Лекция Math [7]

Сегодня мы с вами продолжим разговор про математику. Рассмотрим простые формулы, которые у нас есть для целых чисел, поговорим о том, как быстро возводить числа в степень, делить в арифметике остатков, а также поверхностно про комбинаторику и теорию вероятностей.

Формулы количества и суммы делителей

Из основной теоремы арифметики мы знаем, что всякое натуральное число представимо в виде (каноническое разложение):

$$n = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}$$

Что мы можем получить из этого представления? Мы можем заметить, что количество делителей нашего числа n в точности равно

$$(\alpha_1 + 1) \cdot \ldots \cdot (\alpha_k + 1),$$

так как все делители получаются из комбинаций всех возможных альф в разложении числа от 0 до α_i для i-го множителя.

Хорошо, количество всех делителей мы умеем считать. А как насчёт их суммы? Давайте представим сумму всех делителей числа в виде следующего произведения:

$$(p_1^0 + p_1^1 + \ldots + p_1^{\alpha_1}) \cdot (p_2^0 + p_2^1 + \ldots + p_2^{\alpha_2}) \cdot \ldots \cdot (p_k^0 + p_k^1 + \ldots + p_k^{\alpha_k})$$

Оно действительно равно сумме всех делителей нашего числа. Чтобы убедиться в этом, достаточно перемножить все скобки. Там будет сумма:

$$p_1^0 \cdot p_2^0 \cdot \ldots \cdot p_{k-1}^0 \cdot p_k^0 + p_1^0 \cdot p_2^0 \cdot \ldots \cdot p_{k-1}^0 \cdot p_k^1 + \ldots + p_1^0 \cdot p_2^0 \cdot \ldots \cdot p_{k-1}^0 \cdot p_k^{\alpha_k} + \ldots\ldots + p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_{k-1}^{\alpha_{k-1}} \cdot p_k^{\alpha_k}$$

После этого преобразуем то наше произведение, вспомнив формулку суммы геометрической прогрессии:

$$\frac{p_1^{\alpha_1+1}-1}{p_1-1} \cdot \frac{p_2^{\alpha_2+1}-1}{p_2-1} \cdot \ldots \cdot \frac{p_k^{\alpha_k+1}-1}{p_k-1}$$

С помощью такого вида мы уже можем посчитать сумму всех делителей числа. Посмотрим, как это будет выглядеть в коде.

```
int sum_divs(int n) {
    int res = 1;
    for (int i = 2; i * i <= n; i++) {
        if (n % i != 0)
            continue;
        int al = 0;
        while (n % 2 == 0) {
            al++;
            n /= i;
        }
        res = res * (pow(i, al + 1) - 1) / (i - 1);
    }
    if (n > 1) {
        res = res * (pow(n, 2) - 1) / (n - 1);
    }
    return res;
}
```

Бинарное возведение в степень

Ладно, давайте перейдём к чему-то весёлому. Например, научимся быстро возводить числа в степени вроде 1000000 или больше. Посмотрим, на чём основывается наш метод.

Допустим мы хотим возвести a в степень 19. Представим 19 в двоичном виде: $19 = 10011_2$. Значит мы можем представить это число в виде суммы степеней двойки так:

$$19 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

.

Получается, $a^{19} = a^{16} \cdot a^2 \cdot a^1$. Как мы можем это посчитать?

Будем хранить ответ в переменной res, которая сначала проинициализирована единицей.

1. $19 \,\%\, 2 = 1$, значит $res = res \cdot a$, после чего умножаем a на себя: $a = a \cdot a$ и делим 19 на 2, чтобы получить следующий разряд бинарного представления.
2. $9 \,\%\, 2 = 1$, значит $res = res \cdot a$, затем $a = a \cdot a$ и делим 9 на 2.
3. $4 \,\%\, 2 = 0$, значит res не меняется, затем $a = a \cdot a$ и делим 4 на 2.
4. $2 \,\%\, 2 = 0$, значит res не меняется, затем $a = a \cdot a$ и делим 2 на 2.
5. $1 \,\%\, 2 = 1$, значит $res = res \cdot a$, затем $a = a \cdot a$ и делим 1 на 2.
6. Так как мы прошли все разряды бинарного представления числа, завершаем алгоритм.

В итоге $res = a^1 \cdot a^2 \cdot a^{16}$, что и требовалось.

Сложность. Сложность этого алгоритма (если полагать сложность умножения равной O(1)) равна количеству цифр в двоичном представлении степени n, а оно, в свою очередь, равно $\log n$. Значит, сложность равна $O(\log n)$.

Если вам не понятно это объяснение, можете почитать другое объяснение на E-maxx: `https://e-maxx.ru/algo/binary_pow`.

Давайте напишем функцию для бинарного возведения в степень:

```
int bin_pow(int a, int n) {
    int res = 1;
    while (n > 0) {
        if (n % 2 == 1)
            res = res * a;
        a = a * a;
        n /= 2;
    }
    return res;
}
```

Что здесь можно ещё добавить? Предположим, мы правда хотим возводить в очень большие степени, например 1000000. В *long long* числа в такой степени точно не влезут. Значит, скорее всего, нам нужно будет возводить числа по модулю какого-то числа. Добавим это в функцию.

```
int bin_pow(int a, int n, int mod) {
    int res = 1;
    while (n > 0) {
        if (n % 2 == 1)
            res = (res * a) % mod;
        a = (a * a) % mod;
        n /= 2;
    }
    return res;
}
```

Такой вот прикольный простой алгоритм.

Малая теорема Ферма и обратный элемент в кольце по простому модулю

Давайте теперь вспомним, что мы хотели делить числа в арифметике остатков.

Пусть мы умножили 2 на 4 по модулю 5:

$$2 \cdot 4 \equiv 3 \pmod 5$$

Тогда, навереное, мы хотели бы при делении 3 на 2 получить 4:

$$3 \,/\, 2 \equiv 4 \pmod 5$$

Вспомним, что $\dfrac{1}{a} = a^{-1}$. То есть обратный к a. А что мы знаем про обратные значения чисел? При умножении числа на обратное ему, мы получим единицу: $a \cdot a^{-1} = 1$.

В нашем примере выходит:

$$\begin{cases} 3 \cdot 2^{-1} \equiv 4 \pmod 5 \\ 2 \cdot 2^{-1} \equiv 1 \pmod 5 \end{cases}$$

Но чему равно обратное значение двойки по модулю 5? Здесь можно просто подобрать, так как значения маленькие, так мы получим ответ 3. Но если у нас были бы большие числа?

Можно ограничиться модулями, являющимися простыми числами, и воспользоваться теоремой Ферма:

Теорема. Если p простое и a не делится на p, то $a^{p-1} \equiv 1 \pmod p$.

Доказывать я здесь её не буду, так как это затянется. Кто хочет, можете почитать элементарное доказательство в википедии, оно понятное, если разобраться.

Что это нам даёт? Мы просто разобьём значение в левой части: $a^{p-1} = a \cdot a^{p-2} \equiv 1 \pmod p$. Что значит, обратный элемент по простому модулю от a равен a^{p-2}.

Либо мы ещё можем это получить, домножив в левой и правой части на a^{-1}: $a^{p-1} \cdot a^{-1} \equiv a^{p-2} \equiv a^{-1} \pmod p$

Значит, в программе обратный элемент по простому модулю можно найти, используя бинарное возведение в степень!

Расширенный алгоритм Евклида и обратный элемент в кольце по модулю

Давайте подумаем теперь, как нам действовать в случае, если модуль m не является простым числом.

У нас есть уравнение $a \cdot a^{-1} \equiv 1 \pmod{m}$. Его ещё можно записать так:

$$a \cdot a^{-1} = m \cdot y' + 1$$

Обозначим искомый обратный элемент a^{-1} за x. Перенеся всё, кроме единицы, в левую часть, и приняв $y = -y'$, получим:

$$a \cdot x + m \cdot y = 1$$

А это уже знакомое нам диофантово уравнение, которое мы научились решать с помощью расширенного алгоритма Евклида! Также из такого представления можно заключить, что обратный элемент имеют только числа $a \perp m$, то есть взаимно простые с модулем. Иначе если $gcd(a, b) = d > 1$, то левая часть делится на d, тогда и правая должна делиться. А там единица — противоречие.

Осталось только учесть, что полученный x мы должны взять по модулю, а он может быть отрицательным:

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

Сложность. Вспомним, что сложность расширенного алгоритма Евклида равна $O(\log \min(a, m))$, а так как $a < m$, здесь сложность будет равной $O(\log m)$.

Теорема Эйлера и обратный элемент в кольце по модулю

Конечно, можно находить обратный по любому модулю с помощью предыдущего метода. Но если нам не хочется реализовывать расширенный алгоритм Евклида, мы можем воспользоваться теоремой Эйлера.

Но перед тем, как написать её, введём понятие функции Эйлера.

Функция Эйлера

$\varphi(m)$ — это так называемая функция Эйлера, которая равна количеству натуральных чисел, меньших m, взаимнопростых с ним.

Посмотрим на несколько первых значений этой функции:

$$\varphi(1) = 1$$

$$\varphi(2) = 1$$

$$\varphi(3) = 2$$

$$\varphi(4) = 2$$

$$\varphi(5) = 4$$

$$\varphi(6) = 2$$

Но как её вычислять? Просматривать все числа до m звучит как плохой план. Тогда рассмотрим функцию Эйлера поглубже.

Свойства.

1. Если p — простое числоб $\varphi(p) = p - 1$.
2. Если p — простое, α — натуральное число, $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1}$. (Поскольку с числом p^α не взаимно просты только числа вида pk ($k \in \mathbb{N}$), которых $p^a/p = p^{\alpha-1}$ штук)
3. Мультипликативность: если $a \perp b$, то $\varphi(a \cdot b) = \varphi(a)\varphi(b)$. (Доказательство можете почитать, например, на википедии)

Снова вспомним каноническое разложение числа на простые множители:

$$n = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}$$

Из Него следует:

$$\varphi(n) = \varphi(p_1^{\alpha_1}) \cdot \ldots \cdot \varphi(p_k^{\alpha_k}) = (p_1^{\alpha_1} - p_1^{\alpha_1 - 1}) \cdot \ldots \cdot (p_k^{\alpha_k} - p_k^{\alpha_k - 1}) =$$

$$= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_k}\right)$$

Ура, теперь мы знаем формулку для вычисления функции Эйлера! А как это может выглядеть в коде? Идея — находить простые делители и поочерёдно вносить n в скобки полученной формулы.

```cpp
int euler(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i != 0)
            continue;
        while (n % i == 0) {
            n /= i;
        }
        res = (res - res / i);
    }
    if (n > 1) {
        res = (res - res / n);
    }
    return res;
}
```

Сложность. Помним, что мы такая факторизация имеет сложность $O(\sqrt{n})$. Но если потребуется много раз производить факторизацию, мы может воспользоваться решетом Эратосфена.

Теорема Эйлера

Теорема. Если $a \perp m$, то $a^{\varphi(m)} \equiv 1 \pmod{m}$.

Доказывать мы с вами, опять же не будем. Когда будете проходить теорию групп на дискретной математике, попросите преподавателя доказать это.

Умножим обе части на a^{-1}:

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

Так как мы уже умеем находить значение функции Эйлера и быстро возводить числа в степень по модулю, посчитать обратный элемент не представляет большую сложность.

```cpp
int inv_a = bin_pow(a, euler(mod) - 1, mod);
```

Базовая комбинаторика

Вот мы и закончили со сложной частью. Теперь в основном повторим то, что вы уже знаете.

Что наверное самое простое из комбинаторики?

Правило сложения.

Если элемент A можно выбрать n способами, а элемент B можно выбрать m, то выбрать A или B можно $n + m$ способами.

Правило умножения.

Если элемент A можно выбрать n способами, и при любом выборе A элемент B можно выбрвть m способами, то пару (A, B) можно выбрать $n \cdot m$ способами.

Что есть из более сложных вещей? Всякие числа перестановок, размещений и сочетаний. Последнее нам будет нужнее всего, поэтому первые два просто напомню, а на числе сочетаний остановимся подробнее.

Число перестановок.

Перестановка из n элементов — это упорядоченный набор из n различных чисел от 1 до n.

Количество различных перестановок из n элементов равно факториалу от n: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. Также отдельно указывается, что $0! = 1$.

Число размещений.

Сколькими способами можно разместить k элементов из последовательности $1, 2, \ldots, n$. Притом размещения, в которых будут одинаковые элементы, стоящие в разном порядке, считаются разными. Формула для вычисления этого дела не является секретом.

$$A_n^k = \frac{n!}{(n-k)!}$$

Число сочетаний

Под числом сочетаний понимается количество вариантов выбрать k элементов из некоторого множества, состоящего из n элементов. Формулку тоже, наверное, все знают. А кто не знает, откуда она берётся, почитайте где-нибудь. Важно понимание.

$$C_n^k = \frac{n!}{(n-k)!\,k!}$$

Какие важные свойства есть у числа сочетаний?

1. Правило симметрии:
$$C_n^k = C_n^{n-k}$$

2. Бином Ньютона:
$$(a+b)^n = C_n^0 a^0 b^n + C_n^1 a^1 b^{n-1} + \ldots + C_n^k a^k b^{n-k} + \ldots + C_n^n a^n b^0$$

3. Из бинома Ньютона следует:
$$2^n = (1+1)^n = C_n^0 + C_n^1 + \ldots + C_n^k + \ldots + C_n^n$$

4. Свойство из элемента из треугольника Паскаля:
$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

Давайте посмотрим, как можно считать число сочетаний в коде:

```
int C(int n, int k) {
    int res = 1;
    for (int i = k + 1; i <= n; i++)
        res = res * i;
    for (int i = 2; i <= n - k; i++)
        res = res / i;
    return res;
}
```

Это наивное решение работает за $O(n)$. Его можно немного улучшить, до $O(\min(k, n-k))$, используя даблы. Добавляем 0.1, на случай огругления вниз.

```
int C(int n, int k) {
    double res = 1;
    k = min(k, n - k);
    for (int i = 1; i <= k; i++) {
        res = res * (n - k + i) / i;
    }
    return int(res + 0.1);
}
```

Но вдруг нам нужно много раз считать число сочетаний для не очень больших n и k? Тогда мы можем воспользоваться связью с треугольником Паскаля!

```cpp
const int maxn = 1000; // для примера
vector<vector<int>> C(maxn + 1, vector<int>(maxn + 1));
for (int n = 0; n <= maxn; n++) {
    C[n][0] = C[n][n] = 1;
    for (int k = 1; k < n; k++) {
        C[n][k] = C[n - 1][k - 1] + C[n - 1][k];
    }
}
```

Сложность построения такого треугольника равна $O(maxn^2)$, зато ответ потом можно давать за $O(1)$.

Базовая теория вероятностей

Из теории вероятностей вы знаете, что вероятность какого-то события A равна количеству исходов, когда это событие происходит к количеству всех исходов в данном опыте.

$$P(A) = \frac{N_A}{N}$$

Это просто и понятно. Теперь давайте введём понятие математического ожидания, хоть оно и проходится не во всех школах, но в олимпиадных задачах встречается.

Пусть у нас есть какой-то опыт A из k событий $A_1, \ldots, A_k$. Значение каждого события — это какая-то циферка. Математическим ожиданием называется следующая величина:

$$M[A] = A_1 \cdot P(A_1) + \ldots + A_k \cdot P(A_k) = \sum_{i=1}^{k} A_i \cdot P(A_i)$$

Грубо говоря, она означает следующее: если проводить опыт много раз, то средним значением в нём будет $M[A]$.

Заметим, что мат. ожидание линейно: M[a · A + b] =

Примером является выпадение числа при броске кубика. A = {бросок кубика}.

$$M[A] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5$$