Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Арифметические алгоритмы

О. А. Пестов

Киров
2011

СОДЕРЖАНИЕ

В лекции рассматриваются некоторые арифметические алгоритмы, т.е. алгоритмы над целыми числами, а именно:

- нахождение НОД;
- возведение числа в натуральную степень;
- проверка числа на простоту;
- факторизация целого числа.

1. Наибольший общий делитель

В настоящем разделе мы рассмотрим задачу нахождения наибольшего общего делителя двух целых чисел и познакомимся с некоторыми алгоритмами решения этой задачи.

1.1. Определения

Говорят, что целое число d является *делителем* целого числа u, если для некоторого целого q справедливо $u = qd$. В этом случае также говорят, что u кратно d. Например, 3, −5, 30 являются делителями числа 30, а 7 таковым не является.

Если некоторое целое число d является делителем одновременно каждого из двух заданных чисел u и v, то d называется их общим делителем. Например, общими делителями чисел 12 и 30 являются 1, 2, 6, −3 и др., а, скажем, 5 их общим делителем не является.

Среди всех общих делителей двух заданных целых чисел u и v выберем максимальный. Он называется наибольшим общим делителем этих чисел и обозначается НОД(u, v) или gcd(u, v) (сокращение от greatest common divisor). Например, НОД(12, 30) = 6 или НОД(17, 0) = 17. Очевидно, НОД определен, если, по крайней мере, одно из чисел u и v не равно нулю. Обычно полагают НОД(0, 0) = 0.

Легко видеть, что НОД(u, v) = НОД($|u|$, $|v|$), поэтому достаточно рассматривать задачу нахождения НОД для натуральных u и v.

1.2. Тривиальный алгоритм нахождения НОД

Из определения НОД следует тривиальный алгоритм его нахождения: переберем все числа от 1 до $\min(u, v)$, проверяя, являются ли они делителями u и v. Например, чтобы найти НОД(420, 24), рассмотрим все числа от 1 до

24 и найдем, что общими делителями являются 1, 2, 3, 4, 6 и 12. Наибольшее из них это 12.

```
function gcd_triv(u, v : integer) : integer;
var i, m, d : integer;
begin
  m := min(u, v);
  d := 1;
  for i := 2 to m do
    if (u mod i = 0) and (v mod i = 0) then
      d := i;
  gcd_triv := d;
end;
```

Этот алгоритм легко реализуется, но его недостатком является то, что нам необходимо проверять много вариантов. Например, если u и v порядка одного миллиарда, то придется проверить около 10^9 чисел.

Данный алгоритм можно существенно улучшить. Заметим, что если m делится на i, то m делится также на m/i. Например, если 100 делится на 5, то 100 делится на 20=100/5. Поэтому, чтобы перебрать все делители числа m, нет смысла рассматривать числа от 1 до m: достаточно перебрать числа от 1 до $\sqrt{m}$. Действительно, чтобы найти делители числа 100, достаточно рассмотреть числа от 1 до 10. И когда мы находим, что у числа 100 есть делитель 5, то это значит, что мы автоматически нашли делитель 20=100/5. При этом 10-и вариантов достаточно (в общем случае $\sqrt{m}$), т.к. не существует разложения $100 = i \times j$, где $i > 10$ и $j > 10$.

Для реализации улучшения требуется заменить цикл `for` на `while`. Это позволит перебирать делители до $\sqrt{m}$.

```
function gcd_triv_improved(u, v : integer) : integer;
var i, j, m, d : integer;
begin
  m := min(u, v);
  d := 1;
  i := 2;
  while i*i <= m do
  begin
    j := u div i;
    if (u mod i = 0) and (v mod i = 0) and (i > d) then
```

```
      d := i;
    if (u mod j = 0) and (v mod j = 0) and (j > d) then
      d := j;
    i := i + 1;
  end;
  gcd_triv_improved := d;
end;
```

В этом случае приходится на каждом шаге проверять два варианта, но само количество шагов значительно меньше. Так, если u и v порядка одного миллиарда, то придется проверить около $2\times\sqrt{10^9} \approx 64000$ чисел.

1.3. Алгоритм Евклида

В настоящем разделе мы познакомимся с намного более производительным, чем метод из раздела 1.2, способом нахождения НОД – алгоритмом Евклида, но прежде напомним еще некоторые свойства целых чисел.

Пусть u – целое, а v – натуральное число. Тогда существуют, причем единственные, целые числа q и r, такие, что

$$u = qv + r,\ 0 \le r < v. \qquad (1)$$

Правило, которое произвольной паре целых чисел u, v, где $v \neq 0$, ставит в соответствие пару целых чисел q и r, удовлетворяющих соотношениям (1), называется *операцией деления с остатком*, причем u называется *делимым*, v – *делителем*, q – *неполным частным*, а r – *остатком*. Для неполного частного q и остатка r будем использовать обозначения:

$$q = u \text{ div } v,$$
$$r = u \text{ mod } v.$$

Приведем несколько примеров:

$$\begin{array}{rlcrlrl}
14 &= 2\times5+4 & \rightarrow & 14 \text{ div } 5 &= 2, & 14 \text{ mod } 5 &= 4 \\
-14 &= (-3)\times5+1 & \rightarrow & -14 \text{ div } 5 &= -3, & -14 \text{ mod } 5 &= 1 \\
3 &= 0\times7+3 & \rightarrow & 3 \text{ div } 7 &= 0, & 3 \text{ mod } 7 &= 3 \\
-3 &= (-1)\times7+4 & \rightarrow & -3 \text{ div } 7 &= -1, & -3 \text{ mod } 7 &= 4
\end{array}$$

Алгоритм Евклида основан на последовательном использовании следующего соотношения:

$$\text{НОД}(u, v) = \text{НОД}(v, u \bmod v). \qquad (2)$$

Оно следует из (1). Ведь если u и v имеют общий делитель d, то он также является делителем остатка $r = u \bmod v$. Верно и обратное, что если v и $r = u \bmod v$ делятся на некоторое число d, то на него делится u.

Пример. Пусть, например, требуется найти НОД(88179, 5313).
Применяя (2) многократно, получаем:

НОД(88179, 5313) = НОД(5313, 88179 mod 5313) = НОД(5313, 3171) =
НОД(3171, 5313 mod 3171) = НОД(3171, 2142) =
НОД(2142, 3171 mod 2142) = НОД(2142, 1029) =
НОД(1029, 2142 mod 1029) = НОД(1029, 84) =
НОД(84, 1029 mod 84) = НОД(84, 21) =
НОД(21, 84 mod 21) = НОД(21, 0) = 21.

Заметим, что второй аргумент в выражении НОД(v, u mod v) всегда меньше первого; поэтому последним шагом алгоритма будет НОД(u, 0) = u. Используя это наблюдение и соотношение (2), мы можем привести рекурсивную реализацию:

```
function gcd_rec(u, v : integer) : integer;
begin
  if v = 0 then
    gcd_rec := u
  else
    gcd_rec := gcd_rec(v, u mod v);
end;
```

Здесь на каждом шаге (кроме, возможно, первого) выполняется неравенство $u > v$. Если на первом шаге, то есть при вызове функции, это не так, то после первого шага аргументы меняются местами и приведенное неравенство больше не нарушается. Например,

НОД(12, 30) = НОД(30, 12 mod 30) = НОД(30, 12) =
НОД(12, 30 mod 12) = НОД(12, 6) = НОД(6, 12 mod 6) = НОД(6, 0) = 6.

Нетрудно записать и нерекурсивный (итеративный) вариант реализации:

```
function gcd_iter(u, v : integer) : integer;
var tmp : integer;
begin
  while v > 0 do
  begin
    tmp := u;
    u := v;
    v := tmp mod v;
  end;
  gcd_iter:= u;
end;
```

Добавим две строчки, чтобы метод работал для любых целых (а не только неотрицательных):

```
function gcd_iter_abs(u, v : integer) : integer;
var tmp : integer;
begin
  u := abs(u);
  v := abs(v);
  while v > 0 do
  begin
    tmp := u;
    u := v;
    v := tmp mod v;
  end;
  gcd_iter_abs:= u;
end;
```

Количество шагов в алгоритме Евклида зависит от данных чисел. Но даже в худшем случае его можно оценить, как $O(\log_2(\min(u,v)))$. Если u и v порядка одного миллиарда, то придется выполнить около $\log_2(10^9) \approx 30$ шагов (сравните со сложностью тривиального алгоритма из предыдущего раздела).

1.4. Бинарный алгоритм

Существует модификация алгоритма Евклида, так называемый *бинарный алгоритм*. В его основе лежит несколько наблюдений. Первое заключается в том, что вместо соотношения (2) можно использовать:

$$\text{НОД}(u,v)=\text{НОД}(u-v,v), \text{ если } u \geq v.$$

Это соотношение, оставаясь верным, приводит к тому, что алгоритм Евклида становится медленным. Особенно это заметно когда одно из чисел значительно превосходит другое. Так для вычисления НОД(100, 2) придется выполнить 50 шагов: НОД(100, 2) = НОД(98, 2) = … = НОД(0, 2) = 2. А это уже сравнимо со сложностью тривиального алгоритма.

Однако если дополнительно учитывать четность чисел и выполнять вычитание не на каждом шаге, то можно прийти к эффективному алгоритму:

- если u и v – четные, то $\text{НОД}(u,v)=2\text{НОД}(u/2,v/2)$;
- если u – четное, v – нечетное, то $\text{НОД}(u,v)=\text{НОД}(u/2,v)$;
- если u – нечетное, v – четное, то $\text{НОД}(u,v)=\text{НОД}(u,v/2)$;
- если u и v – нечетные, $u > v$, то $\text{НОД}(u,v)=\text{НОД}(u-v,v)$;
- если u и v – нечетные, $v > u$, то $\text{НОД}(u,v)=\text{НОД}(v-u,u)$;
- если u или v равно нулю, то $\text{НОД}(u,v)=u+v$.

```
function binary_gcd(u, v : integer) : integer;
var d : integer;
begin
  d := 1;
  while (u > 0) and (v > 0) do
  begin
    if (u mod 2 = 1) and (v mod 2 = 1) then
    begin
      if u > v then u := u-v else v := v-u;
    end
    else begin
      if (u mod 2 = 0) and (v mod 2 = 0) then d := d*2;
      if (u mod 2 = 0) then u := u div 2;
      if (v mod 2 = 0) then v := v div 2;
    end;
  end;
  binary_gcd := d*(u+v);
end;
```

Важно, что если на некотором шаге алгоритма одно из чисел u или v нечетно, то на следующих итерациях u и v уже не могут стать одновременно четными. Это позволяет вынести вычисление множителя d за пределы основного цикла. Приходим к усовершенствованному алгоритму.

```
function binary_gcd2(u, v : integer) : integer;
var d : integer;
begin
  d := 1;
  while (u mod 2 = 0) and (v mod 2 = 0) do
  begin
    d := d*2; u := u div 2; v := v div 2;
  end;
  while (u > 0) and (v > 0) do
  begin
    if (u mod 2 = 0) then
      u := u div 2
    else if (v mod 2 = 0) then
      v := v div 2
    else if u > v then
      u := u-v
    else
      v := v-u;
  end;
  binary_gcd2 := d*(u+v);
end;
```

Чтобы оценить количество шагов, заметим, что, как минимум, каждый второй шаг одно из чисел уменьшается в два раза. Это приводит к верхней оценке в $2\times\left(\log_2 u+\log_2 v\right)=2\times\left(\log_2 uv\right)$ шагов. Если u и v порядка одного миллиарда, то придется выполнить, в худшем случае, около $2\times\log_2\left(10^{18}\right)\approx 120$ шагов.

Преимущество данного алгоритма заключается в типе выполняемых операций. Все действия это либо вычитание, либо деление/умножение на 2. Компьютер выполняет их гораздо быстрее, чем вычисление остатка от деления на произвольное число как в основном алгоритме Евклида. Поэтому б*о*льшее количество шагов данного алгоритма не всегда означает б*о*льшее время работы. Наиболее очевидное применение бинарного алгоритма это нахождение НОД для «длинных» чисел, которые не помещаются в стандартные типы данных. В этом случае реализовать деление с остатком

достаточно сложно и у него получается большое время работы. В то время как, вычитание и деление/умножение на 2 реализовать проще и они работают быстрее.

2. Бинарный алгоритм возведения в степень

Рассмотрим задачу возведения действительного числа в натуральную степень. Пусть a – действительное число, а n – натуральное. Требуется вычислить a^n. Прямой способ сделать это – непосредственно перемножить n значений величины a – требует $n-1$ операций умножения. Можно ли предложить более эффективный метод? В настоящем разделе мы рассмотрим, так называемый, *бинарный алгоритм возведения в степень*, который требует не более $2\times\lfloor\log_2 n\rfloor$ операций умножения[1].

Вначале рассмотрим случай, когда n – степень двойки, т.е. $n=2^k$ для некоторого целого k. Тогда для вычисления a^n достаточно выполнить всего k умножений вместо 2^k-1, требуемых при непосредственном умножении значений a. Пусть, например, $n=2^4=16$.

Формула $a^{16}=(((a^2)^2)^2)^2$ подсказывает нам следующую схему вычислений:

$t_1 \leftarrow a$ $\quad (t_1 = a^1)$;
$t_2 \leftarrow t_1 \cdot t_1$ $\quad (t_2 = a^2)$;
$t_4 \leftarrow t_2 \cdot t_2$ $\quad (t_4 = a^4)$;
$t_8 \leftarrow t_4 \cdot t_4$ $\quad (t_8 = a^8)$;
$t_{16} \leftarrow t_8 \cdot t_8$ $\quad (t_{16} = a^{16})$.

В случае произвольного n, нам поможет его двоичное представление. Пусть, например, n=19. Поскольку $19=10011_2$, то $a^{19}=a^{16+2+1}=a^{16}a^2a$. Таким образом, надо перемножить значения a^{16}, a^2, a, которые мы уже умеем вычислять. Получаем следующую схему вычисления a^{19}:

$t_1 \leftarrow a$ $\quad (t_1 = a^1)$;
$t_2 \leftarrow t_1 \cdot t_1$ $\quad (t_2 = a^2)$;
$t_4 \leftarrow t_2 \cdot t_2$ $\quad (t_4 = a^4)$;
$t_8 \leftarrow t_4 \cdot t_4$ $\quad (t_8 = a^8)$;
$t_{16} \leftarrow t_8 \cdot t_8$ $\quad (t_{16} = a^{16})$;
$p \leftarrow t_1 t_2 t_{16}$ $\quad (t_5 = a^{19})$.

Всего выполнено 6 операций умножения.

[1] Запись $\lfloor x \rfloor$ означает наибольшее целое число, меньшее либо равное x. Например $\lfloor 2{,}9 \rfloor=2$, $\lfloor 3 \rfloor=3$, $\lfloor 1{,}25 \rfloor=1$.

В следующем алгоритме формирование окончательного результата осуществляется параллельно с просмотром двоичного представления числа *n*. Мы каждый раз возводим текущее число *t* в квадрат, и проверяем бит в двоичном представлении числа *n*. Если бит равен 1, то результат умножается на число *t*.

```
function binary_power(a : real; n : integer) : real;
var p, t : real;
begin
  p := 1; {результат}
  t := a; {t принимает значения a, a^2, a^4, a^8 ...}
  while n > 0 do
  begin
    if n mod 2 = 1 then {какой бит в двоичной записи}
      p := p*t;
    n := n div 2; {переходим к следующему биту}
    t := t*t;
  end;
  binary_power := p;
end;
```

Количество итераций данного алгоритм равно числу разрядов в двоичной записи числа *n*, т.е. $\lfloor \log_2 n \rfloor$. На каждой итерации выполняется не более 2 умножений. Получаем, что общее число умножений в бинарном алгоритме возведения в степень не превосходит $2\lfloor \log_2 n \rfloor$; точное значение равно $2\lfloor \log_2 n \rfloor + nbits(n)$, где $nbits(n)$ – количество ненулевых разрядов в двоичном представлении числа n. Это значительно меньше количества умножений, $n-1$, требуемых при непосредственном вычислении a^n.

Заметим, однако, несмотря на то, что метод весьма быстрый, не для всякого *n* он использует минимально возможное количество умножений. Так, например, для $n=27$ бинарный метод использует 7 умножений, однако разложение $a^{27} = \left(\left(a^3\right)^3\right)^3$ подсказывает, что достаточно 6 умножений (каких?).

3. Проверка на простоту. Факторизация

3.1. Проверка целого числа на простоту

Пусть задано натуральное число n. Требуется определить, простое ли оно. Тривиальный алгоритм, вытекающий из определения простого числа, состоит в переборе чисел от 2 до $n-1$ и поиске среди них делителей числа n. Если делитель найден, то число n составное. Если делителя среди всех рассмотренных чисел нет, то n простое. Процедура использует не более $n-2$ пробных делений.

Этот метод можно улучшить. Для этого можно применить те же рассуждения, что и в параграфе 1.2. В качестве пробных делителей достаточно рассматривать числа, не превышающие $\sqrt{n}$. Действительно, если у n есть делитель $k > \sqrt{n}$, то его делителем будет также число $\frac{n}{k} \le \sqrt{n}$.

Мы приходим к следующему *алгоритму пробных делений:*

```
function is_prime(n : integer) : boolean;
var i : integer;
begin
  is_prime := false;
  i := 2;
  while i*i <= n do
  begin
    if (n mod i = 0) then
      exit;
    i := i+1;
  end;
  is_prime := true;
end;
```

Данный алгоритм использует не более $\sqrt{n}$ пробных делений. Трудоемкость можно сократить приблизительно в три раза. Для этого заметим, что делить на составные числа не имеет смысла. Ведь каждое из них имеет простой делитель, который был уже проверен ранее. Отсюда следует, что в оптимальном алгоритме переменная i должна пробегать все простые числа от 2 до $\sqrt{n}$. Но если мы простых чисел не знаем, то можно воспользоваться тем фактом, что все простые числа, кроме 2 и 3, имеют вид $6k \pm 1$, где k – натуральное. Поэтому в качестве пробных делителей

достаточно брать только такие числа. Мы приходим к следующему усовершенствованному *алгоритму пробных делений*:

```
function is_prime2(n : integer) : boolean;
var i : integer;
begin
  if (n = 2) or (n = 3) then
    is_prime2 := true
  else if (n mod 2 = 0) or (n mod 3 = 0) then
    is_prime2 := false
  else
  begin
    is_prime2 := true;
    i := 1;
    while (6*i-1)*(6*i-1) <= n do
    begin
      if (n mod (6*i-1) = 0) or (n mod (6*i+1) = 0) then
      begin
        is_prime2 := false;
        break;
      end;
      i := i+1;
    end;
  end;
end;
```

3.2. Таблица простых чисел. Решето Эратосфена

Рассмотрим задачу построения *списка* простых чисел. Пусть требуется найти k первых простых числа. Переберем все числа в порядке возрастания и проверим каждое из них на простоту. Для проверки будем действовать, как и в алгоритме пробных делений, но теперь в качестве пробных делителей будем рассматривать только найденные к настоящему моменту простые числа, не превышающие $\sqrt{p}$. Итерации повторяем, пока не найдем k простых чисел.

```
procedure make_primes(k : integer;
                      var primes : array of integer);
var i, j, p : integer;
    ok : boolean;
```

```
begin
  i := 0;            {количество найденных простых чисел}
  p := 1;            {последнее проверенное число}
  while i < k do
  begin
    p := p+1;        {проверяем следующее число}
    j := 1;
    ok := true;      {верно ли, что число простое}
    while (j < i) and (primes[j]*primes[j] <= p) do
    begin
      if p mod primes[j] = 0 then
      begin
        ok := false;
        break;
      end;
      j := j+1;
    end;
    if (ok) then     {если простое, то добавляем в список}
    begin
      i := i+1;
      primes[i] := p;
    end;
  end;
end;
```

Для поиска простых чисел от 1 до 1 000 000 этим алгоритмом необходимо выполнить около 15 000 000 операций.

Другой (более эффективный) метод построения списка простых чисел, называется *решето Эратосфена*. Рассмотрим его на примере. Предположим, что нам требуется найти все простые числа, не превосходящие 30. Выпишем все числа от 2 до 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Мы знаем, что 2 – простое число. Вычеркнем из списка все четные числа (кроме 2):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~

Следующее невычеркнутое число – это 3. Оно простое. Вычеркиваем все бо́льшие числа, кратные 3 (некоторые из них уже были вычеркнуты ранее):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

В полученном списке находим следующее невычеркнутое число – это 5. Оно простое. Вычеркиваем все бо́льшие числа, кратные 5 (некоторые из них уже были вычеркнуты ранее):

2 3 ~~4~~ *5* ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

Мы утверждаем, что невычеркнутыми останутся все простые числа и только они (почему?).

Рассмотрим общий случай: требуется найти все простые числа, не превосходящие n. Выписываем все числа от 1 до n и начинаем выполнять аналогичные действия: вначале вычеркиваем все числа кратные и превосходящие 2, затем кратные и превосходящие 3 и т. д. На каждой итерации необходимо найти следующее невычеркнутое число p: оно обязательно будет простым, а затем вычеркнуть все кратные ему (и превосходящие его) числа, т.е. $2p$, $3p$, $4p$ и т. д. до конца списка. Итерации достаточно продолжать, пока $p \le \sqrt{n}$. По окончании итераций список будет содержать все простые числа, не большие *n*, и только их.

Заметим, что когда рассматривается число p, достаточно вычеркивать числа, начиная с p^2 (а не $2p$), так как все составные числа, меньшие p^2, будут к тому моменту уже вычеркнуты.

В рассматриваемой ниже реализации данного метода мы используем массив логических (`boolean`) значений `b[0..n]`, причем `b[j] = true` тогда и только тогда, когда число `j` не вычеркнуто из списка (элементы `b[0]`, `b[1]` не используются).

```
procedure sieve(n : integer; var b : array of boolean);
var i, j : integer;
begin
  for i := 2 to n do     {инициализация массива}
    b[i] := true;        {ни одно число не вычеркнуто}
  i := 2;
  while i*i <= n do
  begin
```

```
      if b[i] then {если число не вычеркнуто, то простое}
      begin
        j := i*i;          {вычеркиваем числа, кратные i}
        while j <= n do
        begin
          b[j] := false;
          j := j+i;
        end;
      end;
      i := i+1;
    end;
  end;
```

Теоретическая оценка сложности алгоритма достаточно сложная. С практической точки зрения можно сказать, что для поиска простых чисел от 1 до 1 000 000 необходимо выполнить около 2 200 000 операций, что достаточно эффективно.

3.3. Факторизация целых чисел

Основная теорема арифметики говорит, что любое натуральное число, кроме 1, можно представить и, притом, единственным образом в виде:

$$n = p_1^{k_1} \times p_2^{k_2} \times \ldots \times p_s^{k_s}, \text{ где } k_i > 0,\ p_i - \text{простые},\ p_1 < p_2 < \ldots < p_s \qquad (3)$$

Здесь p_i называется *простым множителем*, а k_i – *кратностью простого множителя* p_i в разложении числа n (обратим внимание, что p_i называется также *простым делителем* числа n). Данное представление называется *разложением числа на простые множители* или *каноническим разложением*. Так, каноническое разложение числа 168 это $168 = 2^2 3^1 7^2$. То есть, $s = 3$, $p_1 = 2$, $k_1 = 2$, $p_2 = 3$, $k_2 = 1$, $p_3 = 7$, $k_3 = 2$.

В настоящем разделе рассмотрим *задачу факторизации целого числа*, другими словами, задачу поиска канонического разложения.

Применим тривиальный алгоритм. Пусть задано число $n \geq 2$. С помощью пробных делений ищем его наименьший простой делитель p. Если $p < n$, то делим n на p и повторяем поиск простых делителей для нового n. Заметим, что поиск делителя для нового n достаточно возобновить с последнего найденного p. Итерации заканчиваются, как только

устанавливается, что n простое. Оставшееся простое число также добавляется к списку делителей.

В приведенной ниже процедуре в качестве пробных делителей будем рассматривать числа 2, 3 и числа вида $6i \pm 1$, где i – натуральное.

```
{делим число n на m, изменяя массив делителей d}
procedure update(m : integer;
                 var n, k : integer;
                 var d : array of integer);
begin
  while n mod m = 0 do
  begin
    n := n div m;
    k := k+1; {увеличиваем количество делителей}
    d[k] := m; {записываем делитель в массив}
  end;
end;

{
вход - число n для разложения;
результат - количество делителей k и массив делителей d
}
procedure factor(n : integer; var k : integer;
                 var d : array of integer);
var i : integer;
begin
  k := 0; {количество делителей}
  update(2, n, k, d); {сокращаем на 2}
  update(3, n, k, d); {сокращаем на 3}

  i := 1;
  while (6*i-1)*(6*i-1) <= n do
  begin
    update(6*i-1, n, k, d); {сокращаем на 6i-1}
    update(6*i+1, n, k, d); {сокращаем на 6i+1}
    i := i+1;
  end;
  if n > 1 then
    update(n, n, k, d);{оставшееся число нужно добавить}
end;
```

4. Задания для самостоятельного решения

1. Для всех приведенных функций написать код запускающих программ, предусматривающий ввод и вывод данных. Скомпилировать и запустить программы на ряде примеров.

2. Напишите программу, которая проверяет, является ли заданное число N совершенным (равным сумме своих положительных делителей, кроме самого себя).

3. Напишите программу, которая проверяет являются ли два заданных числа N и M взаимно простыми ($1 \le N, M \le 10^5$).

4. Напишите программу, которая сокращает дробь N/M. ($-10^9 \le N, M \le 10^9$). Используйте алгоритм Евклида.

5. Разработайте методы возведения в 15 и 27 степень, требующие соответственно 5 и 6 умножений.

6. Какие из приведенных чисел являются простыми 17, 239, 17239, 23917, 170239, 239017?

7. Напишите программу нахождения всех простых чисел в диапазоне $[A \ldots B]$ ($1 \le A \le B \le 10^5$).

8. Найдите сколько простых чисел меньше ста миллионов. Используйте решето Эратосфена.

9. Напишите рекурсивный вариант бинарного метода вычисления НОД.

10. Напишите программу, которая находит, сколько различных простых делителей у числа N ($1 \le N \le 10^9$). Например, если $N = 72$, то простые делители это 2 и 3.

Литература

Арифметическим алгоритмам посвящена большая часть 2-го тома известной монографии Д. Кнута. Помимо рассмотренных в настоящей работе задач и алгоритмов рассматриваются алгоритмы генерации случайных чисел, алгоритмы арифметики произвольной точности, более совершенные алгоритмы проверки простоты и факторизации и др.

1. Кнут Д. Искусство программирования, том 2. Получисленные методы. – 3-е изд. – М.: Вильямс, 2007.

См. также работу:

2. Ноден П., Китте К. Алгебраическая алгоритмика (с упражнениями и решениями): Пер. с франц. – М.: Мир, 1999.

Некоторые алгоритмы с целыми числами рассматриваются в учебнике:

3. Шень А. Программирование: теоремы и задачи. – М.: МЦНМО, 1995.