

Теория чисел

Алгоритм Евклида, решето Эратосфена, факторизация, модульная арифметика.

Версия от 17.10.2025

Содержание

1	Немного теории чисел	3
1.1	Базовые обозначения	3
1.2	Простые числа	3
2	Алгоритм Евклида	3
2.1	Классический алгоритм для НОД	3
2.2	Реализация на Python	4
2.3	Пример руками	4
2.4	Оценка сложности	4
3	Расширенный алгоритм Евклида	4
3.1	Реализация	5
4	Модульная арифметика и быстрое возведение в степень	5
4.1	Что такое «по модулю»	5
4.2	Обратные элементы по модулю	6
5	Как найти обратное число по модулю	6
5.1	Быстрое возведение в степень по модулю	7
5.2	Малая теорема Ферма	8
6	Малая теорема Ферма: интуитивное объяснение	9
7	Решето Эратосфена	10
7.1	Идея алгоритма	10
7.2	Реализация	11
7.3	Сколько это работает	11

8	Факторизация чисел	11
8.1	Наивный метод: простое деление	11
8.2	Зачем нам всё это	12
8.3	Решение задач по пройденным темам	13

1 Немного теории чисел

Всю дорогу будем работать с целыми числами.

1.1 Базовые обозначения

- \mathbb{Z} — множество всех целых чисел.
- \mathbb{N} — множество натуральных чисел (будем считать, что это $1, 2, 3, \dots$).
- $a \mid b$ — « a делит b », то есть $b = ak$ для некоторого целого k .
- $\gcd(a, b)$ — наибольший общий делитель чисел a и b .
- $a \equiv b \pmod{m}$ — числа a и b дают один и тот же остаток при делении на m .

Чуть-чуть про асимптотику (чтобы не гадать «быстро ли это работает»):

- $O(1)$ — время почти не зависит от входа (константное время);
- $O(\log n)$ — растёт медленно, логарифмически;
- $O(n)$ — время пропорционально размеру входа (сканируем массив, строку и т.п.);
- $O(n \log n)$ — типичная сложность «хороших» сортировок;
- $O(n^2)$ и больше — уже может тормозить, если n велико.

1.2 Простые числа

Напоминание: **простое** число $p > 1$ — это такое p , у которого нет делителей, кроме 1 и самого p .

Факты, которые будем использовать без доказательства:

- Любое $n > 1$ единственным образом раскладывается в произведение простых чисел (с точностью до порядка множителей).
- Простых чисел бесконечно много.

Дальше нам нужны инструменты, как с этим всем работать алгоритмически.

2 Алгоритм Евклида

2.1 Классический алгоритм для НОД

Задача: по двум целым a и b найти $\gcd(a, b)$.

Лемма 2.1. Для любых целых a, b (не обоих нулевых) верно:

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

Идея. Пусть $a = qb + r$, где $r = a \bmod b$.

- Если число d делит и a , и b , то оно делит и $r = a - qb$.

- Если d делит b и r , то оно делит и $a = qb + r$.

Значит, множества общих делителей (a, b) и (b, r) совпадают, а значит и их НОД одинаков. \square

Из леммы следует простой алгоритм: пока второй аргумент не стал нулём, заменяем пару (a, b) на $(b, a \bmod b)$.

2.2 Реализация на Python

Algorithm 1 Алгоритм Евклида

```
1 def gcd(a: int, b: int) -> int:
2     while b != 0:
3         a, b = b, a % b
4     return abs(a)
```

В стандартной библиотеке есть `math.gcd`, но лучше один раз написать самому, чтобы понимать, что происходит.

2.3 Пример руками

Найдём $\gcd(252, 105)$.

$$252 = 105 \cdot 2 + 42,$$

$$105 = 42 \cdot 2 + 21,$$

$$42 = 21 \cdot 2 + 0.$$

Последний ненулевой остаток — 21, значит $\gcd(252, 105) = 21$.

2.4 Оценка сложности

Если совсем грубо, то количество шагов алгоритма Евклида порядка $O(\log \min(a, b))$.

То есть даже для огромных чисел алгоритм будет работать очень быстро по сравнению, например, с «перебором всех делителей».

3 Расширенный алгоритм Евклида

Здесь та же идея, но бонусом хотим найти коэффициенты в линейной комбинации.

Определение 3.1. Коэффициенты Безу для чисел a и b — это такие целые x, y , что

$$ax + by = \gcd(a, b).$$

Расширенный алгоритм Евклида как раз и возвращает НОД и эти самые x, y .

3.1 Реализация

Algorithm 2 Расширенный алгоритм Евклида

```
1 from typing import Tuple
2
3 def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
4     """
5     (d, x, y), d = gcd(a, b) ax + by = d.
6     """
7     if b == 0:
8         # gcd(a, 0) = |a|
9         return abs(a), 1 if a >= 0 else -1, 0
10
11     d, x1, y1 = extended_gcd(b, a % b)
12     # a = b * (a // b) + (a % b)
13     # d = x1 * b + y1 * (a % b)
14     x = y1
15     y = x1 - (a // b) * y1
16     return d, x, y
```

Эта функция нам пригодится для модульной арифметики.

4 Модульная арифметика и быстрое возведение в степень

4.1 Что такое «по модулю»

Говорим, что $a \equiv b \pmod{m}$, если m делит разность $a - b$. То есть a и b дают одинаковый остаток при делении на m .

Вычеты по модулю m ведут себя привычно:

$$a \equiv b \pmod{m}, \quad c \equiv d \pmod{m} \Rightarrow \begin{cases} a + c \equiv b + d \pmod{m}, \\ ac \equiv bd \pmod{m}. \end{cases}$$

Часто удобнее сразу работать только с остатками: хранить числа в диапазоне от 0 до $m - 1$ и всё время брать $\% m$.

4.2 Обратные элементы по модулю

Если $\gcd(a, m) = 1$, то существует такое x , что

$$ax \equiv 1 \pmod{m}.$$

Число x называется *обратным* к a по модулю m .

Через расширенный алгоритм Евклида это делается очень быстро.

```
1 def mod_inverse(a: int, m: int) -> int | None:
2
3     d, x, _ = extended_gcd(a, m)
4     if d != 1:
5         return None
6     return x % m
```

5 Как найти обратное число по модулю

Пусть нам дано число a и модуль m . Мы хотим найти такое число x , что

$$ax \equiv 1 \pmod{m}.$$

Число x и называется **обратным к a по модулю m** . Иными словами, мы хотим найти “деление по модулю”:

$$x = a^{-1} \pmod{m}.$$

Когда обратный элемент существует

Обратный существует только тогда, когда

$$\gcd(a, m) = 1.$$

Это легко понять: если a и m имеют общий делитель $d > 1$, то левая часть ax всегда будет делиться на d , а правая часть (1) — нет. Значит, уравнение невозможно.

Как найти обратный элемент

Есть два основных способа.

1. Через расширенный алгоритм Евклида

Расширенный алгоритм Евклида находит такие числа x и y , что

$$ax + my = \gcd(a, m).$$

Если $\gcd(a, m) = 1$, это принимает вид

$$ax + my = 1.$$

Переходим к классам по модулю m . Второй член пропадает:

$$ax \equiv 1 \pmod{m}.$$

Но это ровно то, что мы ищем! Значит, x — и есть обратный элемент (его нужно привести по модулю m).

Например, хотим найти обратный элемент к $a = 7$ по модулю $m = 26$. Расширенный алгоритм Евклида даёт:

$$7 \cdot 15 + 26 \cdot (-4) = 1.$$

Отсюда:

$$7 \cdot 15 \equiv 1 \pmod{26},$$

значит

$$7^{-1} \equiv 15 \pmod{26}.$$

5.1 Быстрое возведение в степень по модулю

Часто требуется вычислить величину $a^e \bmod m$, причём показатель e может быть очень большим. Наивный подход — перемножать a само на себя e раз — работает слишком медленно. Вместо этого используется простой приём, основанный на разборе чётности показателя.

Заметим, что для любого e справедливо:

- если e чётное, то

$$a^e = a^{2 \cdot (e/2)} = (a^2)^{e/2};$$

- если e нечётное, то

$$a^e = a \cdot a^{e-1},$$

причём показатель $e - 1$ уже чётный.

Тем самым алгоритм выглядит так:

1. пока $e > 0$:

- если e нечётное, умножаем результат на a и уменьшаем показатель на 1;

- если e чётное, заменяем a на $a^2 \bmod m$ и делим e пополам.

Так как на каждом шаге показатель либо уменьшается на 1, либо делится на 2, время работы алгоритма составляет порядка $\mathcal{O}(\log e)$.

Algorithm 3 Быстрое возведение в степень по модулю

```

1 def pow_mod(a, e, m):
2     a %= m
3     result = 1
4
5     while e > 0:
6         if e % 2 == 1:
7             result = (result * a) % m
8
9         a = (a * a) % m
10        e //= 2
11
12    return result

```

В Python есть встроенная функция `pow(base, exponent, modulus)`, которая делает примерно то же самое, так что в реальном коде имеет смысл пользоваться именно ей.

5.2 Малая теорема Ферма

Теорема 5.1 (Малая теорема Ферма). *Если p — простое число и $p \nmid a$, то*

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

Эквивалентная формулировка:

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

для любого целого a .

Идея доказательства: если взять все ненулевые остатки по модулю p , умножить их на a и посмотреть на произведение, получится та же самая совокупность классов, только переставленная. Отсюда и вылезает равенство.

Практическое применение: можно считать обратный элемент по модулю простого числа p как

$$a^{-1} \equiv a^{p-2} \pmod{p}.$$

То есть `pow(a, p-2, p)`.

6 Малая теорема Ферма: интуитивное объяснение

Рассмотрим простое число p и целое число a , которое не делится на p . Малая теорема Ферма утверждает:

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

Чтобы понять, почему это работает, посмотрим на всё “на пальцах”.

Идея через остатки

Возьмём все ненулевые остатки по модулю p :

$$1, 2, 3, \dots, p-1.$$

Теперь умножим каждый из них на число a :

$$a, 2a, 3a, \dots, (p-1)a.$$

Рассмотрим эти величины по модулю p . Оказывается, при делении на p они дают *те же самые* остатки $1, 2, \dots, p-1$, только в другом порядке.

Почему? Если бы два разных числа давали один и тот же остаток:

$$ka \equiv la \pmod{p},$$

то можно было бы сократить на a (ведь оно не делится на p), и получить:

$$k \equiv l \pmod{p},$$

что невозможно. Значит, умножение на a просто переставляет все остатки.

Смотрим на произведения

Произведение исходных остатков:

$$1 \cdot 2 \cdot 3 \cdots (p-1).$$

Произведение новых остатков:

$$a \cdot 2a \cdot 3a \cdots (p-1)a = a^{p-1}(1 \cdot 2 \cdot 3 \cdots (p-1)).$$

А так как множества остатков одинаковые (просто в другом порядке), их произве-

дения по модулю p совпадают:

$$a^{p-1}(1 \cdot 2 \cdot \dots \cdot (p-1)) \equiv 1 \cdot 2 \cdot \dots \cdot (p-1) \pmod{p}.$$

Число $1 \cdot 2 \cdot \dots \cdot (p-1)$ не делится на p , поэтому его можно сократить, получая итог:

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

Пример для наглядности

Пусть $p = 7$, возьмём $a = 3$. Считаем степени по модулю 7:

$$3^1 \equiv 3,$$

$$3^2 \equiv 9 \equiv 2,$$

$$3^3 \equiv 6,$$

$$3^4 \equiv 18 \equiv 4,$$

$$3^5 \equiv 12 \equiv 5,$$

$$3^6 \equiv 15 \equiv 1.$$

Получили:

$$3^6 \equiv 1 \pmod{7},$$

что полностью совпадает с формулой $3^{p-1} = 3^6$.

Практическое применение

Если модуль p — простое число, то мы можем находить обратный элемент:

$$a^{-1} \equiv a^{p-2} \pmod{p},$$

и это вычисляется с помощью быстрого возведения в степень.

7 Решето Эратосфена

Теперь хотим уметь быстро находить все простые числа до заданного N .

7.1 Идея алгоритма

1. Создаём булев массив `is_prime` размером $N + 1$ и считаем, что все числа от 2 до N — простые.
2. Идём по p от 2 до $\lfloor \sqrt{N} \rfloor$. Если `is_prime[p]` всё ещё `True`, вычёркиваем все кратные p , начиная с p^2 .

3. В конце все индексы, помеченные как `True`, — это простые числа.

Почему начинаем с p^2 : всё, что меньше, уже вычеркнули на предыдущих шагах.

[Визуализация алгоритма](#)

7.2 Реализация

Algorithm 4 Решето Эратосфена

```
1 def sieve(n: int):
2     if n < 2:
3         return []
4
5     is_prime = [True] * (n + 1)
6     is_prime[0] = is_prime[1] = False
7
8     limit = n ** 0.5
9     i = 2
10    while i <= limit:
11        if is_prime[i]:
12            for j in range(i * i, n + 1, i):
13                is_prime[j] = False
14            i += 1
15
16    return [k for k in range(2, n + 1) if is_prime[k]]
```

7.3 Сколько это работает

- Память: $O(N)$ на массив `is_prime`.
- Время: $O(N \log \log N)$ — очень быстро для разумных N (скажем, до 10^8 при аккуратной реализации).

8 Факторизация чисел

Задача: по целому n найти его разложение на простые множители.

8.1 Наивный метод: простое деление

Самый прямолинейный способ — просто пробовать делители.

Algorithm 5 Факторизация простым делением

```
1 def trial_division(n: int) -> list[int]:
2     factors = []
3     d = 2
4
5     while d * d <= n:
6         while n % d == 0:
7             factors.append(d)
8             n //= d
9         if d == 2:
10            d += 1
11        else:
12            d += 2
13    if n > 1:
14        factors.append(n)
15
16    return factors
```

Этот метод удивительно неплох, если числа не слишком большие.

8.2 Зачем нам всё это

Коротко, где всплывают эти алгоритмы:

- Алгоритм Евклида и его расширенная версия — основа почти всей модульной арифметики.
- Быстрое возведение в степень нужно в тестах простоты и криптографии.
- Решето Эратосфена — стандартный способ сгенерировать много простых чисел сразу.
- Факторизация (в т.ч. Pollard's Rho) — шаг в сторону реальных криптосистем и оценки их стойкости.

8.3 Решение задач по пройденным темам

Задача 1. Count The Divisors

Условие задачи.

Даны два целых числа l и r . Необходимо посчитать общее количество *различных простых делителей* всех целых чисел на отрезке $[l, r]$ включительно.

Иными словами, нужно рассмотреть все числа $l, l + 1, \dots, r$, выписать все их простые делители, затем оставить только **уникальные** простые числа и посчитать их количество.

Пример.

Пусть $l = 5, r = 15$.

Числа на отрезке: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

Их простые делители:

5 : 5

6 : 2, 3

7 : 7

8 : 2

9 : 3

10 : 2, 5

11 : 11

12 : 2, 3

13 : 13

14 : 2, 7

15 : 3, 5

Множество различных простых делителей:

$\{2, 3, 5, 7, 11, 13\}$,

всего 6 чисел.

Значит, ответ для этого примера равен 6.

Формат ввода

Первая строка входных данных содержит два целых числа l и r :

$$1 \leq l \leq r \leq 2 \cdot 10^6.$$

Формат вывода

Выведите одно целое число — количество различных простых делителей всех целых чисел на отрезке $[l, r]$.

Пример

Ввод	Вывод
5 15	6

Python-код решения

```
1 l, r = map(int, input().split())
2
3 prime = [True] * (r + 1)
4 prime[0] = prime[1] = False
5 primes = []
6
7 for i in range(2, r + 1):
8     if prime[i]:
9         primes.append(i)
10        for j in range(i * i, r + 1, i):
11            prime[j] = False
12
13 ans = 0
14
15 for p in primes:
16     first = ((l + p - 1) // p) * p
17     if first <= r:
18         ans += 1
19
20 print(ans)
```

Задача 2. Сложность двоичного поиска (Быки и коровы)

Условие.

Загадано некоторое целое число от 1 до N включительно. Вы можете задавать вопросы вида: «Загаданное число это x ?» и получать один из трёх ответов:

- «Угадал число.»
- «Загаданное число больше.»
- «Загаданное число меньше.»

Используя стратегию **деления пополам** (двоичный поиск), нужно определить:

Какое наименьшее количество вопросов нужно гарантированно задать, чтобы точно угадать число?

Формат ввода

Вводится одно целое число N ($1 \leq N \leq 10^5$).

Формат вывода

Выведите одно целое число — минимальное количество вопросов, которых гарантированно достаточно, чтобы угадать загаданное число на отрезке $[1, N]$.

Комментарий к задаче (идея решения).

Метод деления пополам — это *двоичный поиск*. После каждого вопроса множество возможных чисел делится примерно пополам:

- после 1 вопроса остаётся не более $N/2$ вариантов,
- после 2 вопросов — не более $N/4$,
- после k вопросов — не более $N/2^k$.

Нужно, чтобы после некоторого количества шагов k гарантированно осталось не больше одного возможного числа:

$$\frac{N}{2^k} \leq 1 \iff 2^k \geq N.$$

Минимальное такое k — это

$$k = \lceil \log_2 N \rceil.$$

Именно столько вопросов в худшем случае потребуется при оптимальной стратегии двоичного поиска.

Пример

Ввод	Вывод
8	3

Пояснение: $2^3 = 8$, значит за 3 вопроса можно гарантированно определить число от 1 до 8.

Решение на Python

Ниже приведён код, который вычисляет минимальное k такое, что $2^k \geq N$, используя целочисленный цикл (без плавающей арифметики):

```
1 def min_questions(n: int) -> int:
2     k = 0
3     cur = 1
```

```

4   while cur < n:
5       cur *= 2
6       k += 1
7   return k

```

Задача 3. Суммирование по составным

Условие.

Задано целое число N ($4 \leq N \leq 10^6$). Нужно найти сумму *наименьших простых делителей* всех **составных** чисел, которые:

- строго больше 2,
- не превосходят N .

Напомним, что число называется *составным*, если оно больше 1 и не является простым (то есть имеет нетривиальные делители).

Формат ввода

Вводится одно целое число N .

Формат вывода

Выведите одно целое число — сумму наименьших простых делителей всех составных чисел x , таких что $2 < x \leq N$.

Примеры

Ввод	Вывод
5	2
9	9

Пояснение ко второму примеру: составные числа от 3 до 9 — это 4, 6, 8, 9.

4 : наименьший простой делитель = 2,

6 : 2,

8 : 2,

9 : 3.

Сумма: $2 + 2 + 2 + 3 = 9$.

Основная идея Вместо того чтобы для каждого числа отдельно искать его наименьший простой делитель, мы используем эффективный алгоритм, который заранее вычисляет наименьшие простые делители для всех чисел от 2 до N .

Идея решения

Для эффективного решения задачи используется модифицированный алгоритм решета Эратосфена, который позволяет для каждого числа от 2 до N найти его наименьший простой делитель.

Алгоритм:

1. Создаём массив `min_prime` размером $N + 1$, где `min_prime[i]` будет хранить наименьший простой делитель числа i
2. Инициализируем массив: для каждого i от 2 до N полагаем `min_prime[i] = i`
3. Для каждого числа i от 2 до \sqrt{N} :
 - Если `min_prime[i] == i` (т.е. i — простое число)
 - Для всех чисел $j = i^2, i^2 + i, i^2 + 2i, \dots \leq N$:
 - Если `min_prime[j] == j`, то устанавливаем `min_prime[j] = i`
4. Суммируем значения `min_prime[i]` для всех составных чисел от 4 до N

Обоснование корректности:

- У каждого составного числа есть единственный наименьший простой делитель
- Решето Эратосфена эффективно находит все простые числа до N
- Модификация позволяет сразу отмечать наименьшие простые делители для всех составных чисел

Пример для $N = 9$:

- Составные числа: 4, 6, 8, 9
- Наименьшие простые делители: 2, 2, 2, 3
- Сумма: $2 + 2 + 2 + 3 = 9$

Решение на Python

```
1 n = int(input())
2 min_prime = [0] * (n + 1)
3
4 for i in range(2, n + 1):
5     min_prime[i] = i
6
7 for i in range(2, int(n**0.5) + 1):
8     if min_prime[i] == i:
9         for j in range(i * i, n + 1, i):
10             if min_prime[j] == j:
11                 min_prime[j] = i
12 total = 0
13 for i in range(4, n + 1):
14     if min_prime[i] != i:
```

```
15         total += min_prime[i]
16
17 print(total)
```