые ссылки, которые легко находятся:

- `http://en.wikipedia.org/wiki/Weird_number`;
- `http://mathworld.wolfram.com/WeirdNumber.html`;
- `http://www.research.att.com/~njas/sequences`.

Особенно хочется отметить последнюю из этих трёх ссылок на Интернет-энциклопедию целочисленных последовательностей [18], которая чрезвычайно интересна и может оказаться полезной в самых различных ситуациях.

После знакомства с указанной информацией написать требуемую программу совсем не сложно. Из опубликованных на сайте Ruby Quiz решений [19] самыми эффективными являются программы, которые прислали Edward Faulkner и Rob Leslie. Во всех предлагаемых решениях используется метод `divisors` нахождения списка делителей числа. Обычно им расширяют класс `Integer`, но мы приводим его реализацию в виде функции.

```
def divisors(n)
  res = [1]
  2.upto(Math.sqrt(n).floor) do |i|
    res << i if n % i == 0
  end
  res.reverse.each do |i|
    res << n / i
  end
  res.uniq
end
```

При решении задачи приходится находить списки делителей для всех (точнее, достаточно многих) чисел, не превосходящих заданного. Возникает естественный вопрос: можно ли быстро найти массив массивов делителей всех чисел, не превосходящих заданного? Положительный ответ означал бы возможность улучшить решения не только задачи о «странных» числах, но и целого ряда других задач.

**Постановка новой задачи.** Требуется написать эффективную программу, находящую массив массивов делителей всех натуральных чисел, не превосходящих заданного $n$.

Задача кажется очень простой, и какое-либо её решение можно получить достаточно быстро. Но насколько это решение будет близко к оптимальному?

**Решение новой задачи.** Автор неоднократно предпринимал попытки найти решение поставленной задачи в сети, однако ему не удалось этого сделать. Может быть, оптимальное решение действительно неизвестно? Впрочем, это не так уж и важно.

Решения этой задачи, которые предлагаются подавляющим большинством людей, основаны на вполне естественной идее как-то обобщить и усовершенствовать метод `divisors` нахождения делителей заданного числа, приведённый выше. При этом мы вынуждены тем или иным способом *выбирать* делители из некоторого множества чисел.

Принципиально иная идея заключается в том, что массивы делителей можно просто *строить*, распределяя числа «по своим местам»! Автор не может гарантировать, что приведённое решение является *оптимальным*. Однако оно существенно быстрее[4], чем все известные автору другие решения данной задачи. Более того, можно даже привести некоторые соображения, показывающие, что намного быстрее решить данную задачу нельзя.

```
def divisors(max)
  list = Array.new(max){[1]}
  (2..max).step do |i|
    (i-1...max).step(i){|j| list[j].push(i)}
  end
  list
end
```

[4]Для `max=1_000_000` более, чем втрое!

**Возврат к задаче о «странных» числах.** Теперь мы можем взять одно из лучших приведённых на сайте [16] решений, модифицировать его, использовав описанную выше идею, и сравнить времена работы получающихся программ.

Как уже отмечалось выше, Edward Faulkner предложил весьма эффективное решение, которые мы и возьмём в качестве «эталонного». В модифицированную программу добавим функцию вычисления массива массивов делителей и вместо вызова метода, находящего массив делителей конкретного числа, будем просто использовать уже вычисленный массив. Нахождение и печать всех «странных» чисел до миллиона потребовало

- для «эталонной» программы: 7 минут 58 секунд;
- для модифицированной программы: 1 минуту 36 секунд.

## Заключение

Как уже отмечалось выше, основная цель данной работы — показать, насколько полезным для математика-программиста является решение олимпиадных задач. Оно способствует расширению кругозора, развитию логического мышления и навыков нахождения эффективных алгоритмов. Всё это чрезвычайно полезно и студенту, и программисту-практику, и преподавателю. Автор, конечно же, не предлагает заниматься решением олимпиадных задач в ущерб основному виду деятельности. Однако периодически уделять этому определённое время чрезвычайно полезно во всех отношениях.

Другая цель, которую ставил перед собой автор, — обратить внимание людей, всё ещё не знакомых с Ruby, на этот язык. В МГИУ с 2003 года Ruby является первым из языков, которые изучают студенты-программисты. Его используют для написания простейших программ на занятиях по информатике старшеклассники подшефных школ нашего университета. Ruby применяется сотрудниками информационно-вычислительного центра университета для генерации индивидуальных заданий по математике и информатике для студентов и слушателей факультета довузовского образования. Он же позволяет с минимальными затратами сил и времени решать многие другие задачи организации эффективного учебного процесса. Наконец, именно на Ruby (с использованием Ruby on Rails, см. [20]) реализована основная часть информационной системы, позволившей автоматизировать работу университета [21].

Многим Ruby окажется очень полезным для написания небольших скриптов, позволяющих выполнять простейшие действия по обработке различных типов данных, системному администрированию и проверке гипотез при проведении научных исследований. Те небольшие усилия, которые необходимо затратить на изучение языка, многократно окупятся впоследствии. Ведь Ruby позволит добиваться нужных результатов, затрачивая минимум сил и времени. Кроме того, писать программы на этом языке ещё и приятно!

В заключение отметим, что в процессе решения олимпиадных задач порой можно получить и новые, представляющие самостоятельный научный интерес результаты. Так, при решении рассмотренных выше задач

- построено минимальное индуктивное расширение функции $f\colon \{0..9\}_1^* \longrightarrow \mathbb{N}$, определённой на последовательностях десятичных цифр (наибольшего общего делителя чисел, получаемых всевозможными перестановками цифр последовательности), и реализован оптимальный однопроходный алгоритм её вычисления;
- предложен новый эффективный способ построения массива списков делителей всех натуральных чисел, не превосходящих заданного; данный способ может быть использован в различных алгоритмах, связанных с делителями чисел, и значительно повышает их эффективность.

Список литературы и интернет-ресурсов

[1] http://en.wikipedia.org/wiki/Ruby_programming_language.

[2] http://www.spoj.pl.

[3] http://compsoc.dur.ac.uk/whitespace.

[4] http://www.spoj.pl/problems/TWOSQRS.

[5] http://mathworld.wolfram.com/SumofSquaresFunction.html.

[6] http://mathworld.wolfram.com.

[7] http://www.mccme.ru/mmmf-lectures/books/books/books.php?book=1&page=3.

[8] http://en.wikipedia.org/wiki/Integer_factorization.

[9] http://en.wikipedia.org/wiki/Integer_factorization#Factoring_algorithms.

[10] *Thomas D.*, *Fowler C.*, *Hunt A.* Programming Ruby. The Pragmatic Programmers' Guide. Second Edition. — Dallas: The Pragmatic Bookshelf, 2004.

[11] http://www.rubycentral.com/book.

[12] http://acm.msu.ru.

[13] http://acm.msu.ru/2003/problems.doc.

[14] *Кушниренко А.Г.*, *Лебедев Г.В.* Программирование для математиков. — М.: Наука, 1988.

[15] *Роганов Е.А.* Основы информатики и программирования. — М.: МГИУ, 2001.

[16] http://rubyquiz.com.

[17] http://rubyquiz.com/quiz57.html.

[18] http://www.research.att.com/~njas/sequences.

[19] http://rubyquiz.com/quiz57_sols.zip.

[20] http://www.rubyonrails.org.

[21] http://www.main.msiu.ru.

Материал поступил в редакцию 15 января 2007 года.