Задача A. Серебряный статус в авиакомпании

При данных ограничениях задачу проще всего решать перебором вариантов. А именно, переберём, сколько пакетов по 30 000 будем менять на 5 000 квалификационных, а из оставшихся средств — сколько пакетов по 7 500 миль будем менять на 1 000 каждый. Из этих вариантов выберем тот, при котором количество квалификационных миль будет не менее 20 000, а общее число миль будет как можно больше.

Код программы на языке Python:

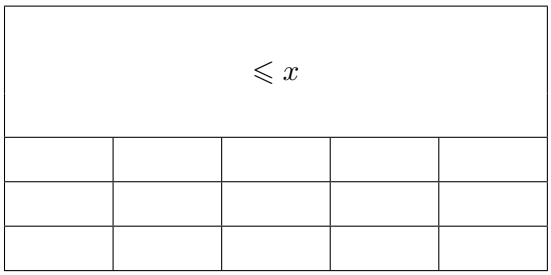
```
n = int(input())
k = int(input())
mx = 0
for i in range(n // 30000 + 1):
    for j in range((n - i * 30000) // 7500 + 1):
        nn = n - i * 30000 - j * 7500
        kk = k
        if nn < k:
            kk = nn #  квалификационные мили были потрачены на обмен
        kk += i * 5000 + j * 1000 #  новое число квалификационных миль
        nn += i * 5000 + j * 1000 #  новое число миль
        if kk >= 20000: #  квалификационных миль хватает для статуса
            if nn > mx:
                mx = nn
if mx > 0:
    print(mx)
else:
    print(-1)
```

Но задачу можно решить и разбором случаев. Определим, сколько пакетов по 1 000 квалификационных миль нам не хватает до достижения 20 000 миль. Разделим это количество на пять и обменяем как минимум такое количество пакетов по 5 000 миль, остаток обменяем по 1 000 миль. Но если осталось четыре пакета по 1 000, то вместо них выгоднее обменять дополнительный пакет в 5 000 миль, потому что при той же стоимости мы получим в итоге на 1 000 миль больше. При этих необходимых операциях на обмены могут расходоваться и имевшиеся квалификационные мили, что может привести к невозможности получения серебряного статуса, даже когда общего количества миль на обмены хватает.

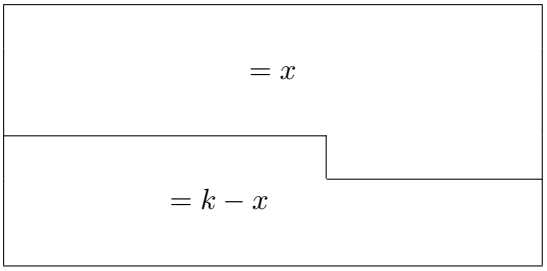
```
n = int(input())
k = int(input())
#  используем округление вверх до 1000 при подсчете недостающих пакетов
l = (max(0, 20000 - k) + 999) // 1000
if l % 5 == 4:
    #  выгоднее обменять лишний пакет в 5000 миль вместо четырех по 1000 миль
    l5 = (l + 1) // 5
    #  квалификационные мили с учетом возможного их уменьшения при обмене
    k = min(k, n - 30000 * l5) + 5000 * l5
    n -= 25000 * l5 #  оставшиеся мили
else:
    l1 = l % 5
    l5 = l // 5
    k = min(k, n - 30000 * l5 - 7500 * l1) + 5000 * l5 + 1000 * l1
    n -= (25000 * l5 + 6500 * l1) #  оставшиеся мили
if k >= 20000: #  квалификационных миль достаточно
    print(n)
else:
    print(-1)
```

Задача B. Разрез таблицы

Пусть в таблице k единиц. Тогда максимально возможное произведение $\left\lfloor \frac{k}{2} \right\rfloor \cdot \left(k - \left\lfloor \frac{k}{2} \right\rfloor \right)$. Покажем, что ответ всегда достигается. Хотим выделить разрезом часть с $x = \left\lfloor \frac{k}{2} \right\rfloor$. Найдем последнюю строку, что сумма единиц на префиксе строк $\leqslant x$.



В следующей строке возьмем на суффиксе нужное количество элементов чтобы сумма сверху стала равна x.



После этого разрез восстанавливается из рисунка выше. Асимптотика $\mathcal{O}(n \cdot m)$.

Задача C. Любовь к комбинаторным объектам

В этой задаче нам нужно искать «кратчайшие» пути в графе, но рёбра на графе не являются взвешенными; время перехода по ребру из u в v зависит от момента времени, в который мы это делаем. Несмотря на это, идеи из стандартных алгоритмов всё ещё применимы*.

Пусть в момент времени T с человеком u провели воспитательную беседу, и мы хотим понять, когда с его соседом v проведут воспитательную беседу. Другими словами, мы хотим понять время перехода по ребру из u в v в момент времени T.

Нам нужно найти первый из моментов времени $a_v, a_v + t_v, \ldots$, который при этом позже T. Сдвинем нашу шкалу времени назад на a_v. Теперь нам нужно найти минимальное число, кратное t_v, большее $T - a_v$. Это будет

$$\left\lceil \frac{T - a_v + 1}{t_v} \right\rceil \cdot t_v$$

Таким образом, мы перейдём в вершину v в момент времени

$$a_v + \max\left(0, \left\lceil \frac{T - a_v + 1}{t_v} \right\rceil \cdot t_v\right)$$

Взяв $T = \mathrm{dist}_u$, мы можем пересчитать этой величиной dist_v.

*Более формально, для алгоритмов поиска расстояний из одной вершины до всех остальных требуется выполнение трёх свойств:

- Монотонность: $\mathrm{Len}(\mathrm{path}) \leqslant \mathrm{Len}(\mathrm{path} + e)$
- Консистентность: если $\mathrm{Len}(\mathrm{path}_1) = \mathrm{Len}(\mathrm{path}_2)$, то $\mathrm{Len}(\mathrm{path}_1 + e) = \mathrm{Len}(\mathrm{path}_2 + e)$
- Оптимальность начала: если $\mathrm{Len}(\mathrm{path}_1) \leqslant \mathrm{Len}(\mathrm{path}_2)$, то $\mathrm{Len}(\mathrm{path}_1 + e) \leqslant \mathrm{Len}(\mathrm{path}_2 + e)$

В подгруппе 2 граф является деревом, то есть до каждой вершины существует единственный простой путь.

В подгруппе 3 время перехода по ребру всегда равно 1, поэтому мы можем использовать обычный поиск в ширину.

В подгруппе 5 время перехода по ребру не больше 20, поэтому мы можем использовать поиск в ширину с несколькими очередями (1-K BFS).

В подгруппах 7 и 8, а также в полном решении нужно применить алгоритм Дейкстры.

Задача D. Черепашка наносит ответный удар

Для $n, m \leqslant 100$ можно написать переборное решение: перебираем, какую клетку выберет Рафаэль, и для каждого варианта считаем, сколько удовольствия получит Микеланджело. Это считается простейшей динамикой-черепашкой.

Для дальнейших решений выделим один из путей S из $(1,1)$ в (n,m), который на исходной пицце доставляет Микеланджело максимальное удовольствие. Клетки этого пути назовём $s_1, s_2, \ldots, s_{n+m-1}$. Заметим, что если Рафаэль не выберет ни одну из этих клеток, то Микеланджело выберет путь S и получит максимально возможное удовольствие (благодаря неотрицательным по удовольствию кускам пиццы Рафаэль всегда может сделать ход, который не увеличит доступное Микеланджело удовольствие). А значит, если Рафаэль претендует на что-то большее, он должен заблокировать одну из клеток $s_1, \ldots, s_{n+m-1}$.

Уже из одного этого замечания мы получаем решение за $\mathcal{O}(nm(n+m))$: давайте переберём клетку на пути, которую выберет Рафаэль, и для каждого из этих вариантов снова посчитаем ответ.

Для дальнейших продвижений заметим, что после блокировки клетки (i,j) у Микеланджело будет два варианта: получить на $2a_{i,j}$ меньше удовольствия, чем максимум, то есть $maxS - 2a_{i,j}$, или пройти из $(1,1)$ в (n,m), не посетив клетку (i,j). Первый вариант не требует дополнительных вычислений, а вот второй создаёт некоторые сложности. Давайте разберёмся с ними.

Для клетки (i,j), чтобы гарантировать, что путь не пройдёт через неё, достаточно показать, что на пути будет лежать клетка (a,b), для которой выполняется либо $a < i$ и $b > j$, либо $a > i$ и $b < j$. Отсюда появляется новое решение с той же асимптотикой, но которое уже можно будет оптимизировать: давайте переберём все клетки (a,b) и для каждой клетки s_k сохраним возможность обойти её за $dpS[a][b] + dpT[a][b] - a[a][b]$, где dpS и dpT — вспомогательные динамики, выдающие максимальное удовольствие при движении из $(1,1)$ в (a,b) и из (a,b) в (n,m) соответственно. По всем таким вариантам для каждой s_k нам нужно сохранить максимум; назовём его $dp[k]$. Тогда ответ формируется как $\min_k \big(\max\big(dp[k],\ maxS - 2a[s_k]\big)\big)$.

Теперь давайте оптимизируем и это решение. Заметим, что нам не нужно перебирать всевозможные клетки (a,b), через которые можно обойти клетку (i,j). Достаточно рассмотреть лишь клетки $(i+1, q < j)$ и клетки $(p < i, j+1)$. К сожалению, напрямую перебирать все такие клетки всё ещё долго. Однако заметим, что нам нужно уметь считать $\max\big(dpS[x][y] + dpT[x][y] - a[x][y]\big)$ для префиксов точек по x в конкретном столбце и по y в конкретной строке, а именно для наборов $(i+1,1), (i+1,2), \ldots, (i+1,j-1)$ и $(1,j+1), (2,j+1), \ldots, (i-1,j+1)$.

Значения таких префиксных максимумов можно предподсчитать. Таким образом, за $\mathcal{O}(nm)$ предподсчёта мы получаем возможность для каждой клетки (i,j) узнавать максимальное удовольствие маршрута, который не будет посещать её, за $\mathcal{O}(1)$. Итоговая асимптотика решения составляет $\mathcal{O}(nm + (n+m)) = O(nm)$.

Отметим также, что значительное количество баллов могло набирать решение за $\mathcal{O}(nm\log(nm))$, которое оптимизирует второе решение за $\mathcal{O}(nm(n+m))$ при помощи структур данных.

Задача E. Составляем многоугольник

Заметим, что из подотрезка массива можно составить многоугольник, если удвоенный максимум на отрезке меньше суммы на отрезке.

Подгруппы 1, 2 ($N \leqslant 5000$).

Перебираем левую и правую границы, поддерживаем сумму и максимум. Время $O(n^2)$.

Подгруппа 3 ($a_i \leqslant a_{i+1}$).

Максимум это всегда самый правый элемент отрезка, поэтому можно зафиксировать правую границу отрезка и искать подходящие левые границы бинарным поиском с префиксными суммами. Время $\mathcal{O}(n \cdot \log n)$.

Подгруппа 4 (массив случайно перемешан).

Находим максимум в массиве. Рекурсивно решаем задачу на массиве всех элементов левее максимум, аналогично решаем на массиве правее максимума.

Теперь нужно посчитать количество хороших отрезков, содержащих максимум. Для этого нужно найти для каждой правой границы минимальную подходящую левую, что можно делать методом двух указателей.

Так как максимум каждый раз делит массив примерно поровну, то решение работает за $\mathcal{O}(n \cdot \log n)$ ожидаемо. Заметим, что оценка времени работы в этом алгоритме такая же, как в алгоритме QuickSort.

Полное решение.

Воспользуемся техникой «разделяй и властвуй».

Рекурсивно делим массив пополам: ответ = левая часть + правая часть + отрезки, пересекающие середину.

Посчитаем все отрезки, которые пересекают середину и в которых максимум находится в левой части, отрезки с максимумом в правой части рассматриваются аналогично.

Если мы переберем левую границу отрезка, то на правую границу наложены всего два ограничения: во первых правая граница должна быть достаточно большой, что бы сумма превосходила два максимума, но не настолько большой, что бы максимум в правой части отрезка превзошел максимум в левой части. Эти две границы можно искать бинарным поиском. После нахождения границ пересчитываем ответ.

Время работы: $\mathcal{O}(n \cdot \log^2 n)$.

Задача F. Опять деревья...

Подгруппа 1.

Для начала заметим, что если XOR чисел в дереве не равен нулю, то ответ 0. Посмотрим на какое-то ребро. При удалении его, дерево разбивается на поддерево и наддерево. Если XOR в поддереве не равен нулю, то мы не сможем дальше удалить ребра так, чтобы XOR в каждой компоненте был равен нулю. При этом если XOR в поддереве равен нулю, то и в наддереве он тоже равен нулю, и удаление этого ребра никак не повлияет на остальные ребра, поэтому ответ — это 2^{cnt}, где cnt — это количество поддеревьев с XOR 0, кроме всего дерева. Асимптотика $\mathcal{O}(n)$.

Подгруппа 2.

Достаточно перебрать все множества ребер и проверить красивое оно или нет. Асимптотика $\mathcal{O}(2^n \cdot n)$.

Подгруппа 3.

Сделаем $\text{dp}[v][x]$ — количество вариантов выбрать множество ребер из поддерева этой вершины, включая ребро из вершины v в предка, так, чтобы отрезанные вершины в поддереве имели XOR равный x. Чтобы пересчитать dp нужно сначала посчитать XOR свертку значений dp сыновей. XOR свертка массивов a и b длины 2^k — это массив $c_i = \sum a_j \cdot b_{j \oplus i}$. Ее пока что будем считать за произведение размеров массивов. После XOR свертки всех сыновей, мы должны обработать вариант удаления ребра из v в ее предка. Для этого переберем какой был x и если $\text{xor\_subtree}[v] \oplus x \in b$, то нужно прибавить $\text{dp}[v][x]$ к $\text{dp}[v][0]$. Таким образом это решение работает за $\mathcal{O}(n \cdot 32^2)$.

Подгруппа 4.

Нам нужно посчитать количество разбиений массива на подотрезки, где XOR в каждом подотрезке лежит в b. Для этого будем идти слева направа по массиву и поддерживать $\text{dp}[i]$ — количество разбиений префикса длины i на подотрезке. Пересчет $\text{dp}[i] = \sum \text{dp}[j]$, где $\text{pref}[i] \oplus \text{pref}[j-1] \in b$, где $\text{pref}[i] = \oplus a_j$ по $j \leqslant i$. Такой пересчет оптимизируется с помощью `std::map` и мы получаем решение за $\mathcal{O}(n \log n \cdot k)$.

Подгруппы 5–7.

Выберем базис из множества b и будем хранить dp из третьей группы в этом базисе. Из-за перехода в другой базис немного усложняется реализация, а в остальном dp остается прежней. Также, есть решение без базиса, но оно не оптимизируется до полного решения. Решение работает за $\mathcal{O}(n \cdot 4^k)$ из-за того, что одна XOR свертка работает за 4^k.

Подгруппы 8, 9.

Будем делать XOR свертку алгоритмом Адамара. Получаем асимптотику $\mathcal{O}(n \cdot 2^k \cdot k)$. Также стоит отметить, что решения без Адамара могли пройти эти группы, если использовать некоторые неасимптотические оптимизации, которые на самом деле могут быть асимптотическими.

Подгруппа 10.

Мы хотим избавиться от домножения на k. Одна XOR свертка адамаром выглядит так: считаем преобразование Адамара от обоих массивов, затем считаем поточечное умножение и делаем обратное преобразование Адамара. Вместо того, чтобы хранить $\mathrm{dp}[v][x]$, будем хранить $\mathrm{hadamard}(\mathrm{dp}[v][x])$, то есть преобразование Адамара. Его формула такая: $\mathrm{hadamard}(a)[i] = \sum a_j \times (-1)^{\mathrm{popcount}(i \,\&\, j)}$. Тогда свертка делается за $\mathcal{O}(2^k)$ поточечным произведением. Теперь возникает сложность со вторым пересчетом, где мы хотели посчитать значение $\mathrm{dp}[v][x]$ в каких-то точках x и прибавить это к $\mathrm{dp}[v][0]$.

Начнем с первого шага. Нужно посчитать $\sum \mathrm{dp}[v][b_i \oplus \mathrm{xor\_subtree}[v]]$. Стоит отметить, что если $\mathrm{xor\_subtree}[v]$ не раскладывается по базису, то такой пересчет делать не нужно. Для начала посчитаем такую сумму в предположении, что $\mathrm{xor\_subtree}[v] = 0$. Пусть p — массив размера 2^k, где на позициях b_i записаны 1, а на остальных позициях нули. Тогда сумма, которую нужно посчитать — это нулевой элемент XOR свертки $\mathrm{dp}[v]$ и p. Саму свертку $\mathrm{dp}[v]$ и p мы посчитать не можем, но мы можем посчитать $q = \mathrm{hadamard}(\text{свертка } \mathrm{dp}[v] \text{ и } p)$, поскольку это поточечное произведение $\mathrm{hadamard}(\mathrm{dp}[v])$ и $\mathrm{hadamard}(p)$, а $\mathrm{hadamard}(p)$ мы можем посчитать один раз в начале программы. Теперь нам интересен нулевой элемент от обратного преобразования q, а это просто $(\sum q_i)/2^k$. Теперь вспомним, что у нас есть $\mathrm{xor\_subtree}[v]$, но чтобы его учесть, достаточно добавить в свертку $\mathrm{dp}[v]$ и p, массив, где единица стоит только на позиции $\mathrm{xor\_subtree}[v]$. hadamard от такого массива вычисляется за $\mathcal{O}(2^k)$ по определению. Таким образом мы за $\mathcal{O}(2^k)$ посчитали нужную сумму.

Теперь нужно посчитать новый $\mathrm{hadamard}(\mathrm{dp}[v])$ после такого, как мы бы прибавили к $\mathrm{dp}[v][0]$ посчитанную сумму. Но по определению преобразования Адамара мы можем найти коэффициенты с которыми учтется прибавленная сумма к старому значению $\mathrm{hadamard}(\mathrm{dp}[v])$.

Таким образом в этом решении делать преобразование Адамара в явном виде нужно только для вычисления $\mathrm{hadamard}(p)$, а в остальном мы явно вычисляем нужны коэффициенты по определению и получаем решение $\mathcal{O}(n \cdot 2^k)$.