Задача A. 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 $1, \ldots, n$ остатки $(i - 1) \bmod k$ распределяются равномерно.

Пусть

$$n = q_n \cdot k + rem_n, \qquad 0 \leqslant rem_n < k.$$

Тогда количество строк, чьи индексы имеют остаток r, равно

$$rows[r] = \begin{cases} q_n + 1, & r < rem_n, \\ q_n, & r \geqslant rem_n. \end{cases}$$

Аналогично для столбцов:

$$m = q_m \cdot k + rem_m,$$

$$cols[s] = \begin{cases} q_m + 1, & s < rem_m, \\ q_m, & s \geqslant rem_m. \end{cases}$$

3. Подсчёт количества клеток каждого цвета

Все клетки с остатками (r, s) дают вклад

$$rows[r] \cdot cols[s]$$

в цвет номер

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

Таким образом:

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

После этого массив ans содержит:

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

4. Временная сложность

Так как $k \leqslant 5$, 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

```python
import sys

data = sys.stdin.read().strip().split()
n = int(data[0])
m = int(data[1])
k = int(data[2])

rows = [0] * k
qn, remn = divmod(n, k)
for r in range(k):
    rows[r] = qn + (1 if r < remn else 0)

cols = [0] * k
qm, remm = divmod(m, k)
for s in range(k):
    cols[s] = qm + (1 if s < remm else 0)

ans = [0] * k
for r in range(k):
    for s in range(k):
        t = (r + s) % k
        ans[t] += rows[r] * cols[s]

for x in ans:
    print(x)
```

Задача B. 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 $n \leqslant 10^6$ и имеет сложность `O(n)`.

```python
k = int(input())
count = 0
n = 0
while count < k:
    n += 1
    if n % 7 == 0 or n % 10 == 7:
        count += 1
print(n)
```

Подзадача 2. [Формула включений-исключений.]

В этом решении применяются комбинаторные рассуждения и не используется перебор.

```python
def count_magic_numbers(n):
    return n // 7 + (n + 3) // 10 -(n + 63) // 70

def find_kth_magic_number(k):
    left, right = 1, 10**18

    while left < right:
        mid = (left + right) // 2
        if count_magic_numbers(mid) < k:
            left = mid + 1
```

```
        else:
            right = mid
    return left

k = int(input())

result = find_kth_magic_number(k)
print(result)
```

Функция `count_magic_numbers(n)` вычисляет количество магических чисел от 1 до n по формуле включений-исключения. 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 C. 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 $m \leqslant 10$ 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 $m \geqslant 8$ 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 D. Хомяк и двоичные тайны

Первая подзадача. При $R \leqslant 2^{17}$ 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 $\operatorname{popcount}(x) \equiv 0 \pmod K$. Общее время работы — $O(R \log R)$, что полностью укладывается в ограничения первой подзадачи.

Вторая подзадача. При $R \leqslant 2^{25}$ 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 $R \leqslant 2^{44}$, 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 $L \leqslant 44$. Разобьём её на две половины: младшие m бит и старшие $L - m$ бит. Запишем число x как

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

Тогда

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

Далее:

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

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

Четвёртая подзадача. При $R \leqslant 2^{63}$ 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 $\leqslant b_i$, если $t = 0$, то можно выбирать 0 и 1 свободно. Количество единиц обновляется как $(r + \mathrm{bit}) \bmod K$.

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 $R \leqslant 2^{127}$, 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 $n \leqslant 200$ десятичных цифр, то есть двоичная длина числа R порядка 664 бит. $K \leqslant 200$, 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 $n \leqslant 200$. Затем выполняется стандартное ДП по битам.

Седьмая подзадача. При $n \leqslant 2000$ двоичная длина достигает ~ 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 $n \leqslant 5000$ 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:

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

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 $n \leqslant 30000$ и $K \leqslant 100000$ 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:

$$s = 0, K, 2K, \ldots, \quad s \leqslant n.$$

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

Рассмотрим двоичную запись $R = b_0 b_1 \ldots b_{n-1}$. 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

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

единиц. Количество способов равно биномиальному коэффициенту

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

Суммируя такие варианты для всех позиций, где R содержит единицу, а также учитывая само число R (если $\mathrm{popcount}(R) = s$), получаем число всех $x \leqslant R$ 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:

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

Такой гибрид полностью покрывает самую жёсткую девятую подзадачу.

Почему вообще гибрид «DP + 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 $s \leqslant n$) 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:

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

Тогда:

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

  $$nK \leqslant n\sqrt{n}, \qquad \frac{n^2}{K} \geqslant \frac{n^2}{\sqrt{n}} = n^{3/2},$$

  значит **DP быстрее комбинаторики**.
- Если $K^2 > n$ (то есть $K > \sqrt{n}$), то

  $$\frac{n^2}{K} \leqslant \frac{n^2}{\sqrt{n}} = n^{3/2}, \qquad nK \geqslant n\sqrt{n},$$

  значит **комбинаторика быстрее DP**.

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

$$\text{если } K^2 \leqslant n, \text{ применяем DP;} \qquad \text{иначе используем комбинаторику.}$$

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