Биномиальные коэффициенты, бином Ньютона, треугольник Паскаля, теорема Люка, функция Мёбиуса, свертка Дирихле, поиск первообразного корня, xor-and-or-свертки

peltorator

4 апреля 2020. Tinkoff, A СПБ

1 Биномиальные коэффициенты: напоминание

/ Скорее всего вы все это уже знаете, но пробегитесь глазами по условиям лемм и теорем, чтобы убедиться. /

Задача. Коронавирус вынудил вас — покинуть свой дом. У вас дома a) небольшая библиотека из 5 книг b) большая библиотека из 100 книг, но забрать с собой вы можете только 3 из них. Сколько способов существует это сделать?

Чтобы решить эту задачу в общем случае, существуют биномиальные коэффициенты. $\binom{n}{k}$ или "цэ из эн по ка" — это количество способов выбрать k элементов из n. Давайте посчитаем, чему равна эта величина. Первым элементом мы можем взять любой из n, вторым любой из оставшихся, то есть $n-1$, и так далее. k-й предмет можно выбрать $n-k+1$ способом. Получается $n\cdot(n-1)\cdot\ldots\cdot(n-k+1)$ вариантов. Но это упорядоченная последовательность элементов (если вы взяли сначала томик Пушкина, а потом Кнута, то это будет не то же самое, что взять сначала Кнута, а потом Пушкина), так что надо не учитывать перестановки элементов выбранного множества. Есть $k!$ перестановок k-элементного множества, поэтому итоговым ответом будет $\frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$. Но это громоздкий ответ, давайте его немного упростим. $\frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!} =$ / домножим числитель и знаменатель на $(n-k)!$ / $= \frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)\cdot(n-k)!}{k!\cdot(n-k)!} =$ / в числителе на самом деле написано $n\cdot(n-1)\cdot\ldots\cdot2\cdot1$ / $= \frac{n!}{k!\cdot(n-k)!}$.

$\binom{n}{k}$ называется биномиальным коэффициентом, числом сочетаний, цэшкой.

Замечание. Очевидно, что если $k>n$, то $\binom{n}{k}=0$, потому что из n элементов нельзя выбрать больше, чем n.

Лемма. $\binom{n}{k}=\binom{n}{n-k}$.

Доказательство. Как часто бывает с сочетаниями, существует два доказательства этого факта: чисто комбинаторное и вычислительное. Приведем оба варианта:

1. Комбинаторное доказательство. Заметим, что выбрать k элементов — это то же самое, что выбрать, какие $n-k$ элементов мы не будем брать, так что эти величины совпадают.
2. Вычислительное доказательство. $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} = \frac{n!}{(n-k)!\cdot k!} = \frac{n!}{(n-k)!\cdot(n-(n-k))!} = \binom{n}{n-k}$.

□

Теорема. ***Бином Ньютона***

Бином[1] *Ньютона позволяет узнавать коэффициенты при одночленах после раскрытия скобок в степени суммы двух слагаемых:*

$$(a+b)^n = \sum_{k=0}^{n}\binom{n}{k}a^k b^{n-k}$$

Доказательство. $(a+b)^n = (a+b)\cdot(a+b)\cdot\ldots\cdot(a+b)$. Из каждого множителя нужно взять либо a, либо b, так что суммарная степень a и b всегда будет равна n. Осталось разобраться с коэффициентами. Чтобы получить степень k, нужно из n скобок в k выбрать a, это и есть $\binom{n}{k}$. Что и требовалось доказать.

□

[1] Частный случай полинома (многочлена), в котором два слагаемых (двучлен)

Лемма. $\sum\limits_{k=0}^{n} \binom{n}{k} = 2^n$.

Доказательство. Докажем этот факт двумя способами: комбинаторно и при помощи бинома Ньютона.

1. Комбинаторный способ. Всего есть 2^n способов взять сколько-то элементов из n (каждый элемент можно либо взять, либо не взять, то есть два варианта). С другой стороны, если мы взяли сколько-то элементов, то мы взяли либо 0, либо 1, ..., либо n элементов, так что это количество равно $\sum\limits_{k=0}^{n} \binom{n}{k}$.

2. Давайте воспользуемся биномом Ньютона при $a = b = 1$. $2^n = (1+1)^n = \sum\limits_{k=0}^{n} \binom{n}{k} \cdot 1^k \cdot 1^{n-k} = \sum\limits_{k=0}^{n} \binom{n}{k}$. □

Лемма. *При $n > 0$ верно тождество:* $\binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \ldots \pm \binom{n}{n} = 0$.

Доказательство. $\binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \ldots \pm \binom{n}{n} = \sum\limits_{k=0}^{n} (-1)^k \cdot \binom{n}{k} = \sum\limits_{k=0}^{n} (-1)^k \cdot 1^{n-k} \cdot \binom{n}{k} = (1-1)^n = 0^n = 0$. □

Лемма. $\sum\limits_{k=0}^{n} \binom{n}{k} \cdot a^k = (a+1)^n$.

Доказательство. Бином Ньютона. □

Формула для биномиальных коэффициентов очень удобна с математической точки зрения, но часто не очень подходит для программирования. Давайте познакомимся с рекурентной формулой.

Лемма. $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

Доказательство. Посмотрим на какой-нибудь элемент множества. Мы могли его либо взять, либо не взять. Если мы его возьмем, то из оставшихся $n-1$ элемента надо будет выбрать еще $k-1$, а если мы его не возьмем, то из оставшихся $n-1$ элемента надо будет выбрать еще k. Так что $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

/ Как и раньше, этот факт можно доказать вычислительно, но мы этого делать не будем / □

С помощью этого рекурентного соотношения можно написать программу, подсчитывающую биномиальные коэффициенты:

```
int c[N][N];

for (int n = 0; n < N; n++)
{
    c[n][0] = c[n][n] = 1;
    /* для них рекурентная формула не совсем работает, потому что одного из слагаемых не существует */
    for (int k = 1; k < n; k++)
        c[n][k] = c[n - 1][k] + c[n - 1][k - 1];
}
```

Все это также связано с треугольником Паскаля. Треугольник Паскаля — это треугольник, в котором на каждом уровне выписаны биномиальные коэффициенты:

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$$

1

1 1

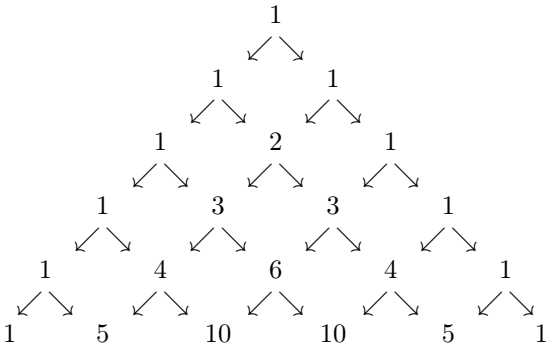
1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Воспользовавшись нашей рекурентной формулой можно заметить, что число в треугольнике паскаля равно сумме чисел, стоящих над ним справа и слева:



Замечание. Треугольник паскаля симметричен относительно вертикальной оси, потому что $\binom{n}{k} = \binom{n}{n-k}$.

Замечание. Числа в одной строке сначала возрастают до середины, а потом убывают.

Доказательство. Для доказательства этого факта сравним два соседних числа в одной строке: $\binom{n}{k}$ и $\binom{n}{k+1}$. Заметим, что $\binom{n}{k+1} = \frac{n!}{(k+1)!\cdot(n-k-1)!} = \frac{n!}{k!\cdot(n-k)!} \cdot \frac{n-k}{k+1} = \binom{n}{k} \cdot \frac{n-k}{k+1}$. Так что следующий коэффициент больше, если $\frac{n-k}{k+1} > 1$, и меньше, если $\frac{n-k}{k+1} < 1$. Иными словами, это зависит от того, кто больше: $n-k$ или $k+1$. $n-k > k+1 \Leftrightarrow \frac{n-1}{2} > k$. Это и значит, что числа до середины возрастают, а потом убывают. □

2 Биномиальные коэффициенты: новое

Мы выяснили, что среди $\binom{n}{k}$ самым большим является $\binom{n}{\lfloor \frac{n}{2} \rfloor}$. Давайте поймем, как быстро растет эта величина, чтобы не получить случайно переполнение инта. Для этого воспользуемся формулой Стирлинга (хорошо бы ее запомнить, она часто используется для оценок).

Теорема. ***Формула Стирлинга*** $n! \sim \sqrt{2\pi n} \cdot (\frac{n}{e})^n$, *где* $e = 2,718\ldots$

/ Эквивалентность означает, что при $n \to +\infty$ отношение этих двух величин стремится к единице. Для простоты можно понимать это в значении "примерно равно". /

Теорема. $\binom{n}{\frac{n}{2}} \sim \frac{2^n \cdot \sqrt{2}}{\sqrt{\pi n}}$.

Доказательство. Подставим формулу Стирлинга в формулу биномиального коэффициента:

$$\binom{n}{\frac{n}{2}} = \frac{n!}{(\frac{n}{2})! \cdot (\frac{n}{2})!} \sim \frac{\sqrt{2\pi n} \cdot (\frac{n}{e})^n}{\sqrt{2\pi \frac{n}{2}} \cdot (\frac{n}{2e})^{\frac{n}{2}} \cdot \sqrt{2\pi \frac{n}{2}} \cdot (\frac{n}{2e})^{\frac{n}{2}}} = \frac{\sqrt{2} \cdot \sqrt{\pi n} \cdot (\frac{n}{e})^n}{\sqrt{\pi n} \cdot \sqrt{\pi n} \cdot (\frac{n}{2e})^n} = \frac{2^n \sqrt{2}}{\sqrt{\pi n}}$$

□

Замечание. Вряд ли вам часто надо будет оценивать биномиальный коэффициент с точностью до константы, так что можно считать, что $\binom{n}{\lfloor \frac{n}{2} \rfloor} \sim C \cdot \frac{2^n}{\sqrt{n}}$ или даже 2^n (но это грубо, конечно).

Пример. Какие биномиальные коэффициенты влезут в базовые типы данных?

$\binom{33}{17} \sim 1,2 \cdot 10^9$ (последнее, которое влезает в int),

$\binom{34}{17} \sim 2,3 \cdot 10^9$ (последнее, которое влезает в unsigned int),

$\binom{66}{33} \sim 1,8 \cdot 10^{18}$ (последнее, которое влезает в long long),

$\binom{67}{33} \sim 1,4 \cdot 10^{19}$ (последнее, которое влезает в unsigned long long).

Замечание. Если считать биномиальные коэффициенты не рекурентно, а как отношение факториалов, то случится переполнение раньше, к примеру $21! \sim 5,1 \cdot 10^{19}$, так что даже $\binom{21}{1}$ не получится посчитать.

Биномиальные коэффициенты часто встречаются в комбинаторике, когда надо посчитать количество каких-то объектов или какую-то комбинаторную вероятность. В таких задачах обычно нужно ответ считать по модулю. Для этого прекрасно подойдет рекурентная формула.

Но что делать, если n — большое число, и у нас нет возможности сделать такой большой предпосчет? Можно попытаться как-то воспользоваться формулой через факториалы (факториалы можно посчитать за линейное время, а для подсчета матрицы биномиальных коэффициентов нам понадобится квадратичное время, так что это хороший выигрыш во времени). Для этого мы посчитаем массив факториалов, а также массив обратных факториалов, после чего вычисление биномиального коэффициента будет происходить за константное время. Но если число n не меньше, чем модуль, то начинаются проблемы с делением.

Действительно, $n! \equiv 0 \pmod p$, если $n \geqslant p$, потому что $n! = 1 \cdot 2 \cdot \ldots \cdot p \cdot \ldots \cdot n$, так что получится какая-то дребедень: и числитель, и знаменатель равны нулю! Чтобы побороть эту проблему, можно отдельно считать $n! \pmod p$ без вхождений p в $n!$, а также отдельно подсчитывать степень вхождения p в $n!$, а потом понимать, что со всем этим происходит при делении, но это достаточно сложно. В этой непростой ситуации нам поможет теорема Люка.

Теорема. ***Люка***

$\binom{n}{m} \equiv \prod\limits_{i=0}^{k-1} \binom{n_i}{m_i} \pmod p$, *где* m *и* n *— не более, чем* k*-значные числа в* p*-ричной системе счисления и* $m = (m_{k-1}, \ldots, m_0)_p$ *и* $n = (n_{k-1}, \ldots, n_0)_p$.

/ Обратите внимание, что среди множителей могут быть цэшки, у которых $n_i < m_i$*. В таком случае ответ равен нулю. /*

Маленькие биномиальные коэффициенты легко посчитать при помощи факториалов, потому что $n_i, m_i < p$. Количество цифр в числе равно логарифму, так что время работы алгоритма, использующего теорему Люка будет равно $O(\log_p n)$ (при условии, что заранее были посчитаны факториалы и обратные факториалы за $O(p)$).

Лемма. $(a+b)^p \equiv a^p + b^p \pmod p$

/ Это же просто мечта! Степень суммы равна сумме степеней. /

Доказательство. Воспользуемся биномом Ньютона:

$(a+b)^p = \sum\limits_{k=0}^{p} \binom{p}{k} \cdot a^k \cdot b^{p-k}$. При этом все биномиальные коэффициенты кроме первого и последнего делятся на p, так что они исчезнут, останется только $a^p + b^p$. □

Доказательство. теоремы Люка:

Рассмотрим коэффициент при x^m в многочлене $(x+1)^n$. С одной стороны, он равен $\binom{n}{m}$ по биному Ньютона. С другой стороны $(x+1)^n = \prod\limits_{i=0}^{k-1} (x+1)^{n_i \cdot p^i} = \prod\limits_{i=0}^{k-1} ((x+1)^{p^i})^{n_i} \equiv$ / по лемме / $\equiv \prod\limits_{i=0}^{k-1} (x^{p^i} + 1^{p^i})^{n_i} = \prod\limits_{i=0}^{k-1} (x^{p^i} + 1)^{n_i}$. Единственный способ получить степень m — это взять степень $m_i \cdot p^i$ из i-го множителя. Так что по биному Ньютона коэффициент равен $\prod\limits_{i=0}^{k-1} \binom{n_i}{m_i}$. Что и требовалось доказать. □

Свойства. *Некоторые неочевидные тождества с биномиальными коэффициентами:*

1. $\sum\limits_{k=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2k} = \sum\limits_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n}{2k+1} = 2^{n-1}$.
2. $\sum\limits_{k=0}^{n} k \cdot \binom{n}{k} = n \cdot 2^{n-1}$.
3. $\sum\limits_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$.

4. $\sum\limits_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$.
5. $\sum\limits_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$.

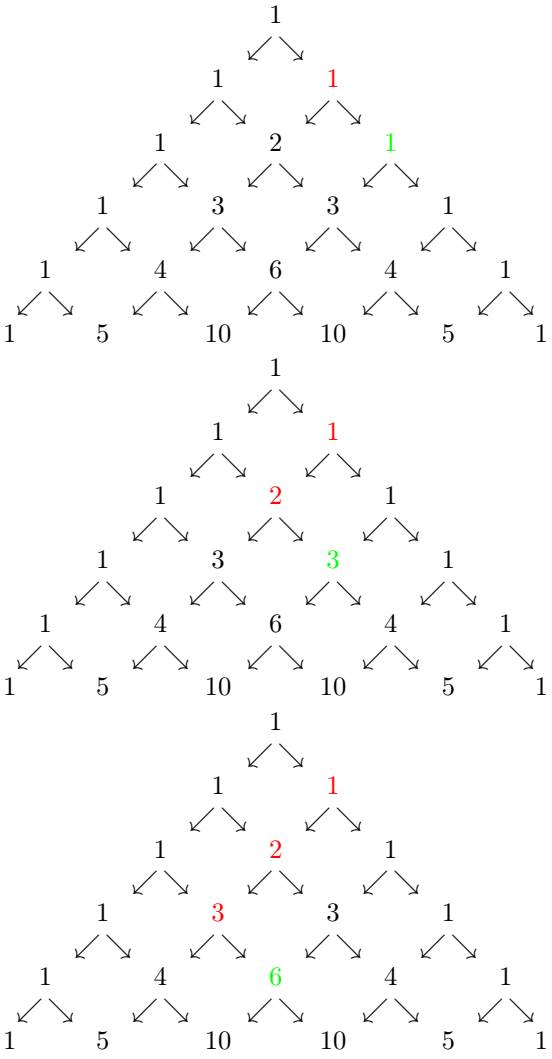
Доказательство. 1. Мы знаем уже, что $\sum\limits_{k=0}^{n} (-1)^k \cdot \binom{n}{k} = 0$, так что сумма четных цэшек равна сумме нечетных. Но сумма всех цэшек равна 2^n, так что отдельно каждых будет по 2^{n-1}.

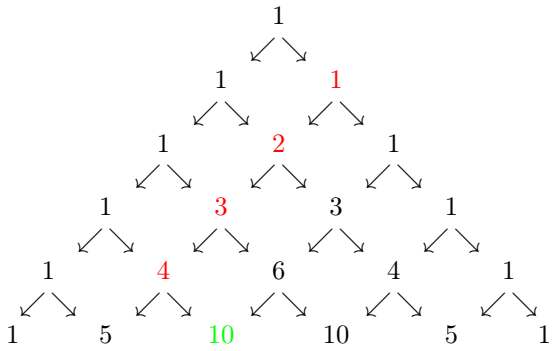
2. Заметим, что $k \cdot \binom{n}{k} = k \cdot \frac{n!}{k!\cdot(n-k)!} = \frac{n!}{(k-1)!\cdot(n-k)!} = n \cdot \frac{(n-1)!}{(k-1)!\cdot((n-1)-(k-1))!} = n \cdot \binom{n-1}{k-1}$ (при $k = 0$ этот член просто равен нулю). Поэтому наша сумма равна $\sum\limits_{k=1}^{n} n \cdot \binom{n-1}{k-1} = n \cdot \sum\limits_{k=0}^{n-1} \binom{n-1}{k} = n \cdot 2^{n-1}$.

Альтернативно это можно доказать так: мы выбираем команду, а в ней капитана. С одной стороны это сумма $k \cdot \binom{n}{k}$, а с другой строоны можно поступить другим образом: сначала выбрать капитана (n), а потом он выберет себе произвольную команду из оставшихся человек (2^{n-1}).

3. Докажем комбинаторно. То, что стоит слева, — это количество способов выбрать n объектов из $2n$. Пусть у нас есть $2 \cdot n$ объектов. Поделим их на две части по n. Тогда что такое выбрать n из них? Это либо выбрать 0 из первой половины и n из второй, либо выбрать 1 из первой половины и $n-1$ из второй и т.д. Запишем это формулой: $\binom{2n}{n} = \sum\limits_{k=0}^{n} \binom{n}{k} \cdot \binom{n}{n-k} = $ / ведь $\binom{n}{n-k} = \binom{n}{k}$ / $= \sum\limits_{k=0}^{n} \binom{n}{k}^2$.

4. Можно обрезать сумму до $\sum\limits_{m=k}^{n} \binom{m}{k}$, потому что остальные члены равны нулю. Такая сумма — это сумма какой-то диагонали треугольника Паскаля (идущей справа налево вниз, но треугольник симметричен, так что это неважно). Проиллюстрируем это тождество на картинке, в треугольнике Паскаля. Сумма красных чисел должна быть равна зеленому.





Из картинки сразу видно решение по индукции: $\sum\limits_{m=k}^{n} \binom{m}{k} = \sum\limits_{m=k}^{n-1} \binom{m}{k} + \binom{n}{k} =$ / по предположению индукции / $= \binom{n}{k+1} + \binom{n}{k} = \binom{n+1}{k+1}$.

Доказательство базы остается читателю в качестве упражнения.

Это весьма полезное тождество, которое позволяет посчитать количество способов выбрать фиксированное количество элементов, если размер множества, из которого они выбираются, может варьироваться.

5. Следствие предыдущего свойства. Необходимо просто заметить, что $\binom{n+k}{k} = \binom{n+k}{n}$ и $\binom{n+m+1}{m} = \binom{n+m+1}{n+1}$. □

Упражнение. Используя четвертое свойство, найдите формулу для $\sum\limits_{k=1}^{n} k$, $\sum\limits_{k=1}^{n} k^2$ и $\sum\limits_{k=1}^{n} k^3$.

Если вы застряли в своих преобразованиях, просто посмотрите на какой-нибудь такой список формул и подумайте, какая формула вам может помочь.

3 Свертки

Рассмотрим различные свертки функций. Функции мы будем рассматривать арифметические, то есть действующие из $\mathbb{N}$ в $\mathbb{R}$ (или $\mathbb{C}$). В каком-то смысле можно думать про арифметические функции как про последовательности чисел: $f(1), f(2), \ldots, f(n), \ldots$.

Мы проходили уже одну свертку на прошлом занятии. Это свертка умножения: $c_n = \sum\limits_{k=1}^{n-1} a_k \cdot b_{n-k}$.

FFT как раз строило по последовательностям a и b их свертку.

В этом параграфе мы рассмотрим несколько примеров других сверток.

Напоминание. Функция Мёбиуса. $\mu(n) = \begin{cases} 0, & \text{если } n \text{ не свободно от квадратов,} \\ (-1)^k, & \text{если } n \text{ свободно от квадратов, и у него } k \text{ простых делителей} \end{cases}$

/ Число свободно от квадратов, если все простые входят в его разложение в степени один /

Напоминание. Функция Эйлера $\varphi(n)$ — это количество чисел, не больших n, которые взаимно просты с ним.

Часто бывает так, что две функции связаны между собой следующим отношением:

$$g(n) = \sum_{d|n} f(d)$$

/ Запись $d|n$ означает "d делит n", то есть сумма берется по всем делителям числа n /

Примеры. 1. Количество делителей числа: $\tau(n) = \sum\limits_{d|n} 1$.

2. Сумма делителей числа: $\sigma(n) = \sum\limits_{d|n} d$.

3. Функция, которая равна единице в единице и нулю во всех остальных натуральных числах: $\chi_1 = \sum\limits_{d|n} \mu(d)$.

Доказательство. 3. Пусть $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}$. Если $\mu(d) \neq 0$, то каждую p-шку мы в нем взяли либо в нулевой степени, либо в первой. Тогда есть $\sum\limits_{i=0}^{\frac{k}{2}} \binom{k}{2i}$ d-шек, для которых $\mu(d) = 1$ и $\sum\limits_{i=0}^{\frac{k-1}{2}} \binom{k}{2i+1}$ d-шек, для которых $\mu(d) = -1$. А такие суммы равны при $n > 1$ (обсуждалось в прошлом параграфе), так что вся сумма равна нулю. При $n = 1$ легко произвести подстановку и проверить, что $\chi_1(1) = 1$. □

Замечание. Есть формула для функции Эйлера, которая имеет похожий вид:
$\varphi(n) = n \prod\limits_{p|n}(1 - \frac{1}{p})$, где произведение берется по всем простым делителям n.

Напоминание. Арифметическая функция f называется мультипликативной, если $f(a \cdot b) = f(a) \cdot f(b)\ \forall\ a, b \in \mathbb{N} : gcd(a, b) = 1$.

Замечание. $\mu, \varphi, \tau, \sigma, \chi_1, id, 1$ — мультипликативные арифметические функции. / 1 — это фунция, равная единице во всех точках /

Теорема. ***Формула обращения Мёбиуса***
Пусть f и g — арифметические функции. Тогда
$g(n) = \sum\limits_{d|n} f(d) \Leftrightarrow f(n) = \sum\limits_{d|n} \mu(d) g\left(\frac{n}{d}\right)$.

Доказательство. $\sum\limits_{d|n} \mu(d) g\left(\frac{n}{d}\right) =$ / по левой формуле, которая у нас уже есть / $= \sum\limits_{d|n} \mu(d) \left(\sum\limits_{d' | \frac{n}{d}} f(d') \right) = \sum\limits_{d, d' : (d \cdot d') | n} \mu(d) \cdot f(d') =$ / поменяем местами просто обозначения / $= \sum\limits_{d, d' : (d \cdot d') | n} \mu(d') \cdot f(d) = \sum\limits_{d|n} g(d) \left(\sum\limits_{d' | \frac{n}{d}} \mu(d') \right) =$ / по примеру 3 / $= \sum\limits_{d|n} g(d) \chi_1(\frac{n}{d}) = g(n)$ (у остальных будет множитель 0).
В обратную сторону аналогично. □

Примеры. Подставим в эту формулу примеры выше:
1. $1 = \sum\limits_{d|n} \mu(d) \tau\left(\frac{n}{d}\right)$.
2. $n = \sum\limits_{d|n} \mu(d) \sigma\left(\frac{n}{d}\right)$.
3. $\mu(n) = \sum\limits_{d|n} \mu(d) \chi_1\left(\frac{n}{d}\right)$ (что очевидно, но показывает, что формула из третьего примера является частным случаем формулы обращения Мёбиуса).

Пример. Мы знаем, что $n = \sum\limits_{d|n} \varphi(d)$ (кстати, помните, почему?). Применим формулу обращения Мёбиуса и получим, что $\varphi(n) = \sum\limits_{d|n} \mu(d) \cdot \frac{n}{d}$.
Неожиданно получили связь функций Эйлера и Мёбиуса.

$\sum\limits_{d|n} \mu(d) g\left(\frac{n}{d}\right)$ является частным случаем свертки Дирихле.

Определение. Свертка Дирихле двух арифметических функций определяется следующим образом:
$(f * g)(n) = \sum\limits_{d|n} f(d) g\left(\frac{n}{d}\right) = \sum\limits_{ab=n} f(a) g(b)$.

Свойства. *1.* $(f * g) * h = f * (g * h)$.
2. $f * g = g * f$.
3. $f * (g + h) = f * g + f * h$.
4. $f * \chi_1 = \chi_1 * f = f$.
5. $f * 0 = 0 * f = 0$ *(0 — это функция, которая всегда равна нулю).*
6. Если f, g — мультипликативные, то $f * g$ тоже является мультипликативной.

Доказательство. 1-5. Очевидно. Подставьте.
6. $gcd(n, m) = 1$, $(f*g)(n \cdot m) = \sum\limits_{d | n \cdot m} f(d) g(\frac{n \cdot m}{d}) =$ / здесь мы пользуемся тем, что числа взаимно просты. Иначе мы бы получили сумму с повторениями / $= \sum\limits_{d_1 | n, d_2 | m} f(d_1 \cdot d_2) g(\frac{n \cdot m}{d_1 \cdot d_2}) =$ / а здесь тем, что числа делители взаимно простых чисел взаимно просты / $= \sum\limits_{d_1 | n, d_2 | m} f(d_1) f(d_2) g(\frac{n}{d_1}) g(\frac{m}{d_2}) = (\sum\limits_{d_1 | n} f(d_1) g(\frac{n}{d_1})) \cdot (\sum\limits_{d_2 | m} f(d_2) g(\frac{m}{d_2})) = (f*g)(n) \cdot (f*g)(m)$. □

Получили, что арифметические функции образуют коммутативное кольцо с единицей относительно свертки Дирихле и сложения. Нулем в этом кольце является тождественно нулевая функция, а единицей χ_1. Это кольцо называется кольцом Дирихле.

Примеры. a. $\tau = 1 * 1$.
b. $\sigma = id * 1$.
c. $\chi_1 = 1 * \mu$.

d. $1 = \tau * \mu$.
e. $id = \sigma * \mu$.
f. $id = \varphi * 1$.
g. $\sigma = \varphi * \tau$.
/ Здесь 1 — это функция, которая всегда равна единице /

Определение. Обращением Дирихле функции f назовем такую функцию g, что $f * g = \chi_1$, то есть фунция $g = f^{-1}$ в кольце Дирихле.

Теорема. $\forall\ f : f(1) \neq 0\ \exists\ g : f * g = \chi_1$.

Доказательство. $g(1) = \frac{1}{f(1)}$.
А для $n > 1$ функцию g можно вычислить рекурсивно:
$g(n) = -\frac{1}{f(1)} \sum\limits_{d|n, d<n} f(\frac{n}{d}) g(d)$.
Доказывается, что такая функция подходит, домножением на $f(1)$ и переносом всего в левую часть. □

Зачем это все нужно?
Рассотрим, как решать задачи при помощи обращения Мёбиуса и свертки Дирихле.

Обозначение. $[P]$ — это функция, которая равна единице, если P верно, и нулю, если P неверно.

Пример. Дано число n. Надо найти количество упорядоченных пар взаимно простых чисел $x, y \leqslant n$. Обозначим ответ за $f(n)$.

Сначала заметим, что $f(n) = 2 \cdot \sum\limits_{k=1}^{n} \varphi(k) - 1$, потому что каждую неупорядоченную пару x, y мы посчитаем один раз в $\varphi(max(x, y))$, а нам надо посчитать упорядоенные пары. Осталось вычесть пары из двух одинаковых чисел. Но число взаимно просто с собой только если оно равно единице.

Запишем формулу: $f(n) = \sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} [gcd(i, j) == 1]$. Заметим, что $[gcd(i, j) == 1] = \chi_1(gcd(i, j))$, так что можно применить свертку Мёбиуса: $\sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} \sum\limits_{d|gcd(i,j)} \mu(d) = \sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} \sum\limits_{d=1}^{n} [d|gcd(i, j)] \cdot \mu(d) = \sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} \sum\limits_{d=1}^{n} [d|i] \cdot [d|j] \cdot \mu(d) =$ / меняем порядок суммирования / $\sum\limits_{d=1}^{n} \mu(d) \left(\sum\limits_{i=1}^{n} [d|i] \right) \left(\sum\limits_{j=1}^{n} [d|j] \right)$. При этом $\sum\limits_{i=1}^{n} [d|i] = \sum\limits_{j=1}^{n} [d|j] =$ количеству чисел, делящихся на d, то есть $\left\lfloor \frac{n}{d} \right\rfloor$. Тогда получается, что $f(n) = \sum\limits_{d=1}^{n} \mu(d) \cdot \left\lfloor \frac{n}{d} \right\rfloor^2$.

А эту формулу можно посчитать уже за линейное время (все значения μ считаются при помощи линейного решета за $O(n)$, это было у нас в начале года).

Нырнем глубже! Мы хотим еще быстрее! Пусть у нас есть некоторая мультипликативная функция f, и мы хотим посчитать ее префиксную сумму. Обозначим $s_f(n) = \sum\limits_{k=1}^{n} f(k)$. Предположим, что мы обнаружили такую функцию g, что s_g и s_{f*g} можно быстро считать. Тогда научимся быстро считать s_f:

$$s_{f*g}(n) = \sum_{k=1}^{n} \sum_{d|k} g(d) f\left(\frac{k}{d}\right) = \sum_{d=1}^{n} g(d) \sum_{k=1}^{\left\lfloor \frac{n}{d} \right\rfloor} f(k) = g(1) \sum_{k=1}^{n} f(k) + \sum_{d=2}^{n} g(d) \sum_{k=1}^{\left\lfloor \frac{n}{d} \right\rfloor} f(k) = g(1) s_f(n) + \sum_{d=2}^{n} g(d) s_f(\left\lfloor \frac{n}{d} \right\rfloor)$$

Теперь перенесем $s_f(n)$ в одну часть, а все остальное в другую:

$$s_f(n) = \frac{s_{f*g}(n) - \sum_{d=2}^{n} s_f\left(\left\lfloor \frac{n}{d} \right\rfloor\right) g(d)}{g(1)}$$

Осталось научиться быстро считать сумму в числителе.

Лемма. *Среди чисел* $\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor$ *не более* $2\sqrt{n}$ *различных.*

Доказательство. Стандартное доказательство. Есть $\sqrt{n}$ чисел вида $\left\lfloor \frac{n}{d} \right\rfloor$, где $d < \sqrt{n}$, а для $d \geqslant \sqrt{n}$ будет выполнено $\left\lfloor \frac{n}{d} \right\rfloor \leqslant \sqrt{n}$, поэтому среди них тоже не больше, чем $\sqrt{n}$ различных.
Вот алгоритм перебора отрезков одинаковых значений $\left\lfloor \frac{n}{d} \right\rfloor$:

```
for (int left = 1, right; left <= n; left = right + 1)
{
    right = n / (int)(n / left); // В отрезке [left; right] одинаковые значения n/d
}
```

□

Из леммы следует, что сумма в числителе разбивается на $O(\sqrt{n})$ рекурсивных вызовов $s_f(\lfloor \frac{n}{d} \rfloor)$ и подсчетов g на отрезке, но сумма g на отрезке — это разность двух префиксных сумм, которые мы по предположению задачи умеем быстро считать.

Получился рекурсивный алгоритм. Соптимизируем его, сохраняя уже посчитанные значения $s_f(k)$ в хэш-мап, чтобы не считать их заново.

Лемма. $\left\lfloor \frac{\lfloor \frac{a}{b} \rfloor}{c} \right\rfloor = \lfloor \frac{a}{bc} \rfloor$.

Доказательство. Очевидно, что $0 \leqslant \frac{a}{b} - \lfloor \frac{a}{b} \rfloor < 1$, поэтому $0 \leqslant \frac{a}{bc} - \frac{\lfloor \frac{a}{b} \rfloor}{c} < \frac{1}{c}$. При этом $\lfloor \frac{a}{b} \rfloor$ — целое число, поэтому дробная часть $\frac{\lfloor \frac{a}{b} \rfloor}{c}$ не больше $\frac{c-1}{c}$, так что при прибавлении чего-то меньшего, чем $\frac{1}{c}$, мы не перепрыгнем на следующую целую часть. □

Из леммы слеудет, что за время алгоритма мы посетим только числа вида $\lfloor \frac{n}{d} \rfloor$, потому что $\left\lfloor \frac{\lfloor \frac{n}{d_1} \rfloor}{d_2} \right\rfloor = \lfloor \frac{n}{d_1 d_2} \rfloor$.

Подсчет $s_f(k)$ занимает $O(\sqrt{k})$ времени + рекурсивные вызовы. Так что итоговая асимптотика будет равна $O(\sum\limits_{k=1}^{n} \sqrt{\lfloor \frac{n}{k} \rfloor}) =$ / сумма берется по различным значениям / $\leqslant O(\sum\limits_{k=1}^{\sqrt{n}} \sqrt{k} + \sum\limits_{k=1}^{\sqrt{n}} \sqrt{\frac{n}{k}})$. Когда в асимптотике есть сумма возрастающей/убывающей функции, это сумму можно заменить на интеграл без потерь.

$\sum\limits_{k=1}^{\sqrt{n}} \sqrt{k} \sim \int\limits_{1}^{\sqrt{n}} \sqrt{k}dk = \frac{2}{3}k^{\frac{3}{2}}|_1^{\sqrt{n}} = O(n^{\frac{3}{4}})$

$\sum\limits_{k=1}^{\sqrt{n}} \sqrt{\frac{n}{k}} \sim \int\limits_{1}^{\sqrt{n}} \sqrt{\frac{n}{k}}dk = 2\sqrt{nk}|_1^{\sqrt{n}} = O(n^{\frac{3}{4}})$

Получили, что асимптотика — $O(n^{\frac{3}{4}})$.

Самое время вспомнить о том, что функция f мультипликативная. А для мультипликативных функций мы умеем считать ее первые k значений за $O(k)$. Тогда и префиксные суммы мы тоже можем посчитать за $O(k)$. Пусть мы предпосчитали префиксные суммы для первых $k \geqslant \sqrt{n}$ чисел. Тогда нам надо брать сумму времен работ только для таких $\lfloor \frac{n}{d} \rfloor$, которые больше k. Получаем время работы $O(k + \sum\limits_{i=1}^{\frac{n}{k}} \sqrt{\frac{n}{i}}) = O(k + \frac{n}{\sqrt{k}})$ (этот интеграл мы уже брали, надо подставить только в другой точке). Минимума эта величина достигает при $k = O(n^{\frac{2}{3}})$. В этом случае асимптотика получается равна $O(n^{\frac{2}{3}})$.

Ура, мы научились искать префиксную сумму мультипликативной фунции f за $O(n^{\frac{2}{3}})$, если есть такая функция g, что s_g и s_{f*g} мы умеем считать за $O(1)$.

Откуда же взять эту функцию g? Мы знаем несколько примеров таких функций: $\chi_1, id, 1$. Надо поперебирать эти фунции, пока их свертка с f не получится равной тоже какой-то простой функции. Если не нашлось, то очень жаль, ничего не получилось.

Пример. Вернемся к нашей задаче поиска количества упорядоченных пар взаимно простых чисел $x, y \leqslant n$.

Мы уже выяснили, что $f(n) = \sum\limits_{d=1}^{n} \mu(d) \lfloor \frac{n}{d} \rfloor^2$. Как мы уже поняли, среди чисел вида $\lfloor \frac{n}{d} \rfloor$ всего $\sqrt{n}$ различных. А μ является мультипликативной функцией, при этом $\mu * 1 = \chi_1$. Так что сумму μ на отрезке мы умеем считать быстро. Посмотрим, в каких точках m нам надо будет считать $s_\mu(m)$. Это будут такие m, что на них число $\lfloor \frac{n}{m} \rfloor$ увеличилось. Нетрудно понять, что это будут как раз числа вида $\lfloor \frac{n}{k} \rfloor$. Поэтому нам надо будет посчитать s_μ ровно в точках вида $\lfloor \frac{n}{k} \rfloor$, а это мы умеем делать суммарно за $O(n^{\frac{2}{3}})$ (только хэшмап не обнуляйте).

Пример. $\varphi * 1 = id$. Так что s_φ мы тоже можем считать за $O(n^{\frac{2}{3}})$.

4 Поиск первообразного корня

Определение. Первообразный корень по модулю m — это такое число g, что $\forall\ n : gcd(n, m) = 1\ \exists\ k \geqslant 0 : g^k \equiv n \pmod{m}$.

Иными словами, степени g пробегают все возможные вычеты, которые взаимно просты с m.

Замечание. Если m — простое число, то степени первообразного корня пробегают все ненулевые вычеты.

Теорема. *По модулю m существует первообразный корень* $\Leftrightarrow m = \begin{cases} 1 \\ 2 \\ 4 \\ p^k, \text{ где } p \text{ — нечетное простое} \\ 2 \cdot p^k, \text{ где } p \text{ — нечетное простое} \end{cases}$

Напоминание. Показатель n по модулю m — это такое минимальное натуральное число k, что $n^k \equiv 1 \pmod{m}$.

Замечание. Для натурального числа n существует показатель тогда и только тогда, когда $gcd(n, m) = 1$.

Теорема. *g — первообразный корень по модулю m $\Leftrightarrow$ показатель g по модулю m равен $\varphi(m)$.*

Доказательство. Во-первых, если показатель меньше, чем $\varphi(m)$, то степени g не успеют пробежать все $\varphi(m)$ взаимно простых с m вычетов перед тем, как зациклиться, с другой стороны показатель не может быть больше $\varphi(m)$, потому что больше нечего пробегать, а зациклиться до единицы по очевидным причинам нельзя.

И в обратную сторону: если показатель равен $\varphi(m)$, то мы пробежим все возможные взаимно простые остатки, потому что нельзя зациклиться до единицы. □

Получаем наивный алгоритм проверки числа на то, что оно является первообразным корнем: перебрать его первые $\varphi(m) - 1$ степеней и проверить, что среди них нет единицы.

Но это достаточно долго. Давайте придумаем более быстрый алгоритм. Заметим, что взаимно простые с m остатки образуют группу по умножению [2] , а степени какого-то конкретного вычета образуют подгруппу ($\{g^k | k \in \mathbb{N}_0\}$). По теореме Лагранжа размер подгруппы делит размер группы, так что показатель любого элемента делит $\varphi(m)$. Поэтому можно не проверять все степени на неравенство единице, а только делители числа $\varphi(m)$.

Можно пойти дальше и еще сильнее соптимизировать алгоритм. Пусть $d|\varphi(m)$, $d \neq \varphi(m)$ и $g^d \equiv 1 \pmod{m}$. Пусть $p|\frac{\varphi(m)}{d}$. Тогда заметим, что $d|\frac{\varphi(m)}{p}$, поэтому $g^{\frac{\varphi(m)}{d}} = g^{d \cdot k} = (g^d)^k \equiv 1^k \equiv 1 \pmod{p}$, поэтому проверять можно только степени вида $\frac{\varphi(m)}{p_i}$, где p_i — простые делители числа $\varphi(m)$.

Различных простых делителей у числа не больше двоичного логарифма, поэтому если использовать бинарное возведение в степень, то алгоритм проверки числа на то, что оно является первообразным корнем, будет работать за $O(\log m \cdot \log \varphi(m))$, если заранее факторизовать число $\varphi(m)$ и найти все его простые делители. Напишем код:

```
int M;
int PHI; // phi(M)

int binpow(int a, int b)
{
    int ans = 1;
    while (b)
    {
        if (b & 1)
            ans = ans * 1LL * a % M;
        b >>= 1;
        a = a * 1LL * a % M;
    }
    return ans;
}

vector<int> phi_prime_divisors; // простые делители phi(M) посчитаны заранее

bool check(int g)
{
    if (gcd(g, M) != 1)
        return false;
    for (int p : phi_prime_divisors)
        if (binpow(g, PHI / p) == 1)
            return false;
    return true;
}
```

[2] Вообще, это очень полезно понимать. Часто помогает в задачах с модулем.

Тогда чтобы найти первообразный корень по модулю M, необходимо просто перебирать числа начиная с единицы и проверять, являются ли они первообразными корнями.

Алгоритм будет работать быстро, доказывать это особо не умеют. В предположении гипотезы Римана известно, что существует первообразный корень порядка $\log^6 M$. Если вдруг вы не найдете быстро первообразный корень, то вы тем самым опровергните гипотезу Римана, что будет тоже весьма неплохо :)

Особо полезен частный случай, когда m — простое число. Тогда $\varphi(m) = m - 1$.

Также вам может понадобиться примитивный корень из 1 степени 2^k по модулю p, то есть такое число g, что его показатель равен 2^k. В частности, такое число вам понадобится, если вы будете писать FFT по модулю. Здесь алгоритм похожий, но только легче. Вам нужно найти первое такое число, что $g^{2^k} \equiv 1 \pmod p$, но при этом $g^{2^{k-1}} \not\equiv 1 \pmod p$.

Если найти первообразный корень, а потом для нужных остатков найти дискретный логарифм [3] по первообразному корню, то это может облегчить вашу жизнь. Теперь остатки по модулю ведут себя понятным образом: $g^a \cdot g^b \equiv g^{(a+b)\%\varphi(m)}$.

5 Преобразование Уолша-Адамара и xor-and-or-свертки

Быстрое преобразование Фурье сворачивает два массива, отправляя $a[i], b[j] \to c[i+j]$, что соответствует тому, что $x^i \cdot x^j = x^{i+j}$.

Сейчас мы рассмотрим случаи, когда $x^i \cdot x^j = x^{i \oplus j}$, $x^i \cdot x^j = x^{i|j}$ и $x^i \cdot x^j = x^{i \& j}$.

Преобразование Фурье выглядело примерно таким образом: была матрица

$$M = \begin{pmatrix} \omega_0^0 & \omega_0^1 & \omega_0^2 & \dots & \omega_0^{n-1} \\ \omega_1^0 & \omega_1^1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{n-1}^0 & \omega_{n-1}^1 & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{pmatrix}$$

Были векторы A и B. Мы применяли матрицу M к A и B, после чего перемножали MA и MB, а затем применяли M^{-1} к $MA \cdot MB$.

5.1 xor-свертка

Здесь мы хотим сделать ровно то же самое, надо лишь поменять матрицу.

Давайте зададим матрицы H_{2^k} рекурсивно. $H_1 = \begin{pmatrix} 1 \end{pmatrix}$ и $H_{2^k} = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{pmatrix}$

Корень из двух — это не очень удобное число, давайте попытаемся от него избавиться. Определим матрицы без этого корня: $G_1 = \begin{pmatrix} 1 \end{pmatrix}$ и $G_{2^k} = \begin{pmatrix} G_{2^{k-1}} & G_{2^{k-1}} \\ G_{2^{k-1}} & -G_{2^{k-1}} \end{pmatrix}$

Тогда легко заметить, что $\sqrt{n} \cdot H_n = G_n$. Мы будем считать именно G_n.

Обозначение. E_n — это матрица тождественного преобразования размера $n \times n$, то есть матрица, у которой на главной диагонали стоят единицы, а в остальных местах нули.

Из-за такого рекурсивного задания сразу видно, как написать быстрое преобразование Уолша-Адамара, то есть как посчитать $G_{2^k} \cdot A$. На нижнем уровне рекурсии не нужно менять ничего, потому что $G_1 = E_1$, а на других уровнях надо сначала запуститься от двух половин, а потом сделать замену $x, y \to x + y, x - y$. Напишем код:

```
void adamar(vector<int> &a, int l, int r)
{
    if (l + 1 == r)
        return;
    int mid = (r + l) / 2;
    adamar(a, l, mid);
    adamar(a, mid, r);
    for (int i = l; i < mid; i++)
    {
        int u = a[i], v = a[mid + (i - l)];
        a[i] = u + v;
        a[mid + (i - l)] = u - v;
    }
}
```

[3] Если вы не знаете, что это такое, нажмите сюда

Лемма. *Удивительным образом получается, что* $H_{2^k}^{-1} = H_{2^k}$.

Доказательство. Докажем по индукции. Для H_1 это очевидно, потому что $H_1 = E_1$. Докажем для H_{2^k}, если уже известно для $H_{2^{k-1}}$. $H_{2^k} \cdot H_{2^k} = \frac{1}{\sqrt{2}}\begin{pmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{pmatrix} \cdot \frac{1}{\sqrt{2}}\begin{pmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 2H_{2^{k-1}}^2 & 0 \\ 0 & 2H_{2^{k-1}}^2 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 2E_{2^{k-1}} & 0 \\ 0 & 2E_{2^{k-1}} \end{pmatrix} = E_{2^k}$ □

Мы получили, что $H_n \cdot H_n = E_n$. Но мы знаем, что $H_n = \frac{G_n}{\sqrt{n}}$, поэтому $G_n \cdot G_n = n \cdot E_n$. Так что вместо того, чтобы домножать на H, можно домножать на G, но в конце просто поделить на n. Таким образом мы научились делать xor-умножение многочленов без использования вещественных или комплексных чисел:

```
vector<int> xormult(vector<int> a, vector<int> b) // считаем, что a.size() == b.size() == 2^k
{
    adamar(a, 0, (int)a.size());
    adamar(b, 0, (int)b.size());
    for (size_t i = 0; i < a.size(); i++)
        a[i] *= b[i];
    adamar(a, 0, a.size()); // как мы уже обсуждали, обратное преобразование совпадает с прямым
    for (size_t i = 0; i < a.size(); i++)
        a[i] /= (int)a.size();
    return a;
}
```

Если вы внимательно следили за происходящим, то могли заметить, что я вас немного обманул. Я доказал кучу всего, но никак не объяснил, почему все это даст нам xor-свертку.

Обозначим xor-свертку за \$, то есть $(a_0, a_1, \ldots, a_n)\$(b_0, b_1, \ldots, b_n) = (\sum\limits_{k=0}^{n} a_k \cdot b_{0 \oplus k}, \sum\limits_{k=0}^{n} a_k \cdot b_{1 \oplus k}, \ldots, \sum\limits_{k=0}^{n} a_k \cdot b_{n \oplus k})$.

Тогда нам необходимо доказать, что $(Ga) \cdot (Gb) = G(a\$b)$. Для G_1 это очевидно, потому что свертка — это просто умножение чисел. Для G_2 необходимо написать условия: $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \$ \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} a_0 \cdot b_0 + a_1 \cdot b_1 \\ a_0 \cdot b_1 + a_1 \cdot b_0 \end{pmatrix}$, $G_2 a = \begin{pmatrix} a_0 + a_1 \\ a_0 - a_1 \end{pmatrix}$, $G_2 b = \begin{pmatrix} b_0 + b_1 \\ b_0 - b_1 \end{pmatrix}$, $(G_2 a) \cdot (G_2 b) = \begin{pmatrix} (a_0 + a_1) \cdot (b_0 + b_1) \\ (a_0 - a_1) \cdot (b_0 - b_1) \end{pmatrix} = G(a\$b)$.

Для больших размерностей все доказывается по индукции, это не очень интересно. Таким образом можно было бы изначально искать матрицу G, предположив, что $G_2 = \begin{pmatrix} c_0 & c_1 \\ c_2 & c_3 \end{pmatrix}$, решить систему уравнений.

5.2 and-свертка

and-свертка делается аналогично, надо только поменять матрицу. $T_1 = (1)$, $T_{2^k} = \begin{pmatrix} T_{2^{k-1}} & T_{2^{k-1}} \\ 0 & T_{2^{k-1}} \end{pmatrix}$.

К сожалению, в отличие от xor-свертки, здесь обратная матрица не совпадает с прямой, но зато нет корней из двойки, поэтому не придется в конце делить на n.

Обратные матрицы: $T_1^{-1} = (1)$, $T_{2^k}^{-1} = \begin{pmatrix} T_{2^{k-1}}^{-1} & -T_{2^{k-1}}^{-1} \\ 0 & T_{2^{k-1}}^{-1} \end{pmatrix}$.

Из-за рекурентного задания матриц, мы опять же легко можем написать алгоритм:

```
void and_fold(vector<int> &a, int l, int r)
{
    if (l + 1 == r)
        return;
    int mid = (r + l) / 2;
    and_fold(a, l, mid);
    and_fold(a, mid, r);
    for (int i = l; i < mid; i++)
        a[i] += a[mid + (i - l)];
}
```

```
void rev_and_fold(vector<int> &a, int l, int r)
{
    if (l + 1 == r)
        return;
    int mid = (r + l) / 2;
    rev_and_fold(a, l, mid);
    rev_and_fold(a, mid, r);
    for (int i = l; i < mid; i++)
        a[i] -= a[mid + (i - l)];
}

vector<int> andmult(vector<int> a, vector<int> b)
{
    and_fold(a, 0, a.size());
    and_fold(b, 0, b.size());
    for (size_t i = 0; i < a.size(); i++)
        a[i] *= b[i];
    rev_and_fold(a, 0, a.size());
    return a;
}
```

Проверка правильности выбранной матрицы и ее обратной остается читателю в качестве упражнения.

5.3 or-свертка

Все аналогично.

$Q_1 = (1)$, $Q_{2^k} = \begin{pmatrix} Q_{2^{k-1}} & 0 \\ Q_{2^{K-1}} & Q_{2^{k-1}} \end{pmatrix}$.

Обратные матрицы:

$Q_1^{-1} = (1)$, $Q_{2^k}^{-1} = \begin{pmatrix} Q_{2^{k-1}}^{-1} & 0 \\ -Q_{2^{K-1}}^{-1} & Q_{2^{k-1}}^{-1} \end{pmatrix}$.

Из-за рекурентного задания матриц, мы опять же легко можем написать алгоритм:

```
void or_fold(vector<int> &a, int l, int r)
{
    if (l + 1 == r)
        return;
    int mid = (r + l) / 2;
    or_fold(a, l, mid);
    or_fold(a, mid, r);
    for (int i = l; i < mid; i++)
        a[mid + (i - l)] += a[i];
}

void rev_or_fold(vector<int> &a, int l, int r)
{
    if (l + 1 == r)
        return;
    int mid = (r + l) / 2;
    rev_or_fold(a, l, mid);
    rev_or_fold(a, mid, r);
    for (int i = l; i < mid; i++)
        a[mid + (i - l)] -= a[i];
}

vector<int> ormult(vector<int> a, vector<int> b)
{
    or_fold(a, 0, a.size());
    or_fold(b, 0, b.size());
    for (size_t i = 0; i < a.size(); i++)
```

```
        a[i] *= b[i];
    rev_or_fold(a, 0, a.size());
    return a;
}
```

Проверка правильности выбранной матрицы и ее обратной остается читателю в качестве упражнения.

Замечание. Заметим, что $a|b = \overline{\overline{a}\&\overline{b}}$, где $\overline{x}$ — замена всех единичных битов числа на нулевые, а нулевых на единичные. Поэтому or-свертку можно написать через *and*-свертку (и наоборот).

5.4 Оптимизации

Убрать рекурсию здесь проще, чем в FFT. Рекурсия головная, поэтому раскрывается очевидным образом. Также прямые и обратные отображения очень похожи, так что их можно объединить в одну функцию.

```
void adamar(vector<int> &a)
{
    for (int len = 1; 2 * len <= (int)a.size(); len <<= 1)
        for (int i = 0; i < (int)a.size(); i += 2 * len)
            for (int j = 0; j < len; j++)
            {
                int u = a[i + j], v = a[i + len + j];
                a[i + j] = u + v;
                a[i + len + j] = u - v;
            }
    // по желанию можно добавить деление на n сюда, тогда надо будет тоже передавать флаг inv
}


void and_fold(vector<int>& a, bool inv)
{
    for (int len = 1; 2 * len <= (int)a.size(); len <<= 1)
        for (int i = 0; i < (int)a.size(); i += 2 * len)
            for (int j = 0; j < len; j++)
            {
                if (inv)
                    a[i + j] -= a[i + len + j];
                else
                    a[i + j] += a[i + len + j];
            }
}

void or_fold(vector<int>& a, bool inv)
{
    for (int len = 1; 2 * len <= (int)a.size(); len <<= 1)
        for (int i = 0; i < (int)a.size(); i += 2 * len)
            for (int j = 0; j < len; j++)
            {
                if (inv)
                    a[i + len + j] -= a[i + j];
                else
                    a[i + len + j] += a[i + j];
            }
}
```