Алгоритмы теории чисел

Рыжичкин Кирилл

Февраль 2024

Оглавление

	Вве	едение	3
1	Про	остейшие алгоритмы теории чисел	4
	1.	Простые числа	4
	2.	НОД и НОК двух чисел	6
	3.	НОД и НОК n чисел	8
	4.	Расширенный алгоритм Евклида	9
	5.	Линейные диофантовы уравнения с двумя неизвестными	11
	6.	Количество/сумма делителей числа	13
	7.	Степень вхождения простого числа в факториал	15

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;
}</pre>
```

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

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

Из этого следует, что если число N не делится ни на одно из чисел $2,3,4,\ldots,\lfloor\sqrt{N}\rfloor$, то оно не делится и ни на одно из чисел $\lceil\sqrt{N}\rceil+1,\ldots,N-2,N-1$, 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;
}</pre>
```

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

2. НОД и НОК двух чисел

Определение. Наибольший общий делитель (НОД) двух натуральных чисел a u b это наибольшее натуральное число d, которое делит оба числа a u b без остатка. Обозначается как d = HOД(a,b).

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

$$HOД(a,b) = HOД(a-b,b).$$

Доказательство. n = HOД(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. $n \le m = \text{HOД}(a,b)$. Аналогично, рассуждая получаем, что a - b : m и b : m = HOД(a,b). Отсюда a = (a - b) + b : m значит, m - делитель a и b, он не превосходит наибольшего и всех общих делителей чисел a и b, т.е. $m \le n = \text{HOД}(a,b)$. Таким образом получаем $m \le n \le m$, т.е. HOД(a,b) = n = m = HOД(a - b, b).

Ход алгоритма Евклида:

- 1. **Инициализация**: Начнем с двух целых чисел a и b, где $a \ge b$.
- 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 HOД(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, HOД(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 HOД(a,b).

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

Реализация алгоритма на С++:

```
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 (HOK) двух натуральных чисел a u b это наименьшее натуральное число l, которое делится на оба числа a u b без остатка. Обозначается как l = HOK(a, b).

Утверждение. $\forall a, b \in \mathbb{N} : HO\mathcal{J}(a, b) \cdot HOK(a, b) = ab.$

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 :

$$a = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k}$$
 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):

$$\mathrm{HOД}(a,b) = p_1^{\min\{n_1,m_1\}} p_2^{\min\{n_2,m_2\}} \dots p_k^{\min\{n_k,m_k\}}$$
 $\mathrm{HOK}(a,b) = p_1^{\max\{n_1,m_1\}} p_2^{\max\{n_2,m_2\}} \dots p_k^{\max\{n_k,m_k\}}$

То есть

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

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);
}
```

з. НОД и НОК п чисел

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

Ход алгоритма Евклида для п чисел:

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

$$d_1 = \text{HOД}(a_1, a_2)$$

3. Обновление НОДа для следующих чисел:

Для каждого i = 3, 4, ..., n:

$$d_i = \text{HOД}(d_{i-1}, a_i)$$

4. Конечный результат:

Наибольший общий делитель всех чисел: $HOД(a_1, a_2, \dots, a_n) = d_n$.

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 НО-Дом 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, di-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, \ldots, a_n .

Реализация данного алгоритма на С++:

```
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;
}
Aбсолютно 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;
}</pre>
```

4. Расширенный алгоритм Евклида

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

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 = HOД(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 HOД(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 ty;
  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);
}
```

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 HOД(a,b), значит c обязано делиться на HOД(a,b), чтобы решение существовало.

Если НОД $(a,b) \mid c$, то поделим обе части на этот НОД, получим новое уравнение a'x + b'y = c', где 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 $a'(x - x_0) = b'(y - y_0)$.

Так как HOД(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 HOД(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'.

Соответственно получам алгоритм решиния линейного диофантова уравнения с двумя неизвестными:

- 1. **Проверка существования решений:** Проверям, делится ли правая часть уравнения на HOД(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

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

Так выглядит реализация данного алгоритма на С++:

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

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 расмотрены в слудющих 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:

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

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

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

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

где $0 \le b_i \le a_i$ для i = 1, 2, ..., 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:

$$\tau(n) = (a_1 + 1) \times (a_2 + 1) \times \ldots \times (a_k + 1)$$

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

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

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

Сумма геометрической прогрессии имеет вид:

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

Подставляя это обратно в наше уравнение, получаем требуемое равенство.

7. Степень вхождения простого числа в факториал

Утверждение. Пусть $ord_p(n!)$ обозначает степень вхожедения простого числа p в n!. Тогда

$$ord_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$$

Доказательство.

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

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

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 $\left|\frac{n}{p^i}\right|$.

Таким образом,

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

Реализация данного алгоритма на С++: