

Алгоритмы теории чисел

Рыжичкин Кирилл

Февраль 2024

Оглавление

Введение	3
1 Простейшие алгоритмы теории чисел	4
1. Простые числа	4
2. НОД и НОК двух чисел	6
2.1. Алгоритм Евклида	6
3. НОД и НОК n чисел	8
4. Расширенный алгоритм Евклида	9
5. Линейные диофантовы уравнения с двумя неизвестными	11
5.1. Алгоритм решения линейного диофантова уравнения с двумя неиз- вестными	11
6. Количество/сумма делителей числа	13
7. Степень вхождения простого числа в факториал	15
8. Решето Эратосфена	16
2 Модульная арифметика	18
1. Сравнения по модулю	18
1.1. Свойства сравнений	18
2. Обратный элемент в кольце вычетов по модулю	18
2.1. Существование и единственность	19
2.2. Нахождение обратного элемента	19
3. Линейные сравнения с неизвестной	20

Введение

Мир алгоритмов теории чисел представляет собой захватывающий путь в глубины математического мира, где числа становятся не только абстрактными сущностями, но и основой для разработки эффективных алгоритмов. Теория чисел, одна из старейших областей математики, занимается изучением свойств целых чисел и их взаимосвязей. Вместе с тем, алгоритмы теории чисел играют критическую роль в современном мире, находя применение в широком спектре областей, включая криптографию, компьютерные науки, теорию игр и многое другое.

Мы начнем с основных понятий, таких как простые числа, делители, наибольший общий делитель и модульная арифметика, и постепенно углубимся в более сложные темы, такие как теорема Эйлера, китайская теорема об остатках, алгоритмы факторизации и многое другое.

Глава 1

Простейшие алгоритмы теории чисел

1. Простые числа

Определение. *Натуральное число p называется простым (англ. prime number), если $p > 1$ и p не имеет натуральных делителей, отличных от 1 и p .*

Определение. *Натуральное число $n > 1$ называется составным (англ. composite number), если n имеет по крайней мере один натуральный делитель, отличный от 1 и n .*

Согласно определениям, множество натуральных чисел разбивается на 3 подмножества:

1. Простые числа.
2. Составные числа.
3. Число 1, которое не причисляется ни к простым, ни к составным числам.

Отсюда сразу возникает идея реализации интуитивно понятного алгоритма проверки числа n на простоту: если $n = 1$, вернуть false, иначе пройтись по всем числам от 2 до $n - 1$, и если ни на одно из них число не поделится, то вернуть true. Соответственно сложность такого алгоритма будет $O(n)$.

Посмотрим на реализацию такого простейшего алгоритма:

```
bool isPrime(int n) {  
    if (n == 1) {  
        return false;  
    }  
    for (int i = 2; i < n; i++) {  
        if (n % i == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

Утверждение. Пусть $N = a \times b$, причем $a \leq b$. Тогда $a \leq \sqrt{N} \leq b$

Доказательство. Если $a \leq b < \sqrt{N}$, то $ab \leq b^2 < N$, но $ab = N$. А если $\sqrt{N} < a \leq b$, то $N < a^2 \leq ab$, но $ab = N$. ■

Из этого следует, что если число N не делится ни на одно из чисел $2, 3, 4, \dots, \lfloor \sqrt{N} \rfloor$, то оно не делится и ни на одно из чисел $\lceil \sqrt{N} \rceil + 1, \dots, N - 2, N - 1$, так как если есть делитель больше корня (не равный N), то есть делитель и меньше корня (не равный 1). Поэтому в цикле for достаточно проверять числа не до N , а до корня. Соответственно сложность такого алгоритма будет уже $O(\sqrt{n})$:

```
bool isPrime(int n) {  
    if (n == 1) {  
        return false;  
    }  
    for (int i = 2; i * i <= n; i++) {  
        if (n % i == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

Однако и этот алгоритм вовсе не оптимален и на практике применяется разве что на маленьких числах. Более оптимальные алгоритмы проверки на простоту будут рассмотрены в следующих главах.

2. НОД и НОК двух чисел

Определение. Наибольший общий делитель (НОД) двух натуральных чисел a и b это наибольшее натуральное число d , которое делит оба числа a и b без остатка. Обозначается как $d = \text{НОД}(a, b)$.

Утверждение. Для любых двух натуральных чисел $a > b$ верно следующее равенство:

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

Доказательство. $n = \text{НОД}(a, b)$ – наибольший среди всех общих делителей чисел a и b , тогда $a : n$ и $b : n$. Отсюда $a - b : n$, значит, n – делитель $a - b$ и b , он не превосходит наибольшего из всех общих делителей чисел $a - b$ и b , т.е. $n \leq m = \text{НОД}(a, b)$. Аналогично, рассуждая получаем, что $a - b : m$ и $b : m = \text{НОД}(a, b)$. Отсюда $a = (a - b) + b : m$ значит, m – делитель a и b , он не превосходит наибольшего из всех общих делителей чисел a и b , т.е. $m \leq n = \text{НОД}(a, b)$. Таким образом получаем $m \leq n \leq m$, т.е. $\text{НОД}(a, b) = n = m = \text{НОД}(a - b, b)$. ■

2.1. Алгоритм Евклида

1. **Инициализация:** Начнем с двух целых чисел a и b , где $a \geq b$.
2. **Шаг 1:** Разделим a на b и найдем остаток от деления. Пусть остаток обозначается как r . То есть, $a = bq_1 + r$, где q_1 – частное, а r – остаток.
3. **Шаг 2:** Если остаток r равен нулю, то $\text{НОД}(a - b, b)$ равен b , и алгоритм завершается.
4. **Шаг 3:** Если остаток r не равен нулю, заменим a на b и b на r , и вернемся к шагу 1.
5. **Шаг 4:** Повторяем процесс, пока остаток r не станет равным нулю. Когда это произойдет, $\text{НОД}(a, b)$ будет равен последнему ненулевому остатку, который был получен на предыдущем шаге.
6. **Вывод результата:** Когда остаток становится равным нулю, последнее значение b будет искомым $\text{НОД}(a, b)$.

Алгоритм Евклида гарантирует, что за конечное число шагов мы получим $\text{НОД}(a, b)$, так как на каждом шаге остаток уменьшается и стремится к нулю.

Реализация алгоритма на C++:

```
int64_t NumberTheory::Gcd(int64_t a, int64_t b) {
    while (b != 0) {
        int64_t temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Определение. Наименьшее общее кратное (НОК) двух натуральных чисел a и b это наименьшее натуральное число l , которое делится на оба числа a и b без остатка. Обозначается как $l = \text{НОК}(a, b)$.

Утверждение. $\forall a, b \in \mathbb{N}: \text{НОД}(a, b) \cdot \text{НОК}(a, b) = ab$.

Доказательство. Во время доказательства нам пригодится следующее простое свойство:

$$\min(n; m) + \max(n; m) = n + m.$$

Оно верно, т.к. $\min(n, m)$ совпадает с одним из чисел n или m , а $\max(n, m)$ с другим. Запишем каноническое разложение на простые множители чисел a и b :

$$a = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k} \text{ и } b = p_1^{m_1} p_2^{m_2} \dots p_k^{m_k}$$

(возможно, что степени некоторых простых нулевые, т.е. простое число входит в разложение одного из чисел a или b на простые множители, но не входит в разложение второго). Запишем теперь разложение НОД (a, b) и НОК (a, b) :

$$\begin{aligned} \text{НОД}(a, b) &= p_1^{\min\{n_1, m_1\}} p_2^{\min\{n_2, m_2\}} \dots p_k^{\min\{n_k, m_k\}} \\ \text{НОК}(a, b) &= p_1^{\max\{n_1, m_1\}} p_2^{\max\{n_2, m_2\}} \dots p_k^{\max\{n_k, m_k\}} \end{aligned}$$

То есть

$$\text{НОД}(a, b) \cdot \text{НОК}(a, b) = \prod_{i=1}^k p_i^{\min\{n_i, m_i\} + \max\{n_i, m_i\}} =$$

в силу свойства, указанного в самом начале доказательства, получаем

$$= \prod_{i=1}^k p_i^{n_i + m_i} = \prod_{i=1}^k p_i^{n_i} \prod_{i=1}^k p_i^{m_i} = ab.$$

■

Отсюда алгоритм нахождения получается элементарным:

```
int64_t NumberTheory::Lcm(int64_t a, int64_t b) {  
    return a * b / Gcd(a, b);  
}
```

3. НОД и НОК n чисел

Алгоритм Евклида для нескольких чисел использует основное свойство наибольшего общего делителя: если d делит каждое из чисел a_1, a_2, \dots, a_n , то d также делит и их НОД.

Ход алгоритма Евклида для n чисел:

1. **Инициализация:** Пусть у нас есть набор чисел a_1, a_2, \dots, a_n .
2. **Нахождение НОДа для первых двух чисел:**

$$d_1 = \text{НОД}(a_1, a_2)$$

3. **Обновление НОДа для следующих чисел:**

Для каждого $i = 3, 4, \dots, n$:

$$d_i = \text{НОД}(d_{i-1}, a_i)$$

4. **Конечный результат:**

Наибольший общий делитель всех чисел: $\text{НОД}(a_1, a_2, \dots, a_n) = d_n$.

Поскольку d_1 является НОДом a_1 и a_2 , он является делителем обоих этих чисел. Следовательно, d_1 делит оба числа нацело.

Когда мы переходим к следующему числу a_i , мы находим НОД между текущим НОДом d_{i-1} и a_i . Это значит, что d_{i-1} делит a_i и d_{i-1} нацело. Поскольку d_{i-1} также является НОДом всех предыдущих чисел, он делит все предыдущие числа нацело. Таким образом, d_{i-1} делит a_i и все предыдущие числа нацело.

Стало быть, после обработки всех чисел мы получаем d_n , который является НОДом всех чисел a_1, a_2, \dots, a_n .

Реализация данного алгоритма на C++:

```
int64_t NumberTheory::Gcd(std::vector<int64_t> numbers) {  
    int64_t res = numbers[0];  
    for (int i = 1; i < numbers.size(); ++i) {  
        res = Gcd(res, numbers[i]);  
    }  
    return res;  
}
```

Абсолютно аналогично получается алгоритм для нахождения НОКа нескольких чисел:

```
int64_t NumberTheory::Lcm(std::vector<int64_t> numbers) {  
    int64_t res = numbers[0];  
    for (int i = 1; i < numbers.size(); ++i) {  
        res = Lcm(res, numbers[i]);  
    }  
    return res;  
}
```

4. Расширенный алгоритм Евклида

Утверждение. $\forall a, b \in \mathbb{Z} \mid \text{НОД}(a, b) = d, \exists x, y \in \mathbb{Z} : ax + by = d$.

Основная идея расширенного алгоритма Евклида заключается в том, что мы можем выразить НОД двух чисел a и b через их линейную комбинацию, то есть такие целые числа x и y , что $ax + by = \text{НОД}(a, b)$. Это следует из того факта, что множество всех линейных комбинаций a и b образует идеал в кольце \mathbb{Z} , а значит, содержит их НОД.

Будем считать, что у нас есть структура `ExtendedEuclideanResult`, которая хранит НОД(a , b), а также коэффициенты x , y , тогда расширенный алгоритм Евклида на C++ будет выглядеть так:

```

struct ExtendedEuclideanResult {
    ExtendedEuclideanResult(int64_t gcd, int64_t x, int64_t y) : gcd(
        gcd), x(x), y(y) {}
    ~ExtendedEuclideanResult() = default;

    int64_t gcd;
    int64_t x;
    int64_t y;

    friend std::ostream& operator<<(std::ostream& os, const
        ExtendedEuclideanResult& result) {
        os << "GCD:" << result.gcd << ",x:" << result.x << ",y:" <<
            result.y;
        return os;
    }
};

ExtendedEuclideanResult NumberTheory::ExtEuclide(int64_t a, int64_t
    b) {
    int64_t x = 0, y = 1, lastX = 1, lastY = 0, temp;
    while (b != 0) {
        int64_t quotient = a / b;
        int64_t remainder = a % b;

        a = b;
        b = remainder;

        temp = x;
        x = lastX - quotient * x;
        lastX = temp;

        temp = y;
        y = lastY - quotient * y;
        lastY = temp;
    }

    return ExtendedEuclideanResult(a, lastX, lastY);
}

```

5. Линейные диофантовы уравнения с двумя неизвестными

Пусть нам дано уравнение $ax + by = c$, где $a, b, c \in \mathbb{Z}$.

Заметим, что левая часть уравнения делится на $\text{НОД}(a, b)$, значит c обязано делиться на $\text{НОД}(a, b)$, чтобы решение существовало.

Если $\text{НОД}(a, b) \mid c$, то поделим обе части на этот НОД, получим новое уравнение $a'x + b'y = c'$, где $\text{НОД}(a', b') = 1$.

Пусть мы угадали какое-то решение (x_0, y_0) этого уравнения. Так как $a'x_0 + b'y_0 = c'$, то для любой пары (x, y) получим $a'x + b'y = a'x_0 + b'y_0$. Получим $a'(x - x_0) = b'(y_0 - y)$.

Так как $\text{НОД}(a', b') = 1$, то $b' \mid x - x_0$, обозначим $x - x_0 = kb'$, тогда $y - y_0 = ka'$. В итоге получаем множество решений $(x, y) = (x_0 + kb', y_0 - ka')$, где $k \in \mathbb{Z}$.

Угадать решение уравнения $a'x + b'y = c'$, где $\text{НОД}(a', b') = 1$ можно с помощью алгоритма Евклида. Сначала найдем решение (x_0, y_0) уравнения $a'x + b'y = 1$ (это можно сделать в силу утверждения со стр. 9), тогда (cx_0, cy_0) - решение уравнения $a'x + b'y = c'$.

5.1. Алгоритм решения линейного диофантова уравнения с двумя неизвестными

1. **Проверка существования решений:** Проверяем, делится ли правая часть уравнения на $\text{НОД}(a, b)$, если да - идём дальше, если нет - решения отсутствуют.
2. **Нахождение одного частного решения:** Применяя алгоритм Евклида, находим частное решение (x_0, y_0) уравнения $ax + by = c$.
3. **Нахождение общего решения:** Общее решение линейного диофантова уравнения может быть представлено в виде

$$(x_0 + \frac{b}{\text{НОД}(a, b)}t, y_0 - \frac{a}{\text{НОД}(a, b)}t), t \in \mathbb{Z}$$

Так выглядит реализация данного алгоритма на C++:

```
struct DiophantusResult {
    DiophantusResult() = default;
    DiophantusResult(int64_t x, int64_t y, int64_t k_1, int64_t k_2) :
        x(x), y(y), k_1(k_1), k_2(k_2), s("good") {}
    ~DiophantusResult() = default;

    int64_t x = 0;
    int64_t y = 0;
    int64_t k_1 = 0;
    int64_t k_2 = 0;
    std::string s = "none";

friend std::ostream& operator<<(std::ostream& os, const
    DiophantusResult& result) {
    std::string sign_1 = "+";
    std::string sign_2 = "-";

    if (result.k_1 < 0) {
        sign_1 = "-";
    }

    if (result.k_2 < 0) {
        sign_2 = "+";
    }

    if (result.s == "good") {
        os << "(x,y)=" << "(" << result.x << sign_1 << std::abs(result
            .k_1) << "t," << result.y << sign_2 << std::abs(result.k_2)
            << "t)";
    }

    else {
        os << "None";
    }

    return os;
}
```

```
};
```

```
DiophantusResult NumberTheory::Diophantus(int64_t a, int64_t b,
    int64_t c) {
    ExtendedEuclideanResult result = ExtEuclide(a, b);

    int64_t g = result.gcd;

    if (c % g != 0) {
        return DiophantusResult();
    }

    int64_t k = c / g;

    return DiophantusResult(k * result.x, k * result.y, b / g, a / g);
}
```

6. Количество/сумма делителей числа

Алгоритм нахождения количества/суммы делителей числа можно реализовать элементарным перебором делителей. Сложность такого алгоритма будет $O(\sqrt{n})$.

Рассмотрим такие алгоритмы:

```
int64_t NumberTheory::DivisorsCount(int64_t n) {
    n = std::abs(n);
    int64_t count = 0;
    for (int64_t i = 1; i <= static_cast<int64_t>(std::sqrt(n)); ++i)
    {
        if (n % i == 0) {
            count += (i == n / i) ? 1 : 2;
        }
    }
    return count;
}
```

```

int64_t NumberTheory::DivisorsSum(int64_t n) {
    for (int64_t i = 1; i <= static_cast<int64_t>(std::sqrt(n)); ++i)
    {
        if (n % i == 0) {
            sum += i;
            if (i != n / i) {
                sum += n / i;
            }
        }
    }
    return sum;
}

```

Однакого количество/сумма делителей любого числа легко выражается через его разложение на простые множители. Соответственно если мы реализуем факторизацию со сложностью, меньшей чем $O(\sqrt{n})$, то получим более оптимальный алгоритм. Такие факторизации будут рассмотрены в слующих главах, а пока будем считать, что у нас есть такой алгоритм разложения на простые множители и получим формулы количества и суммы делителей числа.

Пусть дано целое число n с разложением на простые множители:

$$n = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_k^{a_k},$$

где p_1, p_2, \dots, p_k - простые числа (не обязательно различные), а a_1, a_2, \dots, a_k - их показатели степени.

Утверждение. Пусть $\tau(n)$ обозначает количество положительных делителей числа n . Тогда $\tau(n) = (a_1 + 1) \times (a_2 + 1) \times \dots \times (a_k + 1)$.

Доказательство. Каждый делитель числа n имеет вид:

$$d = p_1^{b_1} \times p_2^{b_2} \times \dots \times p_k^{b_k},$$

где $0 \leq b_i \leq a_i$ для $i = 1, 2, \dots, k$. Чтобы получить все делители числа n , мы можем взять каждый из k простых множителей и выбрать любую комбинацию показателей степени от 0 до a_i . Таким образом, общее количество делителей числа n :

$$\tau(n) = (a_1 + 1) \times (a_2 + 1) \times \dots \times (a_k + 1) \tag{1.1}$$

■

Утверждение. Пусть $\sigma(n)$ обозначает сумму положительных делителей числа n . Тогда

$$\sigma(n) = \frac{p_1^{a_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{a_2+1} - 1}{p_2 - 1} \cdot \dots \cdot \frac{p_k^{a_k+1} - 1}{p_k - 1} \quad (1.2)$$

Доказательство. Сумма делителей числа n может быть выражена как произведение всех возможных комбинаций делителей:

$$\sigma(n) = (1 + p_1 + p_1^2 + \dots + p_1^{a_1}) \cdot (1 + p_2 + p_2^2 + \dots + p_2^{a_2}) \cdot \dots \cdot (1 + p_k + p_k^2 + \dots + p_k^{a_k})$$

Сумма геометрической прогрессии имеет вид:

$$1 + p^1 + p^2 + \dots + p^m = \frac{p^{m+1} - 1}{p - 1}$$

Подставляя это обратно в наше уравнение, получаем требуемое равенство. ■

7. Степень вхождения простого числа в факториал

Утверждение. Пусть $\text{ord}_p(n!)$ обозначает степень вхождения простого числа p в $n!$. Тогда

$$\text{ord}_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor \quad (1.3)$$

Доказательство.

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

Каждый p -ый член этого произведения делится на p , т.е. даёт $+1$ к ответу, количество таких членов равно $\left\lfloor \frac{n}{p} \right\rfloor$.

Далее, заметим, что каждый p^2 -ый член этого ряда делится на p^2 , т.е. даёт ещё $+1$ к ответу (учитывая, что p в первой степени уже было учтено до этого); количество таких членов равно $\left\lfloor \frac{n}{p^2} \right\rfloor$.

И так далее, каждый p^i -ый член ряда даёт $+1$ к ответу, а количество таких членов равно $\left\lfloor \frac{n}{p^i} \right\rfloor$.

Таким образом,

$$\text{ord}_p(n!) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \dots + \left\lfloor \frac{n}{p^i} \right\rfloor + \dots$$

.

■

Реализация данного алгоритма на C++:

```
int64_t NumberTheory::PrimePowerInFactorial(int64_t n, int64_t p) {  
    int64_t res = 0;  
    while (n) {  
        n /= p;  
        res += n;  
    }  
    return res;  
}
```

8. Решето Эратосфена

Решето Эратосфена — достаточно эффективный алгоритм для нахождения всех простых чисел в отрезке от 1 до n за $O(n \log \log n)$. Ход алгоритма:

1. Начинаем с списка чисел от 2 до n .
2. Отмечаем первое простое число в списке (2) как простое.
3. Зачеркиваем все кратные двойке числа в списке (кроме самой двойки).
4. Переходим к следующему незачеркнутому числу в списке (3), отмечаем его как простое.
5. Зачеркиваем все кратные тройке числа в списке (кроме самой тройки).
6. Повторяем этот процесс для каждого незачеркнутого числа в списке, пока не достигнем \sqrt{n} .
7. Все оставшиеся незачеркнутые числа в списке считаются простыми.

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Таблица 1.1 – Иллюстрация решета Эратосфена

Реализация данного алгоритма на C++:

```
std::vector<int64_t> NumberTheory::SieveOfEratosthenes(int64_t n) {
    std::vector<bool> isPrime(n + 1, true);
    std::vector<int64_t> primes;

    for (int p = 2; p * p <= n; p++) {
        if (isPrime[p] == true) {
            for (int64_t i = p * p; i <= n; i += p)
                isPrime[i] = false;
        }
    }

    for (int64_t p = 2; p <= n; p++) {
        if (isPrime[p])
            primes.push_back(p);
    }

    return primes;
}
```

Глава 2

Модульная арифметика

1. Сравнения по модулю

Определение. Пусть a , b , и m — целые числа, где $m > 0$. Мы говорим, что a сравнимо с b по модулю m , если m делит разность $a - b$, обозначается как

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

1.1. Свойства сравнений

1. Если $a \equiv b \pmod{m}$ и $b \equiv c \pmod{m}$, то $a \equiv c \pmod{m}$.
2. Если $a \equiv b \pmod{m}$, то $ak \equiv bk \pmod{m}$ для любого целого числа k .
3. Если $a \equiv b \pmod{m}$ и $c \equiv d \pmod{m}$, то $a + c \equiv b + d \pmod{m}$.
4. Если $a \equiv b \pmod{m}$ и $c \equiv d \pmod{m}$, то $ac \equiv bd \pmod{m}$.
5. Если $a \equiv b \pmod{m}$, то $a^k \equiv b^k \pmod{m}$ для любого натурального числа k .

2. Обратный элемент в кольце вычетов по модулю

Определение. Пусть a и m — целые числа, причем $m > 1$. Целое число a^{-1} называется обратным по модулю m элементом к a , если выполняется условие:

$$aa^{-1} \equiv 1 \pmod{m}$$

2.1. Существование и единственность

Утверждение. Обратный по модулю n элемент к a существует тогда и только тогда, когда a и n взаимно просты.

Доказательство. (\Rightarrow) Предположим, что существует обратный по модулю n элемент к a . Тогда существует такое целое число b , что $ab \equiv 1 \pmod{n}$. Это означает, что существует такое целое число k , что $ab = 1 + kn$. Таким образом, $1 = ab - kn$, что означает, что 1 является линейной комбинацией a и n . Следовательно, a и n взаимно просты по определению.

(\Leftarrow) Теперь предположим, что a и n взаимно просты. По расширенному алгоритму Евклида существуют такие целые числа x и y , что $ax + ny = 1$. Заметим, что $ax \equiv 1 \pmod{n}$. Таким образом, x является обратным по модулю n элементом к a . ■

2.2. Нахождение обратного элемента

Пусть нам надо найти обратный к a элемент по модулю m , то есть мы ищем x такой, что

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

Рассмотрим линейное диофантово уравнение $at + my = 1$, левая часть при делении на m даёт остаток at , а правая — 1, то есть чтобы найти обратный к a элемент, нам достаточно найти коэффициент t с помощью расширенного алгоритма Евклида и взять его по модулю m .

Реализация на C++:

```
int64_t NumberTheory::ModInverse(int64_t a, int64_t m) {  
    // no solutions  
    if (a == 0 || Gcd(m, a) != 1) {  
        return 0;  
    }  
  
    ExtendedEuclideanResult euclide = ExtEuclide(a, m);  
    int64_t result = ((euclide.x % m) + m) % m;  
  
    return result;  
}
```

3. Линейные сравнения с неизвестной

Определение. *Линейное сравнение по модулю m представляет собой уравнение вида $ax \equiv b \pmod{m}$, где $a, b, m \in \mathbb{Z}, m > 0$, а x - неизвестная переменная.*

Обозначим $d = \text{НОД}(a, m)$. Так как ax и b дают одинаковые остатки по модулю m , то b обязано делиться на d . Если b не делится на d , то сравнение решений не имеет. Если же b делится на d , то разделим обе части сравнения на d и перейдём к сравнению

$$a'x \equiv b' \pmod{m'} \quad (2.1)$$

Так как $\text{НОД}(a', m') = 1$, то существует элемент a'^{-1} , для которого верно

$$a'a'^{-1} \equiv 1 \pmod{m'} \quad (2.2)$$

Умножим сравнение (2.1) на a'^{-1} , тогда

$$x \equiv a'^{-1}b' \pmod{m'} \quad (2.3)$$

Соответственно решением исходного сравнения будут являться все x , дающие по модулю m' остаток $a'^{-1}b'$, и находящиеся в кольце $\mathbb{Z}/m\mathbb{Z}$. Окончательно получаем:

$$x \in \{a'^{-1}b' + km' \mid k \in \mathbb{Z}, 0 \leq k < d\} \quad (2.4)$$

Реализация алгоритма решения линейного сравнения на C++:

```
std::vector<int64_t> NumberTheory::SolveLinearCongruence(int64_t a,
    int64_t b, int64_t m) {
    int64_t d = Gcd(a, m);
    if (b % d != 0)
        return std::vector<int64_t>{ 0 }; // no solutions
    a /= d;
    b /= d;
    m /= d;
    int64_t x_0 = ModInverse(a, m) * b % m;
    std::vector<int64_t> result{ x_0 };
    for (int64_t k = 1; k < d; ++k) {
        result.push_back(x_0 + k * m);
    }
    return result;
}
```