Краткое введение в начала элементарной теории чисел

Денис Кириенко

Летняя компьютерная школа, 4 января 2021 года

Целочисленное деление

Пусть дано два целых числа a и b, $b \neq 0$. Целочисленным частным от деления a на b называется величина $q = \left\lfloor \frac{a}{b} \right\rfloor$, где $\lfloor * \rfloor$ обозначают целую часть, т.е. округление результата до ближайшего целого вниз, также используется обозначение $[*]$). Остатком от деления называется величина $r = a - qb$.

Легко показать, что если $b > 0$, то $0 \leqslant r < b$, если $b < 0$, то $b < r \leqslant 0$.

Остаток от деления a на b будем обозначать $a \bmod b$. Обратите внимание, что правила, по которым вычисляются операции `/` и `%` в C++, Java, C# не соответствуют этому определению, если одно из чисел a или b — отрицательное.

Алгоритм Евклида

Через (a, b) будем обозначать наибольший общий делитель (НОД) целых неотрицательных чисел a и b, одновременно не равных 0.

Для вычисления НОД существует эффективный алгоритм Евклида, основанный на свойстве $(a, b) = (a - b, b) = (a \bmod b, b)$, где $a \bmod b$ — операция взятия остатка от деления a на b.

В результате получим следующий рекурсивный алгоритм Евклида. В реализации предполагается, что $a > b$ (подумайте, что произойдет, если это не так).

Бинарный алгоритм Евклида

Бинарный алгоритм Евклида не использует деления, поэтому его удобно использовать вместе с длинной арифметикой, где трудно реализовывать

```python
def gcd(a,b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Listing 1: Рекурсивная реализация алгоритма Евклида

```python
def gcd(a, b):
    while b > 0:
        a, b = b, a % b
    return a
```

Listing 2: Нерекурсивная реализация алгоритма Евклида

деления. Бинарный алгоритм Евклида использует следующие соотношения:

$$(a,b) = \begin{cases} 2(a/2, b/2), & \text{если } a \text{ и } b \text{ — четные,} \\ (a/2, b), & \text{если } a \text{ — четное, } b \text{ — нечетное,} \\ (a, b/2), & \text{если } a \text{ — нечетное, } b \text{ — четное,} \\ (a-b, b), & \text{если } a \text{ и } b \text{ — нечетные, } a > b, \\ (a, b-a), & \text{если } a \text{ и } b \text{ — нечетные, } a < b. \end{cases}$$

Реализацию бинарного алгоритма Евклида оставим читателю.

Расширенный алгоритм Евклида

Рассмотрим два целых числа a и b. Какое множество значений произвольной целочисленной линейной комбинации этих чисел, то есть какие значения может принимать выражение $ax+by$, где x и y — произвольные целые числа x и y?

Пусть $d = (a, b)$. Очевидно, что $ax + by$ делится на d. Докажем, что оно может быть равно в точности d, для чего достаточно найти такие x и y, что $ax + by = d$. Как следствие, значение $ax + by$ может принимать любые значения, кратные d

Такие числа x и y находит расширенный алгоритм Евклида. Реализуем его в виде функции `gcdex(a,b)`, получающей на вход два числа a

и b и возвращающей тройку чисел (d, x, y), где $d = (a, b)$, $d = ax + by$.

```
def gcdex(a, b):
    if b == 0:
        return a, 1, 0
    d1, x1, y1 = gcdex(b, a % b)
    d, x, y =  d1,  y1,  x1 - (a // b) * y1
    return d, x, y
```

Listing 3: Рекурсивная реализация расширенного алгоритма Евклида

```
def gcdex(a, b):
    xa, ya, xb, yb = 1, 0, 0, 1
    while b > 0:
        (a, xa, ya, b, xb, yb) = (b, xb, yb,
            a%b, xa - xb*(a//b), ya - yb*(a/b))
    return a, xa, ya
```

Listing 4: Нерекурсивная реализация расширенного алгоритма Евклида

Подсказка к понимаю алгоритма: если a_0 и b_0 — это первоначальные значения a и b, переданные в качестве параметров, то в любой момент $a = a_0x_a + b_0y_a$, $b = a_0x_b + b_0y_b$.

Подумайте над проблемой реализации бинарного расширенного алгоритма Евклида.

Использование расширенного алгоритма Евклида для решения линейных диофантовых уравнений

Решим уравнение $ax + by = c$ в целых числах.

Пусть $d = (a, b)$. Если c не делится на d, то уравнение решений не имеет, иначе поделим все на d и далее будем считать, что a и b — взаимно простые.

При помощи расширенного алгоритма Евклида найдем два числа x_0 и y_0 таких, что $ax_0+by_0 = 1$. Положим $x_1 = cx_0$, $y_1 = cy_0$. Тогда числа x_1 и y_1 являются одним из решений данного уравнения (частное решение).

Общее решение данного уравнения имеет вид $x = x_1+kb$, $y = y_1-ka$, где k — произвольное целое число. Докажите самостоятельно, что мно-

жество всех решений исходного уравнения исчерпывается этими числами.

Сравнения по модулю

Говорят, что числа a и b сравнимы по модулю n, если они дают одинаковые остатки при делении на n, а если формально, то если $a - b$ делится на n.

Записывается так: $a \equiv b \pmod{n}$

Свойства сравнений

1. Если $a \equiv b \pmod{n}$, $c \equiv d \pmod{n}$, то $a + c \equiv b + d \pmod{n}$.
2. Если $a \equiv b \pmod{n}$, $c \equiv d \pmod{n}$, то $ac \equiv bd \pmod{n}$.
3. Если $ac \equiv ad \pmod{n}$ и $(a, n) = 1$, то $c \equiv d \pmod{n}$.
4. Если $ak \equiv bk \pmod{nk}$ и $k \neq 0$, то $a \equiv b \pmod{n}$.

Мы видим, что работая в терминах вычетов по модулю n, можно числа складывать и умножать. Такая алгебраическая структура с операциями умножения называется кольцом, в данном случае говорят о кольце вычетов по модулю n.

Обратные элементы в кольцах вычетов и решение линейных сравнений

Если же n — простое, то согласно последнему свойству сравнения можно сокращать на любое ненулевое число. Более того, в кольце вычетов по простому модулю можно выполнять деление на любое ненулевое число и, как следствие, можно решать уравнения вида $ax \equiv b \pmod{n}$. Решением такого уравнения является число $x = ba^{-1}$, где a^{-1} - обратный элемент к a в кольце вычетов по модулю n, то есть такое число, что $aa^{-1} \equiv 1 \pmod{n}$.

Если n — простое, то для любого ненулевого числа существует обратный элемент, и, следовательно, можно делить на любое ненулевое число. Для нахождения обратного элемента к a в кольце вычетов по модулю n используем расширенный алгоритм Евклида для чисел a и n. Он найдет такие числа x и y, что $ax + ny = 1$, а, значит, $ax \equiv 1 \pmod{n}$.

Подумайте, для каких чисел a существует обратный элемент в кольце вычетов по модулю n, если n — составное.

Теперь рассмотрим решение произвольного линейного сравнения $ax \equiv b \pmod{n}$.

Шаг 1. Пусть $d = (a, b, n)$ (наибольший общий делитель трех чисел). Тогда сократим a, b, n на d (свойство 4).

Шаг 2. Если a необратим в кольце вычетов по модулю n, то решений нет. Иначе $x = ba^{-1} \pmod{n}$ — решение.

Китайская теорема об остатках

В решении систем линейных сравнений помогает китайская теорема об остатках.

Простейшая система сравнений имеет вид

$$\begin{cases} x \equiv a \pmod{m}, \\ x \equiv b \pmod{n}, \end{cases}$$

где $(m, n) = 1$.

Тогда существует целое число c такое, что эта система верна тогда и только тогда, когда $x \equiv c \pmod{mn}$.

Доказательство. Поскольку m и n — взаимно простые, то существуют числа $r = n^{-1} \pmod{m}$ и $s = m^{-1} \pmod{n}$. Положим $c = arn + bsm$. Тогда если $x \equiv c \pmod{mn}$, то

$$x \equiv c = arn + bsm \equiv arn \equiv ann^{-1} \equiv a \pmod{m}$$

$$x \equiv c = arn + bsm \equiv bsm \equiv bmm^{-1} \equiv b \pmod{n}$$

Докажем утверждение в другую сторону. Предположим, что $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$. Выше было доказано, что $c \equiv a \pmod{m}$ и $c \equiv b \pmod{n}$. Поэтому, $x \equiv c \pmod{m}$, то есть $x - c$ делится на m. Аналогично, $x - c$ делится на n, откуда $x - c$ делится на mn, следовательно, $x \equiv c \pmod{mn}$.

Подумайте сами, как быть с системой из трёх и более неравенств и как быть, если m и n не взаимно просты.

Решето Эратосфена

Начнем с того, что вспомним, что мы умеем проверять число n на простоту и строить его разложение на простые множители за $O(\sqrt{n})$. Останавливаться на этом алгоритме подробно не будем. Это не лучший алгоритм, сейчас известны и лучшие алгоритмы. Один из них мы рассмотрим позднее.

Рассмотрим другую задачу — построение множества всех простых чисел, не превосходящих данного числа n.

Классическое решето Эратосфена. 2 — простое число, а все числа, кратные 2, не простые и вычеркнем их. Первое невычеркнутое число — 3, вычеркиваем далее все числа, кратные 3. Следующее невычеркнутое число — 5, вычеркиваем все числа, кратные 5. Напишем функцию `sieve(n)`, которая строит список `primes` всех простых чисел, не превосходящих n и возвращающую этот список. Вспомогательный массив `is_prime` содержит набор флагов, `is_prime[a] == True`, если a — простое (или мы пока не определили, что оно составное), и `is_prime[a] == False`, если a — составное.

```
def sieve(n):
    is_prime = [True] * (n + 1)
    primes = []
    a = 2
    while a <= n:
        if is_prime[a]:
            primes.append(a)
            i = a * a
            while i <= n:
                is_prime[i] = False
                i += p
        a += 1
    return primes
```

Listing 5: Решето Эратосфена

Сложность данного алгоритма есть $O(n/2 + n/3 + n/5 + n/7 + \dots) = O(n \log \log n)$.

Это хорошая сложность, но можно сделать решето и за $O(n)$. В этом алгоритме мы будем для каждого числа n хранить такое минимальное простое $p = p(n)$, что $n = ap$. Заметим, что если p — простое, то $n = p$, $a = 1$. Если n — составное, то тогда $a \geqslant p$, и, более того, $p(a) \geqslant p$.

Во вспомогательном массиве `p` будем хранить для каждого a значение $p(a)$, определенное выше. Первоначально массив заполнен нулями. Переходя к очередному числу, если мы обнаруживаем, что для этого числа мы еще не определили $p(a)$, то это число объявляется простым. Далее мы вычеркиваем все числа, имеющие вид pa для данного a, где

p – все простые, не превосходящие $p(a)$.

Поскольку в данном случае каждое число будет вычеркнуто ровно один раз (в силу единственности представления $n = ap$), то сложность данного алгоритма $O(n)$.

```
def sieve(n):
    p = [0] * (n + 1)
    primes = []
    a = 2
    while a <= n:
        if p[a] == 0:
            P[a] = a
            primes.append(a)
        i = 0
        while i < len(primes) and \\
                  primes[i] <= p[a] and primes[i]*a <= n:
            p [a * primes[i]] = primes[i]
            i += 1
        a += 1
    return primes
```

Listing 6: Решето Эратосфена за $O(n)$

Малая теорема Ферма

Теорема. Если p — простое, $a \neq 0 \pmod p$, то $a^{p-1} \equiv 1 \pmod p$.

Доказательство. Рассмотрим набор чисел $1, 2, 3, \ldots, p-1$. Рассмотрим другой набор чисел: $a \cdot 1, a \cdot 2, \ldots, a \cdot (p-1)$. Второй набор является перестановкой первого набора в кольце вычетов по модулю p, т.к. $ai \equiv aj \pmod p$ тогда и только тогда, когда $i \equiv j \pmod p$. Но тогда произведения чисел в первом наборе совпадает с произведением чисел во втором наборе, т.е.

$$1 \cdot 2 \cdot \ldots \cdot (p-1) \equiv a \cdot 1 \cdot a \cdot 2 \cdot \ldots \cdot a \cdot (p-1) \pmod p$$

откуда

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

Функция Эйлера

Функцией Эйлера от натурального числа n называется количество чисел, не превосходящих n и взаимно простых с n, и обозначается $\varphi(n)$. Примеры: $\varphi(1) = 1$, $\varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(4) = 2$, $\varphi(5) = 4$.

Свойства функции Эйлера:

1. $\varphi(p) = p - 1$, если p — простое.
2. $\varphi(p^n) = p^n - p^{n-1}$, если p — простое.
3. $\varphi(pq) = (p-1)(q-1)$, если p и q — простые.
4. $\varphi(nm) = \varphi(n) \cdot \varphi(m)$, если $(n, m) = 1$.

Докажем последнее свойство. Существует ровно $\varphi(m)$ чисел, меньших m и взаимно простых с m. Обозначим их x_1, x_2,..., $x_{\varphi(m)}$. Аналогично существует ровно $\varphi(n)$ чисел, меньших n и взаимно простых с n, которые обозначим y_1, y_2,..., $y_{\varphi(n)}$. Рассмотрим $\varphi(m)\varphi(n)$ чисел $z_{ij} = nx_i + my_j \bmod mn$. Эти числа меньше mn, кроме того, $z_{ij} \equiv nx_i \pmod m$, $z_{ij} \equiv my_j \pmod n$, значит, z_{ij} взаимно просто с m и n, а, значит, и с mn.

Отсутствие других чисел, кроме указанных, следует из китайской теоремы об остатках.

Легко видеть, что если разложение n на простые делители имеет вид $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot ... \cdot p_k^{\alpha_k}$, то $\varphi(n) = \varphi(p_1^{\alpha_1}) \cdot \varphi(p_2^{\alpha_2}) \cdot ... \cdot \varphi(p_k^{\alpha_k}) = (p_1^{\alpha_1} - p_1^{\alpha_1 - 1}) \cdot (p_2^{\alpha_2} - p_2^{\alpha_2 - 1}) \cdot ... \cdot (p_k^{\alpha_k} - p_k^{\alpha_k - 1}) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdot ... \cdot (1 - \frac{1}{p_k})$.

Теорема Эйлера

Теорема Эйлера является обобщением малой теоремы Ферма.

Пусть $(a, n) = 1$. Тогда $a^{\varphi(n)} \equiv 1 \pmod n$.

Доказательство аналогично доказательству малой теоремы Ферма.

Прочие числовые функции

$\tau(n)$ — количество делителей

Количество делителей натурального числа n, включая 1 и само число, обозначим через $\tau(n)$.

Легко видеть, что $\tau(n) = (\alpha_1 + 1)(\alpha_2 + 1) \cdot ... \cdot (\alpha_k + 1)$.

$\sigma(n)$ — сумма делителей

Сумму всех делителей числа n будем обозначать $\sigma(n)$.

Например, $\sigma(6) = 1 + 2 + 3 + 6 = 12$.

Легко видеть, что $\sigma(n) = (1+p_1+p_1^2+\cdots+p_1^{\alpha_1})...(1+p_k+p_k^2+\cdots+p_k^{\alpha_k}) = \frac{p_1^{\alpha_1+1}-1}{p_1-1} \cdot ... \cdot \frac{p_k^{\alpha_k+1}-1}{p_k-1}$

$\pi(n)$ — количество простых чисел, не превосходящих n

Через $\pi(n)$ обозначается количество простых чисел, не превосходящих n. Например, $\pi(10) = 4$.

Формулы для вычисления $\pi(n)$ не существует, а оценка $\pi(n)$ — сложная задача современной теории чисел. Одна из самых простых оценок: $\pi(n) \approx \frac{n}{\ln(n)}$, то есть среди первых n натуральных чисел доля простых примерно равна $\frac{1}{\ln(n)}$.

Алгоритм Рабина-Миллера проверки на простоту

Алгоритм Рабина-Миллера позволяет проверять числа на простоту существенно быстрее перебора всех делителей. Этот алгоритм является вероятностным, т.е. он не гарантированно дает правильный ответ, но вероятность ошибки можно сделать сколь угодно малой.

Можем воспользоваться малой теоремой Ферма. Возьмем данное число n, возьмем произвольное число a и вычислим $a^{n-1} \pmod n$. Если получили что-то, отличное от 1, то число n составное, а иначе... Увы, иначе ничего определенного сказать нельзя. Можно только проверив выполнение малой теоремы Ферма для большого числа пробных чисел a предполагать, что число n очень похоже на простое.

У данного алгоритма есть один недостаток: к сожалению, существует бесконечно много (это доказано в 1993 году) составных чисел (наименьшее из которых равно 561), называемых Кармайкловыми числами, для которых выполняется утверждение малой теоремы Ферма. Усовершенствованный тест придуман математиками Рабином и Миллером.

Пусть n — простое, $n-1 = 2^s t$, где t — нечетное. Тогда

$$a^{n-1} - 1 = (a^t - 1)(a^t + 1)(a^{2t} + 1) \dots (a^{2^{s-1}t} + 1)$$

и, значит, если a и n — взаимно простые, то левая часть этого тождества делится на n, и, значит, одна из скобок в правой части делится на n.

Назовем число n псевдопростым в базе a если $(a, n) = 1$ и в записанном выше выражении одна из скобок делится на n, то есть $a^t \equiv 1 \pmod n$ или существует такое k, $0 \leqslant k < s$, что $a^{2^k t} \equiv -1 \pmod n$.

Таким образом мы получим более сильное условие, чем простая проверка выполнения малой теоремы Ферма для данного a.

При определении, является ли число n псевдопростым в базе a, необходимо сначала вычислить a^t при помощи алгоритма быстрого возведения в степень, а затем, возводя его в квадрат, вычислять a^{2t}, a^{4t} и так далее.

Можно доказать, что если n — простое, то для любого a, $1 < a < n$, n псевдопросто в базе a. Если же n — составное, то существует не менее $3n/4$ чисел a таких, что n не псевдопросто в базе a. Таким образом, для проверки числа n на простоту можно выбирать случайные числа $a_1, \dots a_t$. Если число n псевдопросто в базе a_i для любого i, то будем считать n — простым, иначе — составным. Итак, если n — простое, то данный алгоритм всегда даст ответ, что оно простое. Если же n — составное, то с вероятностью не выше чем $1/4^t$ алгоритм даст неправильный ответ, то есть что это число простое. Тем не менее, вероятность ошибки можно сделать сколь угодно малой. Например, если взять $t = 10$, то вероятность ошибки меньше 10^{-6}, а если $t = 20$, то вероятность ошибки меньше 10^{-12}, то есть практически равна нулю.

На практике достаточно проверить псевдопростоту числа n для ряда небольших чисел a, как правило, в качестве a берут несколько наименьших простых чисел. Например, проверка на простоту при помощи четырех тестов Рабина–Миллера для $a = 2, 3, 5, 7$ будет выдавать верный ответ для всех простых, не превосходящих $25 \cdot 10^9$, кроме одного числа (а именно, 3215031751).

В настоящий момент известен и детерминированный (то есть не вероятностный, как алгоритм Рабина–Миллера) полиномиальный (сложность которого есть многочлен от длины числа n, а именно, $O(\log^6 n)$ тест на простоту (тест Агравала – Каяла – Саксены), однако он носит сугубо теоретический характер ввиду большой степени в сложности и больших требований к памяти.

ρ–метод Полларда разложения на простые множители

Обычно для разложения числа на простые множители считается, что достаточно найти какой-нибудь делитель d числа n, тогда число n разлагается в произведение $n = d \times (n/d)$, и если числа d и n/d — составные, то можно повторить этот процесс.

Наивный метод поиска делителя перебором имеет сложность $O(\sqrt{n})$, т.к. если число n разлагается в произведение $n = n_1 n_2$, то хотя бы один из делителей не превосходит $\sqrt{n}$.

В настоящий момент теория алгоритмов разложения на множители и проверки на простоту сильно продвинулась, рассмотрим один из ранних алгоритмов поиска делителя числа n —ρ-метод Полларда.

Рассмотрим последовательность чисел, задаваемую соотношением $x_{n+1} = (x_n^2+c) \bmod n$. Например, можно взять $x_0 = 1$, $x_{n+1} = (x_n^2+1) \bmod n$. Вообще можно взять любую функцию, не являющуюся взаимно-однозначной (нелинейной), поэтому берут квадрат и добавляют константу, при этом значения $c = 0$ и $c = -2$ не подходят.

Эта последовательность зацикливается, то есть существует такое значение p, что $x_{i+p} = x_i$ для всех достаточно больших i (у последовательности может быть непериодическая часть, но рано или поздно последовательность зацикливается).

Пусть у числа n есть делитель d. Рассмотрим последовательность $y_i = x_i \bmod d$. Эта последовательность также зацикливается, причём число p также будет её периодом. Но, возможно, что эта последовательность зацикливается раньше, то есть найдутся два таких элемента x_j и x_i, такие, что $x_j \equiv x_i \pmod d$, но $x_j \not\equiv x_i \pmod n$.

Тогда $x_j - x_i$ делится на d, но не делится на n), и мы можем найти значение d (или другого делителя числа n), взяв НОД чисел n и $x_j - x_i$. Заметим, что мы не знаем число d, но находим его (или другой делитель числа n), используя НОД.

Единственное что мы не знаем — это период последовательности по модулю d, то есть какие два элемента x_j и x_i нужно взять, чтобы они образовывали период последовательности по модулю d. Для поиска этого периода воспользуемся идеей «двух указателей». Возьмём $i = 1$, $j = 2$, тогда $j - i = 1$. Рассмотрим пару (x_i, x_j). Затем увеличим i на 1 и j на 2 и снова рассмотрим пару (x_i, x_j), на этот раз $i = 2$, $j = 4$ и $j - i = 2$. Будем продолжать увеличивать i на 1, j на 2, тогда разность $j - i$ увеличивается на 1, то есть расстояние между рассматриваемыми элементами последовательности растёт. Рано или поздно мы найдём два таких элемента x_j и x_i, что $x_j \equiv x_i \pmod d$, то есть мы найдём период последовательности по модулю d. Если нам повезло и найденный период не будет периодом по модулю n, то тогда $x_j \not\equiv x_i \pmod n$, и $(x_j - x_i, n)$ будет нетривиальным делителем числа n.

Но если нам не повезло и числа x_i и x_j оказались сравнимы по модулю n, то мы «попали» в период по модулю n и попытка оказалась неудачной. Возьмём в этом случае другое значение x_0 или другое значение константы c.

Дадим эвристическую оценку сложности алгоритма. Если последовательность «хорошая», то можно считать, что она выглядит, как псев-

```
def f(x):
        return (x * x + c) % n

xi = 1
xj = f(x)
d = gcd(abs(xj - xi))
while d == 1:
        xi = f(xi)
        xj = f(f(xj))
        d = gcd(abs(xj - xi), n)
 if d != n:
    print(d, 'divisor')
else:
    print('Test failed')
```

Listing 7: ρ-метод Полларда

дослучайная последовательность чисел по модулю d. Тогда исходя из «парадокса дней рождений», если рассмотреть $O(\sqrt{d})$ чисел, то вероятность, что среди них найдутся равные по модулю d будет близка к единице. То есть мы найдём подходящую пару чисел за $O(\sqrt{d})$. Но у числа n будет делитель d не превосходящий n, поэтому понадобится порядка $O(\sqrt{\sqrt{n}})$ шагов алгоритма. Итоговая сложность — $O(n^{1/4}\log n)$ (логарифм появился из-за вычисления НОД на каждом шаге).

При этом перед запуском Полларда (если мы не уверены, что проверяемое число — составное), необходимо выполнить его проверку на простоту быстрым методом, например, Рабином–Миллером, и если число будет простым, то можно использовать метод Полларда.