

---

## Лекция. Введение в теоретико-числовые алгоритмы

Делимость и модульная арифметика, отношение сравнимости, бинарное возведение в степень, простые числа, наименьшее общее кратное, наибольший общий делитель, функция Эйлера, теорема Эйлера, обратный элемент по модулю

### Делимость и модульная арифметика

В этом разделе дается определение остатка от деления на натуральное число, определяется отношение сравнимости по модулю, рассматриваются свойства отношения сравнимости (правила операций над остатками) и особенности операций с остатками в языках программирования.

**Определение.** Пусть  $a$  и  $b$  – целые числа и  $b \neq 0$ . Разделить  $a$  на  $b$  с остатком – значит представить  $a$  в виде  $a = b \cdot q + r$ , где  $q, r \in \mathbb{Z}$  и  $0 \leq r < |b|$ . Число  $q$  называется неполным частным, число  $r$  – остатком от деления  $a$  на  $b$ .

**Пример.** Для  $b = 15$  получаем

$$\begin{aligned}45 &= 3 \cdot 15 + 0, 0 \leq 0 < 15; \\123 &= 8 \cdot 15 + 3, 0 \leq 3 < 15; \\-105 &= (-7) \cdot 15 + 0, 0 \leq 0 < 15 \\-169 &= (-12) \cdot 15 + 11, 0 \leq 11 < 15\end{aligned}$$

**Пример.** Для  $b = -11$  получаем

$$\begin{aligned}44 &= (-4) \cdot (-11) + 0, 0 \leq 0 < 11; \\119 &= (-10) \cdot (-11) + 9, 0 \leq 9 < 11; \\-253 &= 23 \cdot (-11) + 0, 0 \leq 0 < 11 \\-228 &= 21 \cdot (-11) + 3, 0 \leq 3 < 11\end{aligned}$$

**Важно обратить внимание.** Практически во всех часто используемых языках программирования (C/C++, Java, C#, Pascal) взятие остатка у отрицательного числа проводится следующим образом:

1. отбрасывается минус
2. берется остаток
3. к остатку приписывается минус.

Например,  $(-5 \% 3) = -2 \neq 1$ , как было бы верно с точки зрения математики.

*В Python остаток берется по математическим правилам.*

**Важно обратить внимание.** Если вам необходимо взять остаток умножения двух чисел на такое число  $M$ , что  $M$  помещается в 32-битный тип (обычно это `int`), а значение  $M * M$  уже выходит за пределы этого типа, то необходимо использовать для умножения 64-битный тип (`long` в Java, C#, `long long` в C++) для избежания переполнения в промежуточных вычислениях. Напомним информацию по целым типам данных `int` и `long long` в C++

<code>int</code>	4 байт	От -2 147 483 648 до 2 147 483 647
<code>long long</code>	8 байт	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

#### Пример.

*Неправильно*

```
int m = 100 * 1000; // 10^5, 10^5 * 10^5 == 10^10 - не влезит в int
int a = 123456789 % m, b = 98765432 % m;
int c = (a * b) % m; // (a * b) переполнилось, взятие остатка было позже
```

*Правильно*

```
int m = 100 * 1000;
long a = 123456789 % m, b = 98765432 % m;
long c = (a * b) % m; // все верно
int d = (int) ((a * b) % m); // или так
```

*Тоже правильно*

```
int c = (int) (((long)a * b) % m);
```

---

И так правильно

```
int c = (int)((a * 111 * b) % m);
```

## Отношение сравнимости

Многие свойства чисел, а также вопросы разрешимости уравнений в целых числах удобно описывать в терминах сравнений.

**Определение.** Пусть  $m \in \mathbb{N}$ ,  $m > 1$ . Целые числа  $a$  и  $b$  называются сравнимыми по модулю  $m$  (обозначается  $a \equiv b \pmod{m}$ ), если разность  $a - b$  делится на  $m$ .

**Пример.**  $25 \equiv 5 \pmod{5}$ , так как разность  $25 - 5$  делится на 5.

Отношение сравнимости обладает следующими свойствами отношения эквивалентности.

1. Рефлексивность.  $a \equiv a \pmod{m}$ ,
2. Симметричность. Если  $a \equiv b \pmod{m}$ , то  $b \equiv a \pmod{m}$
3. Транзитивность. Если  $a \equiv b \pmod{m}$  и  $b \equiv c \pmod{m}$ , то  $a \equiv c \pmod{m}$

## Свойства отношения сравнимости

Приведем некоторые свойства отношения сравнимости, которые часто используются в решении задач.

1. Сравнения можно почленно складывать
2. Сравнения можно почленно перемножать
3. Обе части сравнения и модуль можно разделить на их общий делитель
4. Обе части сравнения можно разделить на общий делитель, если он взаимно прост с модулем.

Отношение сравнимости является **отношением эквивалентности**. Отношение эквивалентности разбивает множество на **классы эквивалентности**. Для отношения сравнимости классы эквивалентности называются **классами вычетов**. Классы вычетов обозначаются  $a \pmod{m}$ .

Множество классов вычетов по модулю  $m$  обозначается  $\mathbb{Z} / m\mathbb{Z}$ , состоит ровно из  $m$  элементов и относительно операции сложения и умножения является кольцом классов вычетов по модулю  $m$ .

**Пример.** Если  $m = 2$ , то  $\mathbb{Z}/2\mathbb{Z} = \{0 \pmod{2}, 1 \pmod{2}\}$ , где  $0 \pmod{2} = 2\mathbb{Z}$  – множество всех четных,  $1 \pmod{2} = 2\mathbb{Z} + 1$  – множество всех нечетных чисел.

**Пример.**

Если  $m = 15$ , то  $\mathbb{Z}/15\mathbb{Z} = \{0 \pmod{15}, 1 \pmod{15}, \dots, 14 \pmod{15}\}$ , где  $0 \pmod{15} = \{\dots, -30, -15, 0, 15, 30, \dots\}$ ,  $1 \pmod{15} = \{\dots, -29, -14, 1, 16, 31, \dots\}$ ,  $\dots$ ,  $14 \pmod{15} = \{\dots, -16, -1, 29, 44, \dots\}$ .

Можно сказать ещё и так: при выполнении арифметических операций по модулю  $N$  любые промежуточные результаты можно заменять на их остатки по модулю  $N$ .

**Пример.**

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}$$

## Бинарное возведение в степень

Число  $A$  можно возвести в степень  $n$  за  $O(n)$  стандартным образом или за  $O(\log n)$  при помощи бинарного возведения.

Самый простой способ на примере вычисления  $5^{11}$  содержит 11 операций умножения и выглядит следующим образом:

$$5^{11} = 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5$$

Код вычисления таким итеративным способом приведен ниже:

---

```
int iterPow(int base, int power) {
    int value = 1;
    for (int i = 1; i <= power; ++i) {
        value *= base;
    }
    return value;
}
```

Сложность такого алгоритма - **O(n)**.

Приведем известные факты операции возведения числа в степень.

$$A^0 = 1$$

$$A^{(2 * k)} = (A^k)^2$$

$$A^{(2 * k + 1)} = A^{(2 * k)} * A$$

На основе этих тождеств предложим другой способ возведения в степень. Для ранее рассмотренного примера он содержит 5 операций умножения и выглядит следующим образом:

$$5^{11} = 5 * (5 * (5^2)^2)^2$$

### Идея алгоритма бинарного возведения в степень

Заметим, что для любого числа  $a$  и **чётного** числа  $n$  выполнимо очевидное тождество (следующее из ассоциативности операции умножения):

$$a^n = \left(a^{\frac{n}{2}}\right) * \left(a^{\frac{n}{2}}\right)$$

Оно и является основным в методе бинарного возведения в степень. Действительно, для чётного  $n$  мы показали, как, потратив всего одну операцию умножения, можно свести задачу к вдвое меньшей степени.

Осталось понять, что делать, если степень  $n$  **нечётна**. Здесь мы поступаем очень просто: перейдём к степени  $n - 1$ , которая будет уже чётной:

$$a^n = a^{n-1} * a$$

Ниже представлен код быстрого возведения в степень в рекурсивном варианте:

```
int binPow(int base, int power) {
    if (power == 0) return 1;
    if (power % 2 == 0) {
        half = binPow(base, power / 2);
        return half * half;
    }
    else {
        prev = binPow(base, power - 1);
        return prev * base;
    }
}
```

Один из случаев применения быстрого возведения в степень — вычисление больших степеней чисел по какому-то модулю. На деле вся модификация, которая потребуется — это преобразование операций умножения в умножение по модулю:

```
int binPow(int base, int power, int modulo) {
    if (power == 0) return 1;

    if (power % 2 == 0) {
        int half = binPow(base, power / 2, modulo);
        return half * half % modulo;
    }
    else {
        int prev = binPow(base, power - 1, modulo);
        return prev * base % modulo;
    }
}
```

---

Итак, мы фактически нашли рекуррентную формулу: от степени  $n$  мы переходим, если она чётна, к  $\frac{n}{2}$ , а иначе — к  $n - 1$ . Понятно, что всего будет не более  $2 \log n$  переходов, прежде чем мы придём к  $n = 0$  (базе рекуррентной формулы). Сложность такого алгоритма составляет  $O(\log N)$  операций. Для любого числа, представимого в 64-битном целочисленном типе — а это могут быть даже миллиарды миллиардов — будет сделано не более 130 операций!

## Простые числа

**Определение.** Натуральное число  $n$  называется **простым**, если оно имеет ровно два делителя — 1 и  $n$ . Натуральные числа, имеющие более двух делителей, называются составными, в то время как число 1 не является ни простым, ни составным числом.

**Пример.**  $14 = 2 * 7$  — составное,  $13$  — простое.

Известно, что простых чисел бесконечно много. Также известно, что плотность простых чисел убывает с увеличением ряда натуральных чисел.

Существует оценка Чебышева о распределении простых чисел: количество простых чисел на промежутке  $[1; n]$  (обозначается как  $\pi(n)$ ) растёт с увеличением  $n$  как  $\frac{n}{\ln n}$ .

### Основная теорема арифметики

Любое натуральное число  $n > 1$  представимо в виде:  $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdot \dots \cdot p_k^{\alpha_k}$ , где  $p_i$  — простые числа, причем такое представление единственно. Такой вид представления натурального числа называется каноническим разложением числа.

Данная теорема позволяет вычислить количество делителей числа:  $\tau(n) = (\alpha_1 + 1)(\alpha_2 + 1) \dots (\alpha_k + 1)$ . Данное равенство нетрудно доказать, основываясь на базовых знаниях комбинаторики. Для того, чтобы некоторое число было делителем  $n$  необходимо, чтобы в его каноническом разложении присутствовали лишь простые числа из канонического разложения числа  $n$ , а их степени были не больше, чем в каноническом разложении  $n$ . Таким образом,  $p_i$  может входить в состав делителя со степенью  $0, 1, 2, \dots, \alpha_i$ , всего  $\alpha_i + 1$  способов. Перемножив полученные выражения для всех простых делителей получим итоговую формулу.

**Пример.**

$$\tau(75) = \tau(3 * 5^2) = (1 + 1) * (1 + 2) = 6$$

Пользуясь аналогичными рассуждениями, можно получить и формулу для суммы делителей числа:  $\sigma(n) = \frac{p_1^{\alpha_1+1}-1}{p_1-1} \dots \frac{p_k^{\alpha_k+1}-1}{p_k-1}$ .

**Пример.**  $\sigma(75) = \frac{3^2-1}{3-1} * \frac{5^3-1}{5-1} = 124$

Рассмотрим простой алгоритм проверки числа на простоту. Переберем все целые числа от 2 до  $\sqrt{n}$  и для каждого числа проверим, делится ли  $n$  на текущее число. Если мы найдем хотя бы одно число, на которое делится число  $n$ , то оно не простое, иначе  $n$  простое. Таким образом, можно проверять число на простоту за  $O(\sqrt{n})$ .

Покажем, что наш алгоритм определяет все возможные делители числа. Пусть это не так и существует какой-то делитель  $d > \sqrt{n}$ , который мы не проверили. Тогда у него есть парный делитель  $\frac{n}{d} < \sqrt{n}$ , который мы проверили, а значит наш алгоритм нашел бы и делитель  $d$ .

---

```

bool isPrime(int n) {
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

```

### **Решето Эратосфена поиска всех простых чисел на промежутке [1; n].**

Рассмотрим наивный алгоритм: переберем все числа из промежутка и каждое проверим на простоту при помощи описанного выше алгоритма. Получим асимптотику  $O(n\sqrt{n})$ .

Улучшим алгоритм. Создадим массив на  $n$  элементов, в котором будем хранить *true*, если число простое и *false* в противном случае. Изначально скажем, что все числа в промежутке [2; n] простые. Перебираем числа от 2 до  $n$ . Для каждого из них переберем все числа из нашего промежутка, которые делятся на взятое число, и запишем в массиве, что они не простые. Оценим сложность получившегося алгоритма. Когда текущее число равно  $i$  мы переберем  $\frac{n}{i}$  чисел, которые делятся на число  $i$ . Таким образом, суммарно будет совершено  $\frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \dots + \frac{n}{n-1} + \frac{n}{n}$  итераций.

Наконец, еще немного улучшим наш алгоритм, превратив его в решето Эратосфена. Создадим такой же массив, сделаем для начала все числа простыми. Перебираем текущее число. Если оно составное, пропускаем его, в противном случае нужно перебрать все числа, которые делятся на него, и пометить их как составные. Более того, если текущее число равно  $p$ , то перебор можно начинать с  $p^2$  (в отличие от предыдущего способа), так как числа  $2p, 3p, 4p, \dots, (p-1)p$  были помечены составными на предыдущих итерациях алгоритма. Оценка сложности данного алгоритма -  $O(n \log \log n)$ .

```

bool prime[N];
for (int i = 2; i < N; ++i) {
    prime[i] = true;
}
for (int i = 2; i < N; ++i) {
    if (prime[i]) {
        for (long long j = 1ll * i * i; j < N; j += i) {
            prime[j] = false;
        }
    }
}

```

---

## Наибольший общий делитель и наименьшее общее кратное

### Определение.

Целое число  $d \neq 0$  называется **наибольшим общим делителем** (НОД) целых чисел  $a_1, a_2, \dots, a_k$  (обозначается  $d = \text{НОД}(a_1, a_2, \dots, a_k)$ ), если выполняются следующие условия:

- 1) Каждое из чисел  $a_1, a_2, \dots, a_k$  делится на  $d$
- 2) Если  $d_1 \neq 0$  – другой общий делитель чисел  $a_1, a_2, \dots, a_k$ , то  $d$  делится на  $d_1$ .

**Пример.**  $\text{НОД}(4, 6) = 2$ ,  $\text{НОД}(5, 6) = 1$

**Наименьшее общее кратное** (НОК) двух целых чисел  $m$  и  $n$  есть наименьшее натуральное число, которое делится на  $m$  и на  $n$  без остатка.

Наибольший общий делитель и наименьшее общее кратное взаимосвязаны следующим соотношением:  $\text{НОД}(a, b) * \text{НОК}(a, b) = a * b$ .

**Пример.**  $\text{НОД}(4, 6) = 12$ ,  $\text{НОД}(5, 6) = 30$

Целые числа называются **взаимно простыми**, если они не имеют никаких общих делителей, кроме 1.

**Пример.** 14 и 25 взаимно просты, 15 и 25 не взаимно просты, так как у них имеется общий делитель 5.

### Свойства наибольшего общего делителя

$$\text{НОД}(a, 0) = a$$

$$\text{НОД}(a, b) = \text{НОД}(a - b, b)$$

$$\text{НОД}(a, b) = \text{НОД}(a \% b, b)$$

$$\text{НОД}(a, b) = 2 * \text{НОД}(a/2, b/2), \text{ если оба числа являются четными}$$

$$\text{НОД}(a, b) = \text{НОД}\left(\frac{a}{2}, b\right), \text{ если } a - \text{четное, а второе нечетное}$$

$$\frac{\text{НОК}(a_1, a_2, \dots, a_k)}{\text{НОД}(a_1, a_2, \dots, a_k)} = a_1 * a_2 * \dots * a_k$$

НОД двух чисел Фибоначчи равен числу Фибоначчи с индексом, равным НОД индексов чисел Фибоначчи, то есть  $\text{НОД}(F_i, F_j) = F(\text{НОД}(i, j))$ .

### Алгоритм нахождения НОД

Пусть  $a$  и  $b$  — целые числа, не равные одновременно нулю.

$a = b * q_0 + r_1$ , // разделим число  $a$  на  $b$ , получим неполное частное  $q_0$  и остаток  $r_1$ .

$b = r_1 * q_1 + r_2$ , //заменяем делимое делителем, неполное частное – остатком.

...

$r_{k-2} = r_{k-1} * q_{k-1} + r_k$ , // будем продолжать итерации.

...

$r_{n-1} = r_n * q_n$  // продолжаем до тех пор, пока не получим нулевой остаток.

Тогда  $\text{НОД}(a, b)$  - наибольший общий делитель  $a$  и  $b$ , равен  $r_n$ , последнему ненулевому члену последовательности остатков.

**Корректность** этого алгоритма вытекает из следующих утверждений:

Пусть  $a = b * q + r$ , тогда  $\text{НОД}(a, b) = \text{НОД}(b, r)$ .

$\text{НОД}(r, 0) = r$  для любого ненулевого  $r$  (так как 0 делится на любое целое число).

---

### Рекурсивный алгоритм определения НОД

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

### Нерекурсивный алгоритм определения НОД

```
int gcd(int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

### Теорема о существовании и линейном представлении НОД

Для любых целых чисел  $a_1, a_2, \dots, a_k$  существует  $\text{НОД} = d$ , и его можно представить в виде линейной комбинации этих чисел:  $d = c_1 a_1 + c_2 a_2 \dots + c_k a_k$ , где  $c_i \in \mathbb{Z}$

Для двух целых чисел  $a$  и  $b$  существует  $\text{НОД} = d$ , и его можно представить в виде линейной комбинации  $d = xa + yb$ .

Приведем рекурсивный расширенный алгоритм Евклида, который возвращает не только наибольший общий делитель двух чисел, но и коэффициенты его линейного представления.

### Расширенный алгоритм Евклида

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; // База рекурсии. Если b=0, то НОД=a, 1*a+0=d
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1); //рекурсивный переход
    x = y1;
    y = x1 - (a / b) * y1;
    return d;
}
```

---

## Функция Эйлера. Теорема Эйлера. Малая теорема Ферма

**Определение.** Функция Эйлера  $\varphi(n)$  — мультипликативная функция, равная количеству чисел, меньших  $n$ , взаимно простых с  $n$ , включая единицу. Иными словами, это количество таких чисел в отрезке  $[1; n]$ , наибольший общий делитель которых с  $n$  равен единице. Рассмотрим основные свойства функции Эйлера, которые необходимы, чтобы вычислять её для любых чисел:

Если  $p$  — простое число, то  $\varphi(p) = p - 1$ .

Если  $p$  — простое,  $a$  — натуральное число, то  $\varphi(p^a) = p^a - p^{a-1}$ .

(Поскольку с числом  $p^a$  не взаимно просты только числа вида  $pk$  ( $k \in \mathbb{N}$ ), которых  $\frac{p^a}{p} = p^{a-1}$  штук.)

Если  $a$  и  $b$  — взаимно простые, то  $\varphi(ab) = \varphi(a) * \varphi(b)$  (свойство мультипликативности функции Эйлера).

Отсюда можно получить функцию Эйлера для любого  $n$  через его факторизацию:

$$\varphi(n) = \varphi(p_1^{\alpha_1})\varphi(p_2^{\alpha_2}) \dots \varphi(p_s^{\alpha_s}) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_s}\right)$$

**Примеры.**  $\varphi(3) = 2$ ,  $\varphi(8) = 8 - 4 = 4$ ,  $\varphi(35) = \varphi(7) * \varphi(5) = 6 * 4 = 24$

$\varphi(18) = \varphi(2) * \varphi(9) = 1 * (9 - 3) = 6$

**Теорема Эйлера.** Если  $\text{НОД}(a, m) = 1$ , то  $a^{\varphi(m)} \equiv 1 \pmod{m}$ , где  $\varphi(m)$  — функция Эйлера.

Важным следствием теоремы Эйлера для случая простого модуля является **малая теорема Ферма**:

$$\text{если } \text{НОД}(a, p) = 1 \text{ и } p - \text{простое, то } a^{p-1} \equiv 1 \pmod{p}$$

**Пример.**  $18^{\varphi(25)} = 18^{20} \equiv 1 \pmod{25}$ ,  $18^4 = 18^{5-1} \equiv 1 \pmod{5}$

### Обратный элемент по модулю

**Определение.** Обратным к числу  $a$  по модулю  $m$  называется такое число  $b$ , что:

$$a * b \equiv 1 \pmod{m}$$

и обозначается как  $a^{-1}$ .

Пусть задан некоторый натуральный модуль  $m$ . Рассмотрим кольцо, образуемое этим модулем (т.е. состоящее из чисел от 0 до  $m - 1$ ). Тогда для некоторых элементов этого кольца можно найти **обратный элемент**. Утверждается, что обратный существует только для тех элементов  $a$ , которые **взаимно просты** с модулем  $m$ .

Вычисление с помощью расширенного алгоритма Евклида.

$a * b \equiv 1 \pmod{m}$  //  $b$  — обратный по модулю.

$a * b = -y * m + 1$  // по определению отношения сравнимости

$$a * b + m * y = 1$$

Реализация кода алгоритма поиска обратного по модулю:

---

```
int x, y;  
int g = gcdex(a, m, x, y);  
if (g != 1)  
    cout << "no solution";  
else {  
    x = (x % m + m) % m;  
    cout << x;  
}
```

Вычисление обратного по модулю при помощи теоремы Эйлера.

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

$$a * a^{\varphi(m)-1} \equiv 1 \pmod{m}$$

$$b \equiv a^{\varphi(m)-1} \pmod{m}$$

**Пример.**  $a = 3, m = 5, a^{m-2} = 27 \equiv 2 \pmod{m}$ .