Математичка, ты довольна?

Сапожников Денис

Contents

1 Алгоритм Евклида

1.1 Определения

Определение (НОД)**.** Наибольшим общим делителем (НОДом) двух чисел a и b называется такое число g, что $\begin{cases} a \vdots g \\ b \vdots g \\ g \text{ --- } \textit{максимальное из возможных} \end{cases}$

Обозначения: НОД(a, b), или просто (a, b).

Лемма 1. *Пусть* $(a, b) = g$. *Пусть* d *– какой-то (не обязательно наибольший) общий делитель* a *и* b. *Тогда* $g \vdots d$.

Proof. Через разложение на простые множители по ОТА. □

Определение (НОК)**.** Наименьшим общим кратным (НОКом) двух чисел a и b называется такое число l, что $\begin{cases} l \vdots a \\ l \vdots b \\ l \text{ --- } \textit{минимальное из возможных} \end{cases}$

Обозначения: НОК(a, b), или просто $[a, b]$.

Лемма 2. $(a, b) \cdot [a, b] = a \cdot b$

Proof. Через разложение на простые множители по ОТА. □

Данная лемма позволяет находит НОК по НОДу, таким образом надо научиться искать лишь НОД.

1.2 Медленный алгоритм

Лемма 3. *Пусть* $a \geq b$, *тогда* $(a, b) = (a - b, b)$.

Proof. Пусть $(a, b) = g_1, (a - b, b) = g_2$.

Докажем, что $g_2 \vdots g_1$. Если $(a, b) = g_1$, то по определению НОДа: $a \vdots g_1, b \vdots g_1$. Значит и $a - b \vdots g_1$. То есть получили, что $a - b$ и $b \vdots g_1$. По лемме 1 $g_2 \vdots g_1$.

Теперь докажем, что $g_1 \vdots g_2$. Если $(a - b, b) = g_2$, то b и $a - b \vdots g_2$. Значит и $a \vdots g_2$. Получили: a *и* $b \vdots g_2$. Значит по лемме 1 $g_1 \vdots g_2$.

Итого: $g_1 \vdots g_2$ *и* $g_2 \vdots g_1$. Значит, очевидно, $g_1 = g_2$ что и т.д. □

Лемма 3 позволяет легко находить НОД двух чисел без разложения на простые множители.

```
int gcd(int a, int b) {
  if (a == 0 || b == 0)
    return a + b;
  if (a >= b)
    return gcd(a - b, b);
  else
    return gcd(a, b - a);
}
```

По сути, мы уже доказали, что алгоритм корректный, но, увы, он долго работает, например, на тесте $(10^9, 1)$. На данном тесте будет выполняться 10^9 преобразований $(a, 1) = (a - 1, 1) = \ldots (1, 1) = (0, 1) = 1$.

1.3 Быстрый алгоритм

Идея по ускорению: заметим, что из $(a, b) = (a - b, b)$ следует, что $(a, b) = (a\%b, b)$, так как операция взятия по модулю эквивалента большому количеству вычитаний b.

Теперь алгоритм имеет вид

```
int gcd(int a, int b) {
  if (a == 0 || b == 0)
    return a + b;
  return gcd(b, a % b);
}
```

Более того, начиная с C++17 этот алгоритм есть в стандартной бибилоитеке numeric и называется gcd.

1.4 Время работы

Сейчас будет трюк. Следите за руками.

Пусть $a \geq b$. Тогда $0 \leq a\%b < b$. Что эквивалентно тому, что остатки от деления a на b лежат в интервале $\left[-\dfrac{b}{2}; \dfrac{b}{2}\right]$. Тогда после каждой итерации НОДа модулю одного из чисел уменьшается хотя бы в 2 раза.

Таким образом время работы алгоритма будет $O(\log n)$.

1.5 Упражнения

Задача 4. $(a_1, a_2, \ldots, a_n) = (a_1, (a_2, a_3, \ldots a_n))$

Задача 5. Докажите, что простых чисел бесконечно много.

Решение. Возможно, стоит посмотреть на числа вида $n! - 1$.

Задача 6. Пусть $x = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}$. Найдите количество делителей числа.

Решение. $\alpha(x) = (\alpha_1 + 1)(\alpha_2 + 1) \ldots (\alpha_k + 1)$

Задача 7. Докажите, что число $\underbrace{111\ldots1}_{n}$ для какого-то n делится на 123456789.

Решение. Посмотрим на остатки от деления первых 123456789+1 чисел вида $\underbrace{111\ldots1}_{n}$. Различных остатков не больше 123456789. Значит по принципу Дирихле найдутся два одинаковых остатка, пусть это будут $\underbrace{111\ldots1}_{k}$ и $\underbrace{111\ldots1}_{m}$, где $k > m$.

Посмотрим на число $\underbrace{111\ldots1}_{k} - \underbrace{111\ldots1}_{m}$. Оно делится на 123456789 и имеет вид $\underbrace{111\ldots1}_{k-m}\underbrace{000\ldots0}_{m}$. Поделим его на 10^m, результат будет иметь вид $\underbrace{111\ldots1}_{k-m}$ и всё ещё делиться на 123456789, так как 10 и 123456789 были взаимнопросты.

Задача 8. Научитесь вычислять $\lceil \frac{a}{b} \rceil$ без if.

Решение.

```
(a + b - 1) / b;
```

2 Решето Эратосфера

2.1 Задача

Хотим найти все простые числа от 2 до n.

2.2 Самый простой подход за $O(n\sqrt{n})$

Заметим, что все имеет смысл перебирать делители лишь до $\sqrt{x}$, так как если есть делитель больше корня - d, то будет и делитель меньше корня: $\frac{x}{d}$. Поэтому код будет такой:

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i <= n; i++)
3   for (int j = 2; j * j <= i; j++)
4     if (i % j == 0)
5       is_prime[i] = false;
```

2.3 Решение за $O(n \log n)$

Пусть если число простое, то переберём все числа, которые делятся на него, то есть $2p, 3p, 4p, \ldots$ и пометим их как составные. Ну и действительно так мы пометим все числа составными, когда в качестве p мы возьмем любой простой делитель составного числа.

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i <= n; i++)
3   if (is_prime[i])
4     for (int j = 2 * i; j <= n; j += i)
5       is_prime[j] = false;
```

Время работы. Для каждого простого числа p внутренний фор сделает $\frac{n}{p}$ итераций. Значит всего алгоритм сделает не больше чем $\sum_{i=1}^{n} \frac{n}{i} = O(n \log n)$ [1], что и т.д.

2.4 Решение за $O(n \log \log n)$

Заметим факт, что нам необходимо помечать все простые делители, начиная не с $2i$, а с i^2 по идее из пункта 1. Таким образом нужно лишь заменить строку 4 и поменять условия для строки 2, чтобы не возникало переполнений:

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i * i <= n; i++)
3   if (is_prime[i])
4     for (int j = i * i; j <= n; j += i)
5       is_prime[j] = false;
```

Почему это работает за $O(n \log \log n)$? Для доказательства используется факт, что простых чисел от 1 до n порядка $\frac{n}{\ln n}$ и они распределены примерно равномерно, а далее считается интеграл, более подробно можно прочитать на e-maxx.

[1] В целом, это известный факт, но доказывать мы его, конечно же, не будем

2.5 Решение за $O(n)$

Вместо того, чтобы считать только пометку, простое ли было число, но и минимальное простое, на которое оно делится - p_i. Ещё будем поддерживать массив простых чисел от 2 до текущего i.

Пусть у нас есть текущее i. Обновим массив p с помощью i: пройдём по всем числам вида $x_j = p_j \cdot i$ (пока $p_j \leq p_i$) и отметим для них минимальное простое как p_j.

Итого код получается очень простым и лаконичным.

```
vector<int> pr;
vector<int> p(n, 0);
for (int i = 2; i < n; i++) {
  if (p[i] == 0) {
    p[i] = i;
    pr.push_back(i);
  }
  for (int j = 0; j < pr.size() && pr[j] <= pr[i] && i * pr[j] < n; ++j)
    pr[i * pr[j]] = pr[j];
}
```

Рассмотрим произвольное число x. Из того, что оно единственным образом представляется в виде $x = p(x) \cdot y$, где $p(x)$ - минимальное простое число, на которое делится x следует, что алгоритм посетит каждое число ровно один раз. Значит время работы $O(n)$

3 Малая теорема Ферма (МТФ)

Теорема (Малая теорема Ферма). *Пусть* $a \neq 0$ *и* $p \in \mathbb{P}$[1], *тогда*

$$a^{p-1} \equiv 1 \pmod p$$

Proof. Докажем по индукции, что $a^p \equiv a \pmod p$.

База. $a = 1 : 1^p = 1$.

Переход. $(a+1)^p = a^p + C_p^1 a + \dots + 1 = p \cdot (\dots) + a^p + 1 \Rightarrow$
$\Rightarrow (a+1)^p \equiv a^p + 1 \equiv a + 1 \pmod p$, что и т.д.

Теперь мы можем написать тривиальную цепочку сравнений по модулю:

$$\begin{aligned}
a^p &\equiv a \pmod p \\
a^p - a &\equiv 0 \pmod p \\
a(a^{p-1} - 1) &\equiv 0 \pmod p \; a \neq 0, \textit{значит} \\
a^{p-1} - 1 &\equiv 0 \pmod p \\
a^{p-1} &\equiv 1 \pmod p
\end{aligned}$$

□

[1] $\mathbb{P}$ обозначает множество простых чисел

4 Обратный элемент по модулю

Определение (Обратный элемент по модулю)**.** Обратный элемент для числа a по модулю p — это такое число b, что $a \cdot b \equiv 1 \pmod{p}$. Такое число b ещё иногда обозначают как a^{-1}.

По МТФ мы можем найти обратный элемент для любого числа по простому модулю:

$$a^{p-1} = 1$$
$$a^{p-2} \cdot a = 1$$
$$b \cdot a = 1, \textit{где } b = a^{p-2}$$

4.1 Бинарное возведение в степень

Заметим, что $a^{2k} = (a^{\frac{k}{2}})^2$ и $a^{2k+1} = (a^{\frac{k}{2}})^2 \cdot a$

Но что мы сделали такой операцией? свели нашу задачу к подсчету задачи в 2 раза меньшей по k. Значит можем сводить так $\log n$ раз и радоваться жизни.

```
int bin_pow(int a, int p, int m) {
  if (p == 0)
    return 1;
  else {
    long long t = bin_pow(a, p / 2, m);
    t = t * t % m;
    if (p % 2 == 1)
      t = t * a % m;
    return t;
  }
}
```

4.2 Диофантово уравнение

$ab \equiv 1 \pmod{n}$

$ab + kn = 1$

А это просто диофантово уравнение, решив которое мы найдем искомое a.

Итоговая сложность алгоритма $O(\log n)$

При этом мы решили для всех случаев, когда $\gcd(a, n) = 1$, а для других доказали, что решений нет.

4.3 Обратные ко всем от 1 до m

Но что если нам понадобиться найти обратные ко всем числам на интервале $[1; m]$ по модулю p за $O(m)$? Оказывается, и такое можно решить!

Пусть r_i - обратный элемент для i. Докажем факт из которого будет следовать очевидный подсчет всех r_i:

$r_i \equiv -\lfloor \frac{p}{i} \rfloor r_{p \bmod i}$

Почему это верно? Совершим цепочку тривиальных преобразований:

$p \bmod i = p - \lfloor \frac{p}{i} \rfloor i$

$p \bmod i \equiv -\lfloor \frac{p}{i} \rfloor i$

Домножим обе части на произведение обратного к i и обратного к $p \bmod i$:

$r_i \equiv -\lfloor \frac{p}{i} \rfloor r_{p \bmod i}$

Итого код такой:

```
r[1] = 1;
for (int i = 2; i <= m; i++)
  r[i] = (p - r[p % i] * (p / i)) % p;
```

5 Диофантовы уравнения и расширенный алгоритм Евклида

Задача. Найти все (или одно) решения уравнения $ax + by = g$, где g — это НОД(a, b).

Пусть (x_0, y_0) – это решение уравнения $(b\%a)x + ay = g$. Научимся находить решение исходного уравнения.

По определению остатка от деления: $b \bmod a = b - \lfloor\frac{b}{a}\rfloor a$

Значит: $g = (b \bmod a)x_1 + ay_1 = (b - \lfloor\frac{b}{a}\rfloor a)x_1 + ay_1 = bx_1 + a(y_1 - \lfloor\frac{b}{a}\rfloor x_1)$

В итоге формулы для пересчёта следующие:

$$\begin{cases} x_0 = y_1 - \lfloor\frac{b}{a}\rfloor x_1 \\ y_0 = x_1 \end{cases}$$

```
int gcd(int a, int b, int &x, int &y) {
  if (a == 0) {
    x = 0, y = 1;
    return a + b;
  }
  int x1, y1;
  int g = gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return g;
}
```

А теперь давайте решим уравнение $ax + by = c$ для произвольного c.

Если $c \,\vdots\, gcd(a, b)$, то всё просто: знаем решение уравнения

$$ax_0 + by_0 = g$$

Домножим обе части на $\frac{c}{g}$

$$a\left(\frac{c}{g}x_0\right) + b\left(\frac{c}{g}y_0\right) = c$$

То есть

$$\begin{cases} x = \frac{c}{g}x_0 \\ y = \frac{c}{g}y_0 \end{cases}$$

Если же $c \not\vdots\, gcd(a, b)$, то решений, очевидно, нет, так как

Ещё важный факт, который стоит знать - это то, что все решения такого уравнения - это:

$$\begin{cases} x = x_0 + \frac{b}{g}t \\ y = y_0 - \frac{a}{g}t \end{cases} \quad t \in \mathbb{Z}$$

6 Функция Эйлера и теорема Эйлера

Задача. Найти количество чисел, взаимнопростых с n на отрезке $[1; n]$. Такая функция от n будет иметь название Функция Эйлера и обозначаться $\varphi(n)$.

Например, $\varphi(10) = 4$ (1, 3, 7, 9 взаимнопросты с 10).

О функции Эйлера следует знать следующие факты:

- $\varphi(p) = p - 1, p \in \mathbb{P}$
- $\varphi(p^a) = p^a - p^{a-1}, p \in \mathbb{P}$
- $\varphi(ab) = \varphi(a)\varphi(b)$, если НОД$(a, b) = 1$.

Тогда можно красиво посчитать функцию Эйлера, используя разложение на простые множители: $\varphi(n) = \varphi(p_1^{k_1})\varphi(p_2^{k_2})\ldots\varphi(p_t^{k_t}) = \left(p_1^{k_1} - p_1^{k_1-1}\right)\left(p_2^{k_1} - p_2^{k_1-1}\right)\ldots\left(p_t^{k_t} - p_t^{k_t-1}\right)$

Благодаря такому разложению код почти не будет отличаться от обычной факторизации:

```
int phi(int n) {
  int result = 1;
  for (int i = 2; i * i <= n; ++i)
    if (n % i == 0) {
      int degp = 1;
      while (n % i == 0)
        n /= i, degp *= i;
      result *= degp - degp / i;
    }
  if (n > 1)
    ans *= n - 1;
  return result;
}
```

Ещё один факт, который стоит знать про функцию Эйлера - это **теорема Эйлера:**

$$a^{\varphi(n)} = 1 \pmod n, \forall a \in \mathbb{N}$$

Proof. Построим граф:

$V = \{1 \le x \le n | (x, n) = 1\}, E = \{(x, ax) | x \in V\}$

Заметим, что граф - набор циклов, докажем, что все циклы одинаковой длины:

Посмотрим на то, куда отображаются цикл при домножении каждого элемента на произвольное b: он перейдет в другой цикл, а значит этот цикл той же длины. А теперь воспользуемся свойством построенного множества, что $\forall a \in V$: здесь должен быть какой-то факт, но я его забыл.

Пусть k - длина цикла, тогда $k = \varphi(n)$

Значит $1 \equiv a^{\frac{|V|}{k}} \pmod n \Leftrightarrow 1 \equiv a^{\varphi(n)} \pmod n$

□

7 Проверка на простоту Миллера-Рабина

Мы сейчас будем идти к вероятностному алгоритму проверки числа на простоту за $O(\log^3 n)$.

Лемма. *Если p – нечётное простое число и $k \geq 1$, то уравнение*

$$x^2 \equiv 1 \pmod{p^k}$$

имеет лишь 2 решения: $x = 1, x = -1$.

Proof. Это уравнение эквивалентно $(x-1)(x+1) \equiv 0 \pmod{p^k}$. Не более чем одно из чисел $x-1$ или $x+1$ может делиться на p, поскольку если они делятся оба, то и их разность (число 2) также делится на p, что невозможно. Если $x-1$ не делится на p, то $x+1$ должно делиться на p^k, значит $x = -1 \pmod{p^k}$. Второй случай аналогичный. □

Числа 1 и -1 – тривиальные квадратные корни из единицы. Теорема утверждает, что по модулю степени нечётного простого других квадратных корней из единицы не бывает. Если по некоторому модулю n они вдруг найдутся, это значит, что n – составное. Алгоритм Миллера–Рабина для проверки простоты числа, кроме проверки условия малой теоремы Ферма, проверяет ещё и это условие.

Алгоритм, получив на входе число n, сперва записывает число $n-1$ в виде $n-1 = 2^t u$, где u нечётно. Тогда, в частности, $a^{n-1} = a^{2^t u} = (a^u)^{2^t}$, и можно сперва вычислить a^u, а потом дальше возводить в квадрат t раз. Если довозводить до конца, то потом останется только проверить условие Ферма. Но по дороге, после каждого возведения в квадрат, делается ещё одна проверка: если получилась единица, а на прошлом шаге была не 1 и не -1, то найден квадратный корень из единицы.

TO-DO: напиши код.

Теорема. *Если алгоритм выдаёт результат, что число составное, то оно действительно составное, иначе оно выдаёт неверный ответ с вероятностью менее чем $\frac{1}{2^k}$.*

Proof. Легко понять часть про составное число: алгоритм выдаёт такой результат только если не выполнятся МТФ или если нашли нетривиальный корень из единицы, что противоречит условию леммы.

Теперь заметим, что если $\gcd(a, n) \neq 1$, то не выполнится условие МТФ, а значит мы рассматриваем только случай, когда $\gcd(a, n) = 1$ и алгоритм сказал, что n – простое.

Случай 1: $\exists b : \gcd(b, n) = 1, b^{n-1} \not\equiv 1 \pmod{n}$. Иными словами, n – не число Кармайкла. Тогда при каком-то выборе a, в частности при $a = b$, алгоритм найдёт подтверждение тому, что n – составное. Чтобы оценить вероятность этого события, нужно понять, сколько всего будет таких чисел b. Оказывается, что их достаточно много, не меньше половины. Сперва показывается, что множество всех остальных b – образует подгруппу, обозначим её за $\mathbb{Z}_n^\times$.

Действительно, если b и c принадлежат этому множеству, то их произведение тоже принадлежит: $(bc)^{n-1} \equiv b^{n-1}c^{n-1} \equiv 1^2 \equiv 1 \pmod{n}$. Если b принадлежит множеству и $c = b^{-1}$, то, стало быть, $bc \equiv 1 \pmod{n}$, и потому $(bc)^{n-1} \equiv 1 \pmod{n}$. Так как $b^{n-1} \equiv 1 \pmod{n}$,отсюда следует, что $c^{n-1} \equiv 1 \pmod{n}$. Далее, по теореме Лагранжа из теории групп, размер подгруппы – делитель числа элементов в группе, и потому их не больше, чем $\frac{\varphi(n)}{2}$. Стало быть, такое число b будет выбрано с вероятностью, не превышающей $\frac{1}{2}$.

Случай 2: $\forall b : \gcd(b, n) = 1 \Rightarrow b^{n-1} \equiv 1 \pmod{n}$. Или n – это число Кармайкла. Сошлёмся на результат из ТЧ: любое такое число n не может быть степенью простого числа. Пусть $n = pq, \gcd(p, q) = 1, p, q \geq 3$.

Для всякого значения a, алгоритм строит последовательность $a^u, a^{2u}, a^{4u}, \ldots, a^{2^t u}$, при этом последний элемент равен 1. Далее, если условие леммы ни на каком шаге не будет нарушено, то последовательность может или полностью состоять из единиц, или же на каком-то шаге в ней впервые встретится -1, после чего пойдут одни единицы. В этом и только в этом случае алгоритм не найдёт подтверждения тому, что n – составное.

Пусть $j \in \{0, ..., t\}$ – наибольшее число, для которого $c^{2^j u} \equiv -1 \pmod{n}$ верно для какого-то $c \in \mathbb{Z}_n^\times$. Заметим, что существует по крайней мере одна такая пара (c, j): для $j = 0$ и $c = -1$ выполняется $(-1)^{2^0 u} = -1$, поскольку u нечётно. Для **наибольшего** j рассматриваем множество:

$$X = \{x \in \mathbb{Z}_n^\times \big| x^{2^j u} \equiv \pm 1 \pmod{n}\}$$

Все значения a, для который алгоритм скажет, что число простое лежат в X. Заметим, что X – это подгруппа в $\mathbb{Z}_n^\times$. Если докажем, что $X \neq \mathbb{Z}_n^\times$ тогда мы опять получим, что размер X не превышает половины группы, а значит вероятность выбрать «плохое» a и в этом случае не превышает $\dfrac{1}{2}$.

Из $c^{2^j u} \equiv -1 \pmod{n} \Rightarrow c^{2^j u} \equiv -1 \pmod{p}$, тогда по КТО найдётся y :
$\begin{cases} y \equiv -1 & \pmod{p} \\ y \equiv 1 & \pmod{q} \end{cases} \Rightarrow \begin{cases} y^{2^j u} \equiv -1 & \pmod{p} \\ y^{2^j u} \equiv 1 & \pmod{q} \end{cases}$.

Отсюда следует, что $y^{2^j u} \neq \pm 1$, то есть $y^{2^j u} \notin X$. Теперь покажем, что $y \in \mathbb{Z}_n^\times$.
$\begin{cases} \gcd(c, n) = 1 \Rightarrow \gcd(c, p) = 1 \Rightarrow \gcd(y, p) = 1 \\ y \equiv 1 \pmod{q} \Rightarrow \gcd(y, q) = 1 \end{cases} \Rightarrow \gcd(y, n) = 1 \Rightarrow y \in \mathbb{Z}_n^\times$ □