

## Задача А. Хомячья раскраска

Пусть дана доска размера  $n \times m$ , которую требуется раскрасить диагонально в  $k$  цветов. Цвет клетки  $(i, j)$  определяется формулой

$$\text{color}(i, j) = ((i + j - 2) \bmod k) + 1,$$

то есть каждая диагональ с одинаковой суммой  $i + j$  получает один цвет, а цвета циклически повторяются.

### 1. Остатки по модулю $k$

Рассмотрим остатки

$$r = (i - 1) \bmod k, \quad s = (j - 1) \bmod k.$$

Тогда

$$(i + j - 2) \bmod k = (r + s) \bmod k.$$

Следовательно, цвет клетки зависит только от пары остатков  $(r, s)$ .

### 2. Сколько строк и столбцов имеют каждый остаток

Для чисел  $1, \dots, n$  остатки  $(i - 1) \bmod k$  распределяются равномерно.

Пусть

$$n = q_n \cdot k + \text{rem}_n, \quad 0 \leq \text{rem}_n < k.$$

Тогда количество строк, чьи индексы имеют остаток  $r$ , равно

$$\text{rows}[r] = \begin{cases} q_n + 1, & r < \text{rem}_n, \\ q_n, & r \geq \text{rem}_n. \end{cases}$$

Аналогично для столбцов:

$$\begin{aligned} m &= q_m \cdot k + \text{rem}_m, \\ \text{cols}[s] &= \begin{cases} q_m + 1, & s < \text{rem}_m, \\ q_m, & s \geq \text{rem}_m. \end{cases} \end{aligned}$$

### 3. Подсчёт количества клеток каждого цвета

Все клетки с остатками  $(r, s)$  дают вклад

$$\text{rows}[r] \cdot \text{cols}[s]$$

в цвет номер

$$t = (r + s) \bmod k.$$

Таким образом:

$$\text{ans}[t] += \text{rows}[r] \cdot \text{cols}[s].$$

После этого массив  $\text{ans}$  содержит:

$$\text{ans}[0] = \text{клеток цвета 1}, \quad \text{ans}[1] = \text{клеток цвета 2}, \dots$$

### 4. Временная сложность

Так как  $k \leq 5$ , перебор всех пар остатков  $(r, s)$  занимает не более 25 итераций — решение работает за  $O(k^2)$ .

Типы данных должны быть 64-битными, поскольку результат может достигать  $4 \cdot 10^{18}$ .

## 5. Полный код решения на Python

```
1 import sys
2
3 data = sys.stdin.read().strip().split()
4 n = int(data[0])
5 m = int(data[1])
6 k = int(data[2])
7
8 rows = [0] * k
9 qn, remn = divmod(n, k)
10 for r in range(k):
11     rows[r] = qn + (1 if r < remn else 0)
12
13 cols = [0] * k
14 qm, remm = divmod(m, k)
15 for s in range(k):
16     cols[s] = qm + (1 if s < remm else 0)
17
18 ans = [0] * k
19 for r in range(k):
20     for s in range(k):
21         t = (r + s) % k
22         ans[t] += rows[r] * cols[s]
23
24 for x in ans:
25     print(x)
```

## Задача В. Магическая семёрка

### Подзадача 1. [ПЕРЕБОРНОЕ РЕШЕНИЕ.]

Будем последовательно проверять числа начиная с 1, пока не будет найдено  $k$ -е подходящее число. Счётчик `count` отслеживает количество найденных магических чисел. Переменная  $n$  соответствует текущему натуральному числу. Условие  $n \% 7 == 0$  or  $n \% 10 == 7$  проверяет, является ли число  $n$  магическим. Цикл продолжается до тех пор, пока не будет найдено  $k$ -е магическое число.

Это решение работает для  $n \leq 10^6$  и имеет сложность  $O(n)$ .

```
1 k = int(input())
2 count = 0
3 n = 0
4 while count < k:
5     n += 1
6     if n % 7 == 0 or n % 10 == 7:
7         count += 1
8 print(n)
```

### Подзадача 2. [ФОРМУЛА ВКЛЮЧЕНИЙ-ИСКЛЮЧЕНИЙ.]

В этом решении применяются комбинаторные рассуждения и не используется перебор.

```
1 def count_magic_numbers(n):
2     return n // 7 + (n + 3) // 10 - (n + 63) // 70
3
4 def find_kth_magic_number(k):
5     left, right = 1, 10**18
6
7     while left < right:
8         mid = (left + right) // 2
9         if count_magic_numbers(mid) < k:
10             left = mid + 1
```

```
11     else:
12         right = mid
13     return left
14
15 k = int(input())
16
17 result = find_kth_magic_number(k)
18 print(result)
```

Функция `count_magic_numbers(n)` вычисляет количество магических чисел от 1 до  $n$  по формуле включений-исключений. Сначала подсчитываем количество чисел, кратных 7. Они образуют арифметическую прогрессию, их количество равно  $n // 7$ . Затем учитываем числа, оканчивающиеся на 7; они имеют вид  $10k + 7$ , их количество равно  $(n + 3) // 10$ . Наконец, подсчитаем количество чисел с обоими свойствами; эти числа имеют вид  $70m + 7$ , их количество равно  $(n + 63) // 70$ . Для того, чтобы избежать двойного учёта, вычисляем сумму первых двух величин за вычетом третьей.

Функция `find_kth_magic_number(k)` использует бинарный поиск для нахождения  $k$ -го магического числа. На каждой итерации эта функция проверяет, сколько магических чисел содержится в диапазоне  $[1, mid]$ . Если количество меньше  $k$ , сдвигает левую границу, иначе — правую. По завершении цикла `left` указывает на  $k$ -е магическое число.

Сложность этого решения  $O(\log(\max\_answer))$  — бинарный поиск по числам до  $10^{17}$ , память  $O(1)$  — используется константное количество памяти.

Это решение эффективно работает для больших значений  $k$  (до  $10^{17}$ ), так как использует только целочисленную арифметику и бинарный поиск.

## Задача С. Нечётно-степенные числа

ПЕРВАЯ ПОДЗАДАЧА. Для небольших значений  $m$  и  $r$  задачу можно решить перебором. Будем проверять делимость конкретного числа  $n$  из промежутка  $[l; r]$  на все числа  $i$ , начиная с  $i = 2$  и до  $i = n - 1$ . Если при очередной проверке  $n$  не делится на  $i$ , пропускаем это значение  $i$ , а если делится, проверяем делимость на  $i$  ещё раз и так далее. Если  $n$  делится на  $i$  нечётное число раз, значит, этот множитель  $i$  входит в разложение  $n$  в нечётной степени. (Это число  $i$  будет также простым числом!) Затем переходим к следующему значению  $i$ . Таким образом, мы сможем установить, будет ли число  $n$  нечётно-степенным. Если количество нечётно-степенных чисел  $n$  в промежутке  $[l; r]$  равно заданному  $m$ , получаем требуемый ответ.

ВТОРАЯ ПОДЗАДАЧА: Для значений  $m \leq 10$  предыдущее решение можно улучшить. Дело в том, что если число  $n$  — составное, его наименьший простой делитель не превосходит  $\sqrt{n}$ , поэтому его простые делители можно найти за  $O(\sqrt{n})$  операций. Для этого проверяем делимость конкретного числа  $n$  из промежутка  $[l; r]$  на все числа  $i$ , не превосходящие  $\sqrt{n}$ . Такое решение проходит все тесты второй группы.

ТРЕТЬЯ ПОДЗАДАЧА. Улучшим решение для значений  $m > 10$ . Для этого нужно заметить, что если требуется найти последовательность из  $m \geq 8$  подряд идущих нечётно-степенных чисел, ответ в задаче отрицательный, так как таких наборов нет. Действительно, восемь последовательных натуральных чисел дают различные остатки при делении на 8. Поэтому среди них найдётся число, которое при делении на 8 дает остаток 4. Это число делится на 4, но не делится на 8, поэтому в разложении его на простые множители двойка входит во второй степени. Другими словами, при  $m > 7$  ответ в задаче -1, то есть не существует 8 последовательных нечётно-степенных чисел.

## Задача Д. Хомяк и двоичные тайны

ПЕРВАЯ ПОДЗАДАЧА. При  $R \leq 2^{17}$  достаточно просто перебрать все числа  $x$  от 0 до  $R$ . Для каждого числа можно вручную получить двоичную запись делением на два: на каждом шаге вычисляем остаток  $x \bmod 2$  и делим  $x$  на 2. Такое разложение занимает  $O(\log x)$  времени. Подсчитав количество единиц в двоичном представлении числа  $x$ , проверяем условие  $\text{popcount}(x) \equiv 0 \pmod{K}$ . Общее время работы —  $O(R \log R)$ , что полностью укладывается в ограничения первой подзадачи.

ВТОРАЯ ПОДЗАДАЧА. При  $R \leq 2^{25}$  количество чисел уже достигает  $3 \cdot 10^7$ , и ручное деление на 2 для каждого числа становится заметно медленнее. Однако современные языки программирования предоставляют готовые функции подсчёта количества установленных битов — `popcount`. Например:

- C++: `__builtin_popcount`, `__builtin_popcountll`;
- Java: `Integer.bitCount()`, `Long.bitCount()`;
- Python: метод `x.bit_count()`;
- C#: `System.Numerics.BitOperations.PopCount()`.

Эти функции реализованы на уровне процессора и работают за  $O(1)$  на большинстве платформ. Поэтому решение сводится к простому перебору  $x$  от 0 до  $R$ , вызову `popcount` и проверке делимости. Такое решение работает за  $O(R)$  и проходит вторую подзадачу.

ТРЕТЬЯ ПОДЗАДАЧА. В этой группе  $R \leq 2^{44}$ , и полный перебор  $x$  невозможен. Однако двоичная длина числа  $R$  не превосходит 44, и можно применить идею meet in the middle.

Пусть длина двоичной записи  $R$  равна  $L \leq 44$ . Разобьём её на две половины: младшие  $m$  бит и старшие  $L - m$  бит. Запишем число  $x$  как

$$x = \text{hi} \cdot 2^m + \text{lo}.$$

Тогда

$$\text{popcount}(x) = \text{popcount}(\text{hi}) + \text{popcount}(\text{lo}).$$

Далее:

- переберём все значения  $\text{lo}$  (их  $2^m$ ), сгруппируем по количеству единиц;
- переберём все значения  $\text{hi}$ , для каждого найдём максимально возможное  $\text{lo}_{\max}$ , такое что  $x \leq R$ ;
- по количеству единиц в  $\text{hi}$  определим требуемое количество единиц в  $\text{lo}$ , и найдём количество подходящих  $\text{lo}$  бинарным поиском.

Такое решение имеет асимптотику около  $O(2^{L/2} \cdot L)$ , что хорошо работает при  $L \leq 44$ .

ЧЕТВЁРТАЯ ПОДЗАДАЧА. При  $R \leq 2^{63}$  двоичная длина не превосходит 63. Здесь оптимально использовать стандартное битовое ДП по двоичной записи  $R$ .

Пусть  $b_0 b_1 \dots b_{n-1}$  — двоичная запись  $R$  (старший бит слева). Рассмотрим ДП

$$dp[i][r][t],$$

где:

- $i$  — обработано первых  $i$  битов;
- $r$  — остаток по модулю  $K$  от числа единиц;
- $t \in \{0, 1\}$  — флаг `tight`, равный 1, если префикс числа совпадает с префиксом  $R$ .

Переходы стандартны для digit-DP: если  $t = 1$ , то на текущем бите можно выбрать только биты  $\leq b_i$ , если  $t = 0$ , то можно выбирать 0 и 1 свободно. Количество единиц обновляется как  $(r + \text{bit}) \bmod K$ .

После обработки всех  $n$  битов искомый ответ равен

$$dp[n][0][0] + dp[n][0][1].$$

Сложность такого решения —  $O(nK)$ , что проходит четвертую группу.

ПЯТАЯ ПОДЗАДАЧА. В этой группе  $R \leq 2^{127}$ , и обычных 64 бит уже недостаточно. В C++ существует тип `__int128`, который позволяет хранить такие числа. Тогда можно:

- прочитать  $R$  как десятичную строку;
- перевести её в тип `_int128`;
- извлечь двоичную запись с помощью сдвигов и `&1`;
- выполнить битовое ДП, как в четвёртой подзадаче.

В других языках прямого аналога `_int128` нет:

- в **Java** есть класс `BigInteger` с методами `shiftRight`, `testBit`;
- в **Python** тип `int` поддерживает произвольно длинные числа, и можно использовать операции `»` и `& 1`. Однако у современных версий Python есть ограничение на максимальное число цифр строки, которое можно преобразовать в `int`. По умолчанию разрешено не более 4300 цифр; это параметр `sys.int_info.default_max_str_digits` (его можно увеличивать вызовом `sys.set_int_max_str_digits()`).
- в **C#** доступен класс `System.Numerics.BigInteger`.

Но в любом случае двоичная длина здесь не превышает 127 бит, поэтому битовое ДП работает очень быстро.

**ШЕСТАЯ ПОДЗАДАЧА.** Теперь  $n \leq 200$  десятичных цифр, то есть двоичная длина числа  $R$  порядка 664 бит.  $K \leq 200$ , поэтому можно использовать битовое ДП, но требуется эффективно перевести  $R$  в двоичную систему.

Проще всего выполнить деление большой десятичной строки на 2: проходим по всем её символам слева направо, получаем частное и остаток. Остаток образует очередной бит результата. Повторяем до тех пор, пока строка не станет «0».

Время работы такой конвертации:  $O(n \log R)$ , что достаточно быстро при  $n \leq 200$ . Затем выполняется стандартное ДП по битам.

**СЕДЬМАЯ ПОДЗАДАЧА.** При  $n \leq 2000$  двоичная длина достигает  $\sim 6644$  бит, и хранить трёхмерное ДП  $dp[i][r][t]$  уже слишком тяжело: память  $\Theta(nK)$  может не поместиться.

Но для digit-DP по битам достаточно хранить только два слоя: текущий и следующий:

$$dp[r][t], \quad ndp[r][t].$$

После обработки каждого бита слои меняются местами, и  $ndp$  обнуляется. Это стандартная оптимизация «rolling array», и она позволяет уложиться как по времени, так и по памяти.

**ВОСЬМАЯ ПОДЗАДАЧА.** При  $n \leq 5000$  конвертация десятичной строки в двоичное число с делением по одной цифре становится слишком медленной. Нужно ускорять операции над большими числами.

Строка  $R$  разбивается на блоки по 18 цифр (основание  $10^{18}$ ). Так получается массив целых 64-битных чисел:

$$R = a_0 \cdot 10^{18k} + a_1 \cdot 10^{18(k-1)} + \dots + a_k.$$

Деление большого числа на 2 выполняется уже по блокам: каждый блок делится как обычное 64-битное число, а остаток переносится в следующий блок. Количество блоков уменьшается примерно в 18 раз, и перевод становится существенно быстрее.

После получения двоичной записи выполняется ДП или комбинаторика (см. следующую подзадачу).

**ДЕВЯТАЯ ПОДЗАДАЧА.** При  $n \leq 30000$  и  $K \leq 100000$  переходы  $O(nK)$  становятся слишком медленными. Здесь используется комбинаторный подход и корневая декомпозиция относительно параметра  $K$ .

Заметим, что существует естественный порог когда  $K$  большое, проще считать числа с точным количеством единиц.

Пусть длина двоичной записи  $R$  равна  $n$ , и рассмотрим значения  $s$ , кратные  $K$ :

$$s = 0, K, 2K, \dots, \quad s \leq n.$$

Для каждого такого  $s$  можно подсчитать количество чисел  $x \leq R$  с  $\text{popcount}(x) = s$  комбинаторно.

Рассмотрим двоичную запись  $R = b_0b_1\dots b_{n-1}$ . Будем проходить её слева направо и следить, сколько единиц содержится в префиксе  $R[0..i-1]$ . Когда на позиции  $i$  стоит единица, у нас появляется вариант поставить в числа  $x$  на этой позиции ноль («уйти вниз» относительно  $R$ ). После этого хвост из  $(n-i-1)$  бит можно выбирать произвольно, но нужно поставить ровно

$$\text{need} = s - \text{cnt\_ones}$$

единиц. Количество способов равно биномиальному коэффициенту

$$\binom{n-i-1}{\text{need}}.$$

Суммируя такие варианты для всех позиций, где  $R$  содержит единицу, а также учитывая само число  $R$  (если  $\text{popcount}(R) = s$ ), получаем число всех  $x \leq R$  с ровно  $s$  единицами.

Так как  $K$  большое, количество значений  $s$  имеет порядок  $\frac{n}{K}$ . Каждое вычисление занимает  $O(n)$ , всего сложность порядка

$$O\left(\frac{n^2}{K}\right),$$

что эффективно при большом  $K$ .

Именно поэтому в финальном решении применяется корневая декомпозиция:

$$\text{если } K^2 \leq n: \quad \text{выгодно ДП}; \quad \text{если } K^2 > n: \quad \text{выгодна комбинаторика.}$$

Такой гибрид полностью покрывает самую жёсткую девятую подзадачу.

Почему вообще гибрид «ДП + комбинаторика» является оптимальным алгоритмом?

- При **маленьком**  $K$  выгодно использовать битовое ДП. Его сложность:  $O(nK)$ .
- При **большом**  $K$  выгодно использовать комбинаторный подход, так как количество подходящих значений  $s$  (таких что  $s \equiv 0 \pmod{K}$  и  $s \leq n$ ) равно  $\frac{n}{K}$ , поэтому общая сложность:  $O\left(\frac{n^2}{K}\right)$ .

Чтобы понять, какой метод быстрее, рассмотрим границу между ними. Ищем значение  $K$ , при котором оба варианта дают одинаковую сложность:

$$nK \approx \frac{n^2}{K} \implies K^2 \approx n \implies K \approx \sqrt{n}.$$

Тогда:

- Если  $K^2 \leq n$  (то есть  $K \leq \sqrt{n}$ ), то

$$nK \leq n\sqrt{n}, \quad \frac{n^2}{K} \geq \frac{n^2}{\sqrt{n}} = n^{3/2},$$

значит **ДП быстрее комбинаторики**.

- Если  $K^2 > n$  (то есть  $K > \sqrt{n}$ ), то

$$\frac{n^2}{K} \leq \frac{n^2}{\sqrt{n}} = n^{3/2}, \quad nK \geq n\sqrt{n},$$

значит **комбинаторика быстрее ДП**.

Таким образом, порог  $K \approx \sqrt{n}$  является разделителем между двумя методами. Именно поэтому итоговое решение использует правило:

если  $K^2 \leq n$ , применяем DP;      иначе используем комбинаторику.

Итоговая сложность:  $O(n \cdot \sqrt{n})$

## Задача Е. Сказ о Луче Великой Справедливости

**ПЕРВАЯ ПОДЗАДАЧА.** При очень малом количестве комнат ( $n \leq 2$ ) ответ всегда **Yes**. Действительно, любую одну или две осево-ориентированные комнаты всегда можно пересечь одной прямой: при необходимости достаточно провести прямую через одну из сторон или углов. Поэтому эта подзадача решается без вычислений.

**ВТОРАЯ ПОДЗАДАЧА.** Здесь требуется рассматривать только горизонтальные и вертикальные прямые. Горизонтальная прямая  $y = y_0$  пересекает комнату  $[x_1, x_2] \times [y_1, y_2]$  тогда и только тогда, когда  $y_1 \leq y_0 \leq y_2$ . Поэтому существование одной горизонтальной прямой, проходящей через все комнаты, эквивалентно непустоте пересечения всех отрезков  $[y_1, y_2]$ . Аналогично вертикальная прямая  $x = x_0$  пересекает все комнаты тогда и только тогда, когда пересечение всех отрезков  $[x_1, x_2]$  непусто. Если хотя бы одно из двух пересечений непусто, ответ — **Yes**. Это решение работает за  $O(n)$  и полностью покрывает вторую подзадачу.

**ТРЕТЬЯ ПОДЗАДАЧА.** Теперь, если ответ **Yes**, достаточно существования прямой под углом  $45^\circ$  или  $135^\circ$ . Для таких прямых удобно ввести новые координаты:

$$u = x + y, \quad v = x - y.$$

Тогда прямая под углом  $45^\circ$  фиксируется параметром  $u$ , а под углом  $135^\circ$  — параметром  $v$ . Каждая комната задаёт интервал возможных значений  $u$  (или  $v$ ), достижимых в её углах. Поэтому задача сводится к проверке непустоты пересечения всех отрезков по  $u$  и затем по  $v$ . Если хотя бы одно из двух пересечений непусто, ответ — **Yes**. Это решение также линейно по  $n$  и покрывает третью подзадачу.

**ЧЕТВЁРТАЯ ПОДЗАДАЧА.** При  $n \leq 200$  и ограничении  $\sum n^3 \leq 3 \cdot 10^6$  можно позволить себе перебор всех «кандидатных» направлений прямой. В предельном положении подходящая прямая обязательно касается одного или двух углов некоторых прямоугольников. Поэтому достаточно перебрать все пары углов (всего  $O(n^2)$  направлений), построить соответствующую прямую и проверить за  $O(n)$ , пересекает ли она все комнаты. Итоговая асимптотика  $O(n^3)$  проходит условия четвёртой подзадачи.

**ПЯТАЯ И ШЕСТАЯ ПОДЗАДАЧИ.** При больших  $n$  полный перебор направлений уже слишком медленный, но можно воспользоваться геометрической структурой задачи.

Рассмотрим прямые положительного наклона. В предельном положении такая прямая проходит:

- ниже всех верхних левых углов комнат,
- выше всех нижних правых углов.

Пусть  $A$  — множество верхних левых углов, а  $B$  — множество нижних правых. Построим выпуклые оболочки этих множеств:  $A'$  (верхняя оболочка) и  $B'$  (нижняя оболочка).

Тогда для существования прямой положительного наклона, проходящей между всеми комнатами, необходимо и достаточно, чтобы оболочки  $A'$  и  $B'$  не пересекались. Интуитивно это отражает тот факт, что для любого наклона  $k$  верхняя граница допустимого положения прямой определяется оболочкой  $A'$ , а нижняя — оболочкой  $B'$ , и для существования прямой нужно, чтобы нижняя граница не поднималась выше верхней.

Проверить пересечение двух выпуклых ломаных можно за  $O(n^2)$ , что укладывается в ограничения пятой и шестой подзадач. Аналогичное рассуждение проводится для прямых отрицательного наклона (используются остальные два противоположных угла).

СЕДЬМАЯ ПОДЗАДАЧА. В полном решении требуется асимптотика  $O(n \log n)$  при  $\sum n \leq 4 \cdot 10^5$ . Здесь используется та же геометрическая идея, но реализованная оптимально.

Как и ранее, рассматриваем три типа прямых: вертикальные, прямые с положительным наклоном и прямые с отрицательным наклоном.

- Для вертикальных прямых достаточно проверить непустоту пересечения всех отрезков  $[x_1, x_2]$ .
- Строительство выпуклых оболочек занимает  $O(n \log n)$ , а проверка их пересечения — линейное время (например, с использованием суммы Минковского или двухуказательного обхода). Аналогично для прямых отрицательного наклона.

Таким образом, итоговая асимптотика составляет  $O(n \log n)$ , что позволяет получить полный балл.

## Задача А. Хомячья раскраска

Пусть дана доска размера  $n \times m$ , которую требуется раскрасить диагонально в  $k$  цветов. Цвет клетки  $(i, j)$  определяется формулой

$$\text{color}(i, j) = ((i + j - 2) \bmod k) + 1,$$

то есть каждая диагональ с одинаковой суммой  $i + j$  получает один цвет, а цвета циклически повторяются.

### 1. Остатки по модулю $k$

Рассмотрим остатки

$$r = (i - 1) \bmod k, \quad s = (j - 1) \bmod k.$$

Тогда

$$(i + j - 2) \bmod k = (r + s) \bmod k.$$

Следовательно, цвет клетки зависит только от пары остатков  $(r, s)$ .

### 2. Сколько строк и столбцов имеют каждый остаток

Для чисел  $1, \dots, n$  остатки  $(i - 1) \bmod k$  распределяются равномерно.

Пусть

$$n = q_n \cdot k + \text{rem}_n, \quad 0 \leq \text{rem}_n < k.$$

Тогда количество строк, чьи индексы имеют остаток  $r$ , равно

$$\text{rows}[r] = \begin{cases} q_n + 1, & r < \text{rem}_n, \\ q_n, & r \geq \text{rem}_n. \end{cases}$$

Аналогично для столбцов:

$$\begin{aligned} m &= q_m \cdot k + \text{rem}_m, \\ \text{cols}[s] &= \begin{cases} q_m + 1, & s < \text{rem}_m, \\ q_m, & s \geq \text{rem}_m. \end{cases} \end{aligned}$$

### 3. Подсчёт количества клеток каждого цвета

Все клетки с остатками  $(r, s)$  дают вклад

$$\text{rows}[r] \cdot \text{cols}[s]$$

в цвет номер

$$t = (r + s) \bmod k.$$

Таким образом:

$$\text{ans}[t] += \text{rows}[r] \cdot \text{cols}[s].$$

После этого массив  $\text{ans}$  содержит:

$$\text{ans}[0] = \text{клеток цвета 1}, \quad \text{ans}[1] = \text{клеток цвета 2}, \dots$$

### 4. Временная сложность

Так как  $k \leq 5$ , перебор всех пар остатков  $(r, s)$  занимает не более 25 итераций — решение работает за  $O(k^2)$ .

Типы данных должны быть 64-битными, поскольку результат может достигать  $4 \cdot 10^{18}$ .

## 5. Полный код решения на Python

```
1 import sys
2
3 data = sys.stdin.read().strip().split()
4 n = int(data[0])
5 m = int(data[1])
6 k = int(data[2])
7
8 rows = [0] * k
9 qn, remn = divmod(n, k)
10 for r in range(k):
11     rows[r] = qn + (1 if r < remn else 0)
12
13 cols = [0] * k
14 qm, remm = divmod(m, k)
15 for s in range(k):
16     cols[s] = qm + (1 if s < remm else 0)
17
18 ans = [0] * k
19 for r in range(k):
20     for s in range(k):
21         t = (r + s) % k
22         ans[t] += rows[r] * cols[s]
23
24 for x in ans:
25     print(x)
```

## Задача В. Магическая семёрка

### Подзадача 1. [ПЕРЕБОРНОЕ РЕШЕНИЕ.]

Будем последовательно проверять числа начиная с 1, пока не будет найдено  $k$ -е подходящее число. Счётчик `count` отслеживает количество найденных магических чисел. Переменная  $n$  соответствует текущему натуральному числу. Условие  $n \% 7 == 0$  or  $n \% 10 == 7$  проверяет, является ли число  $n$  магическим. Цикл продолжается до тех пор, пока не будет найдено  $k$ -е магическое число.

Это решение работает для  $n \leq 10^6$  и имеет сложность  $O(n)$ .

```
1 k = int(input())
2 count = 0
3 n = 0
4 while count < k:
5     n += 1
6     if n % 7 == 0 or n % 10 == 7:
7         count += 1
8 print(n)
```

### Подзадача 2. [ФОРМУЛА ВКЛЮЧЕНИЙ-ИСКЛЮЧЕНИЙ.]

В этом решении применяются комбинаторные рассуждения и не используется перебор.

```
1 def count_magic_numbers(n):
2     return n // 7 + (n + 3) // 10 - (n + 63) // 70
3
4 def find_kth_magic_number(k):
5     left, right = 1, 10**18
6
7     while left < right:
8         mid = (left + right) // 2
9         if count_magic_numbers(mid) < k:
10             left = mid + 1
```

```
11     else:
12         right = mid
13     return left
14
15 k = int(input())
16
17 result = find_kth_magic_number(k)
18 print(result)
```

Функция `count_magic_numbers(n)` вычисляет количество магических чисел от 1 до  $n$  по формуле включений-исключений. Сначала подсчитываем количество чисел, кратных 7. Они образуют арифметическую прогрессию, их количество равно  $n // 7$ . Затем учитываем числа, оканчивающиеся на 7; они имеют вид  $10k + 7$ , их количество равно  $(n + 3) // 10$ . Наконец, подсчитаем количество чисел с обоими свойствами; эти числа имеют вид  $70m + 7$ , их количество равно  $(n + 63) // 70$ . Для того, чтобы избежать двойного учёта, вычисляем сумму первых двух величин за вычетом третьей.

Функция `find_kth_magic_number(k)` использует бинарный поиск для нахождения  $k$ -го магического числа. На каждой итерации эта функция проверяет, сколько магических чисел содержится в диапазоне  $[1, mid]$ . Если количество меньше  $k$ , сдвигает левую границу, иначе — правую. По завершении цикла `left` указывает на  $k$ -е магическое число.

Сложность этого решения  $O(\log(\max\_answer))$  — бинарный поиск по числам до  $10^{17}$ , память  $O(1)$  — используется константное количество памяти.

Это решение эффективно работает для больших значений  $k$  (до  $10^{17}$ ), так как использует только целочисленную арифметику и бинарный поиск.

## Задача С. Нечётно-степенные числа

ПЕРВАЯ ПОДЗАДАЧА. Для небольших значений  $m$  и  $r$  задачу можно решить перебором. Будем проверять делимость конкретного числа  $n$  из промежутка  $[l; r]$  на все числа  $i$ , начиная с  $i = 2$  и до  $i = n - 1$ . Если при очередной проверке  $n$  не делится на  $i$ , пропускаем это значение  $i$ , а если делится, проверяем делимость на  $i$  ещё раз и так далее. Если  $n$  делится на  $i$  нечётное число раз, значит, этот множитель  $i$  входит в разложение  $n$  в нечётной степени. (Это число  $i$  будет также простым числом!) Затем переходим к следующему значению  $i$ . Таким образом, мы сможем установить, будет ли число  $n$  нечётно-степенным. Если количество нечётно-степенных чисел  $n$  в промежутке  $[l; r]$  равно заданному  $m$ , получаем требуемый ответ.

ВТОРАЯ ПОДЗАДАЧА: Для значений  $m \leq 10$  предыдущее решение можно улучшить. Дело в том, что если число  $n$  — составное, его наименьший простой делитель не превосходит  $\sqrt{n}$ , поэтому его простые делители можно найти за  $O(\sqrt{n})$  операций. Для этого проверяем делимость конкретного числа  $n$  из промежутка  $[l; r]$  на все числа  $i$ , не превосходящие  $\sqrt{n}$ . Такое решение проходит все тесты второй группы.

ТРЕТЬЯ ПОДЗАДАЧА. Улучшим решение для значений  $m > 10$ . Для этого нужно заметить, что если требуется найти последовательность из  $m \geq 8$  подряд идущих нечётно-степенных чисел, ответ в задаче отрицательный, так как таких наборов нет. Действительно, восемь последовательных натуральных чисел дают различные остатки при делении на 8. Поэтому среди них найдётся число, которое при делении на 8 дает остаток 4. Это число делится на 4, но не делится на 8, поэтому в разложении его на простые множители двойка входит во второй степени. Другими словами, при  $m > 7$  ответ в задаче -1, то есть не существует 8 последовательных нечётно-степенных чисел.

## Задача Д. Хомяк и двоичные тайны

ПЕРВАЯ ПОДЗАДАЧА. При  $R \leq 2^{17}$  достаточно просто перебрать все числа  $x$  от 0 до  $R$ . Для каждого числа можно вручную получить двоичную запись делением на два: на каждом шаге вычисляем остаток  $x \bmod 2$  и делим  $x$  на 2. Такое разложение занимает  $O(\log x)$  времени. Подсчитав количество единиц в двоичном представлении числа  $x$ , проверяем условие  $\text{popcount}(x) \equiv 0 \pmod{K}$ . Общее время работы —  $O(R \log R)$ , что полностью укладывается в ограничения первой подзадачи.

ВТОРАЯ ПОДЗАДАЧА. При  $R \leq 2^{25}$  количество чисел уже достигает  $3 \cdot 10^7$ , и ручное деление на 2 для каждого числа становится заметно медленнее. Однако современные языки программирования предоставляют готовые функции подсчёта количества установленных битов — `popcount`. Например:

- C++: `__builtin_popcount`, `__builtin_popcountll`;
- Java: `Integer.bitCount()`, `Long.bitCount()`;
- Python: метод `x.bit_count()`;
- C#: `System.Numerics.BitOperations.PopCount()`.

Эти функции реализованы на уровне процессора и работают за  $O(1)$  на большинстве платформ. Поэтому решение сводится к простому перебору  $x$  от 0 до  $R$ , вызову `popcount` и проверке делимости. Такое решение работает за  $O(R)$  и проходит вторую подзадачу.

ТРЕТЬЯ ПОДЗАДАЧА. В этой группе  $R \leq 2^{44}$ , и полный перебор  $x$  невозможен. Однако двоичная длина числа  $R$  не превосходит 44, и можно применить идею meet in the middle.

Пусть длина двоичной записи  $R$  равна  $L \leq 44$ . Разобьём её на две половины: младшие  $m$  бит и старшие  $L - m$  бит. Запишем число  $x$  как

$$x = \text{hi} \cdot 2^m + \text{lo}.$$

Тогда

$$\text{popcount}(x) = \text{popcount}(\text{hi}) + \text{popcount}(\text{lo}).$$

Далее:

- переберём все значения  $\text{lo}$  (их  $2^m$ ), сгруппируем по количеству единиц;
- переберём все значения  $\text{hi}$ , для каждого найдём максимально возможное  $\text{lo}_{\max}$ , такое что  $x \leq R$ ;
- по количеству единиц в  $\text{hi}$  определим требуемое количество единиц в  $\text{lo}$ , и найдём количество подходящих  $\text{lo}$  бинарным поиском.

Такое решение имеет асимптотику около  $O(2^{L/2} \cdot L)$ , что хорошо работает при  $L \leq 44$ .

ЧЕТВЁРТАЯ ПОДЗАДАЧА. При  $R \leq 2^{63}$  двоичная длина не превосходит 63. Здесь оптимально использовать стандартное битовое ДП по двоичной записи  $R$ .

Пусть  $b_0 b_1 \dots b_{n-1}$  — двоичная запись  $R$  (старший бит слева). Рассмотрим ДП

$$dp[i][r][t],$$

где:

- $i$  — обработано первых  $i$  битов;
- $r$  — остаток по модулю  $K$  от числа единиц;
- $t \in \{0, 1\}$  — флаг `tight`, равный 1, если префикс числа совпадает с префиксом  $R$ .

Переходы стандартны для digit-DP: если  $t = 1$ , то на текущем бите можно выбрать только биты  $\leq b_i$ , если  $t = 0$ , то можно выбирать 0 и 1 свободно. Количество единиц обновляется как  $(r + \text{bit}) \bmod K$ .

После обработки всех  $n$  битов искомый ответ равен

$$dp[n][0][0] + dp[n][0][1].$$

Сложность такого решения —  $O(nK)$ , что проходит четвертую группу.

ПЯТАЯ ПОДЗАДАЧА. В этой группе  $R \leq 2^{127}$ , и обычных 64 бит уже недостаточно. В C++ существует тип `__int128`, который позволяет хранить такие числа. Тогда можно:

1. прочитать  $R$  как десятичную строку;
2. перевести её в тип `_int128`;
3. извлечь двоичную запись с помощью сдвигов и `&1`;
4. выполнить битовое ДП, как в четвёртой подзадаче.

В других языках прямого аналога `_int128` нет:

- в **Java** есть класс `BigInteger` с методами `shiftRight`, `testBit`;
- в **Python** тип `int` поддерживает произвольно длинные числа, и можно использовать операции `>` и `& 1`. Однако у современных версий Python есть ограничение на максимальное число цифр строки, которое можно преобразовать в `int`. По умолчанию разрешено не более 4300 цифр; это параметр `sys.int_info.default_max_str_digits` (его можно увеличивать вызовом `sys.set_int_max_str_digits()`).
- в **C#** доступен класс `System.Numerics.BigInteger`.

Но в любом случае двоичная длина здесь не превышает 127 бит, поэтому битовое ДП работает очень быстро.

**ШЕСТАЯ ПОДЗАДАЧА.** Теперь  $n \leq 200$  десятичных цифр, то есть двоичная длина числа  $R$  порядка 664 бит.  $K \leq 200$ , поэтому можно использовать битовое ДП, но требуется эффективно перевести  $R$  в двоичную систему.

Проще всего выполнить деление большой десятичной строки на 2: проходим по всем её символам слева направо, получаем частное и остаток. Остаток образует очередной бит результата. Повторяем до тех пор, пока строка не станет «0».

Время работы такой конвертации:  $O(n \log R)$ , что достаточно быстро при  $n \leq 200$ . Затем выполняется стандартное ДП по битам.

**СЕДЬМАЯ ПОДЗАДАЧА.** При  $n \leq 2000$  двоичная длина достигает  $\sim 6644$  бит, и хранить трёхмерное ДП  $dp[i][r][t]$  уже слишком тяжело: память  $\Theta(nK)$  может не поместиться.

Но для digit-DP по битам достаточно хранить только два слоя: текущий и следующий:

$$dp[r][t], \quad ndp[r][t].$$

После обработки каждого бита слои меняются местами, и  $ndp$  обнуляется. Это стандартная оптимизация «rolling array», и она позволяет уложиться как по времени, так и по памяти.

**ВОСЬМАЯ ПОДЗАДАЧА.** При  $n \leq 5000$  конвертация десятичной строки в двоичное число с делением по одной цифре становится слишком медленной. Нужно ускорять операции над большими числами.

Строка  $R$  разбивается на блоки по 18 цифр (основание  $10^{18}$ ). Так получается массив целых 64-битных чисел:

$$R = a_0 \cdot 10^{18k} + a_1 \cdot 10^{18(k-1)} + \dots + a_k.$$

Деление большого числа на 2 выполняется уже по блокам: каждый блок делится как обычное 64-битное число, а остаток переносится в следующий блок. Количество блоков уменьшается примерно в 18 раз, и перевод становится существенно быстрее.

После получения двоичной записи выполняется ДП или комбинаторика (см. следующую подзадачу).

**ДЕВЯТАЯ ПОДЗАДАЧА.** При  $n \leq 30000$  и  $K \leq 100000$  переходы  $O(nK)$  становятся слишком медленными. Здесь используется комбинаторный подход и корневая декомпозиция относительно параметра  $K$ .

Заметим, что существует естественный порог когда  $K$  большое, проще считать числа с точным количеством единиц.

Пусть длина двоичной записи  $R$  равна  $n$ , и рассмотрим значения  $s$ , кратные  $K$ :

$$s = 0, K, 2K, \dots, s \leq n.$$

Для каждого такого  $s$  можно подсчитать количество чисел  $x \leq R$  с  $\text{popcount}(x) = s$  комбинаторно.

Рассмотрим двоичную запись  $R = b_0b_1\dots b_{n-1}$ . Будем проходить её слева направо и следить, сколько единиц содержится в префиксе  $R[0..i-1]$ . Когда на позиции  $i$  стоит единица, у нас появляется вариант поставить в числа  $x$  на этой позиции ноль («йти вниз» относительно  $R$ ). После этого хвост из  $(n-i-1)$  бит можно выбирать произвольно, но нужно поставить ровно

$$\text{need} = s - \text{cnt\_ones}$$

единиц. Количество способов равно биномиальному коэффициенту

$$\binom{n-i-1}{\text{need}}.$$

Суммируя такие варианты для всех позиций, где  $R$  содержит единицу, а также учитывая само число  $R$  (если  $\text{popcount}(R) = s$ ), получаем число всех  $x \leq R$  с ровно  $s$  единицами.

Так как  $K$  большое, количество значений  $s$  имеет порядок  $\frac{n}{K}$ . Каждое вычисление занимает  $O(n)$ , всего сложность порядка

$$O\left(\frac{n^2}{K}\right),$$

что эффективно при большом  $K$ .

Именно поэтому в финальном решении применяется корневая декомпозиция:

если  $K^2 \leq n$ : выгодно ДП;      если  $K^2 > n$ : выгодна комбинаторика.

Такой гибрид полностью покрывает самую жёсткую девятую подзадачу.

Почему вообще гибрид «ДП + комбинаторика» является оптимальным алгоритмом?

- При **маленьком**  $K$  выгодно использовать битовое ДП. Его сложность:  $O(nK)$ .
- При **большом**  $K$  выгодно использовать комбинаторный подход, так как количество подходящих значений  $s$  (таких что  $s \equiv 0 \pmod{K}$  и  $s \leq n$ ) равно  $\frac{n}{K}$ , поэтому общая сложность:  $O\left(\frac{n^2}{K}\right)$ .

Чтобы понять, какой метод быстрее, рассмотрим границу между ними. Ищем значение  $K$ , при котором оба варианта дают одинаковую сложность:

$$nK \approx \frac{n^2}{K} \implies K^2 \approx n \implies K \approx \sqrt{n}.$$

Тогда:

- Если  $K^2 \leq n$  (то есть  $K \leq \sqrt{n}$ ), то

$$nK \leq n\sqrt{n}, \quad \frac{n^2}{K} \geq \frac{n^2}{\sqrt{n}} = n^{3/2},$$

значит **ДП быстрее комбинаторики**.

- Если  $K^2 > n$  (то есть  $K > \sqrt{n}$ ), то

$$\frac{n^2}{K} \leq \frac{n^2}{\sqrt{n}} = n^{3/2}, \quad nK \geq n\sqrt{n},$$

значит **комбинаторика быстрее ДП**.

Таким образом, порог  $K \approx \sqrt{n}$  является разделителем между двумя методами. Именно поэтому итоговое решение использует правило:

если  $K^2 \leq n$ , применяем ДП;      иначе используем комбинаторику.

Итоговая сложность:  $O(n \cdot \sqrt{n})$