Разбор задач

Задача 1. Рекламные паузы

К длине фильма нужно добавить количество рекламных пауз, умноженное на продолжительность одной паузы b. Рекламные паузы вставляются через каждые a минут, поэтому для подсчёта их числа нужно поделить n нацело на a. Но если n делится на a (остаток от деления n на a равен 0), то время показа последней рекламной паузы совпадёт с окончанием фильма, поэтому реклама показываться не будет и число рекламных пауз нужно уменьшить на 1. Возможное решение.

```
a = int(input())
b = int(input())
n = int(input())
n_pauses = n // a
if n % a == 0:
    n_pauses -= 1
print(n + n_pauses * b)
```

Можно написать решение и без использования if.

```
a = int(input())
b = int(input())
n = int(input())
n_pauses = (n - 1) // a
print(n + n_pauses * b)
```

Задача 2. Популярный пост

Если какое-то из чисел a, b, c больше суммы двух других, то это число и является ответом: меньшее количество пользователей не могло оставить столько реакций, а если каждый пользователь поставил самую популярную реакцию из трёх и какую-то из двух оставшихся (или поставил одну самую популярную реакцию), то этот вариант будет реализован. Поэтому можно просто вывести наибольшее из трёх чисел.

```
a = int(input())
b = int(input())
c = int(input())
print(max(a, b, c))
```

Такое решение набирает 50 баллов.

Рассмотрим теперь случай, когда максимальное из трёх чисел меньше суммы двух других. Необходимо наименьшему количеству пользователей назначить по две или одной реакции, чтобы общее число реакций трёх видов было равно a, b, c. Реализуем «жадный» алгоритм — назначим очередному пользователю две различные реакции, уменьшив на 1 счётчики их количеств. Чтобы последующим пользователям также оставалась возможность выбрать две реакции, необходимо избежать ситуации, когда только одно из чисел a, b, c является ненулевым. то есть выбирать на каждом шаге два наибольших значения из a, b, c.

Приведём пример такого решения: входные числа упорядочиваются так, чтобы выполнялось условие $a \leqslant b \leqslant c$, затем в цикле наибольшие значения c и b уменьшаются на 1, а значение ответа увеличивается на 1, после чего числа опять переупорядочиваются.

```
a = int(input())
b = int(input())
c = int(input())
ans = 0
a, b, c = sorted([a, b, c])
while c > 0:
```

```
    c -= 1
    b -= 1
    ans += 1
    a, b, c = sorted([a, b, c])
print(ans)
```

Такое решение набирает 50 баллов. Но если объединить эти два решения и выводить $\max(a, b, c)$ в случае, когда одно из чисел не меньше суммы двух других, а иначе реализовать «жадный» алгоритм, то решение получит 75 баллов.

Теперь избавимся от цикла в «жадном» алгоритме. Поскольку пользователь не мог оставить более двух реакций, то для количества пользователей k выполняется неравенство

$$k \geqslant \left\lceil \frac{a+b+c}{2} \right\rceil,$$

где $\lceil x \rceil$ — значение x, округлённое вверх до целого числа.

Наш жадный алгоритм и реализует такую формулу: на каждом шаге, кроме, возможно, последнего, будут уменьшаться два из трёх чисел a, b, c.

Таким образом, решение задачи дают формулы: $k = \max(a, b, c)$ если одно из чисел не меньше суммы двух других (то есть если $2\max(a, b, c) \geqslant a + b + c$) или $k \left\lceil \frac{a+b+c}{2} \right\rceil$ иначе. Это можно записать и при помощи одной формулы

$$k = \max\left(a, b, c, \left\lceil \frac{a+b+c}{2} \right\rceil\right).$$

Деление с округлением вверх можно реализовать, например, на языке Python при помощи выражения (a+b+c+1) // 2.

Пример такого решения.

```
a = int(input())
b = int(input())
c = int(input())
print(max(a, b, c, (a+b+c+1) // 2))
```

Задача 3. Встреча у фонтана

Каждый из двух персонажей перед встречей пройдёт расстояние от своего дома до фонтана нечётное число раз. Условие задачи формализуем так: нужно найти такое минимальное t, что $t = mx$, $t = py$, где x и y — нечётные числа.

В неэффективном решении переберём возможные значения t и найдём такое значение t, что t делится на m и p, и что частные от деления нечётные.

Пример такого решения.

```
m = int(input())
p = int(input())
t = 0
while not(t % m == 0 and t % p == 0 and t // m % 2 == 1 and t // p % 2 == 1):
    t += 1
print(t)
```

Такое решение набирает 20 баллов из первой группы тестов. Чтобы набрать 30 баллов нужно добавить дополнительную проверку — выводить число -1, если не удалось найти ответ, проверив значения t до 10^6.

Улучшим это решение. Можно перебирать не все значения t, а только значения, равные mx для нечётного x или, наоборот, py для нечётного y. А лучше рассмотреть два случая, и если $m > p$, то перебирать числа вида mx, иначе перебирать значения вида py.

Пример такого решения.

```
m = int(input())
p = int(input())
if m > p:
    t = m
    while not(t % p == 0 and t // p % 2 == 1):
        t += 2 * m
else:
    t = p
    while not(t % m == 0 and t // m % 2 == 1):
        t += 2 * p
print(t)
```

Это решение набирает 50 баллов (или 60 баллов, если выводить -1, когда не удалось найти ответ).

Рассмотрим полное решение. В равенстве $mx = py$ поделим обе части на общий делитель чисел m и p. Тогда если d — наибольший общий делитель m и p, и $a = m/d$, $b = p/d$, то числа a и b — взаимно простые и $ax = by$.

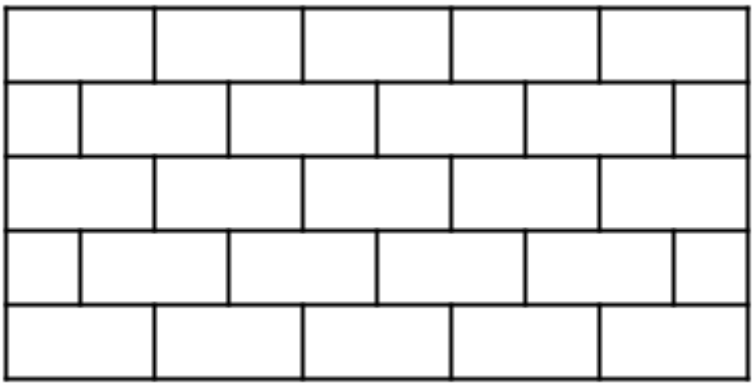
Поскольку x и y — нечётные, то при одном чётном из чисел a или b (они не могут быть чётными одновременно) одна сторона равенства будет чётной, а другая — нечётной, то есть такое невозможно и нужно вывести -1. Иначе в силу взаимной простоты a и b необходимо взять $y = a$ и $x = b$.

Алгоритм Евклида можно реализовать самостоятельно, заменяя большее число на его остаток от деления на меньшее, или использовать функцию gcd из модуля math в Python.

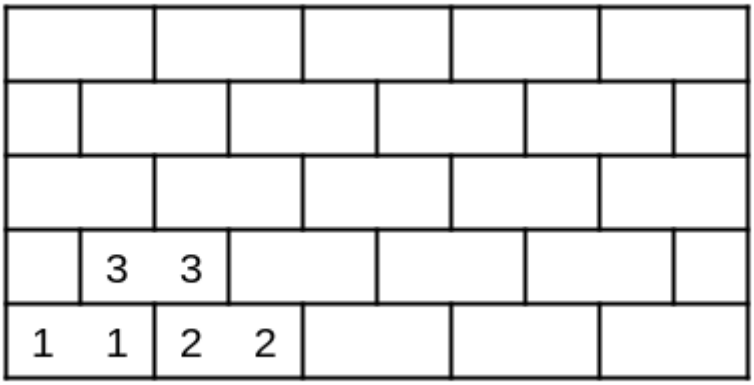
```
m = int(input())
p = int(input())

def gcd(a, b):
    while b > 0:
        a, b = b, a % b
    return a

d = gcd(m, p)
a = m // d
b = p // d
if a % 2 == 0 or b % 2 == 0:
    print(-1)
else:
    print(m * b)
```

Задача 4. Раскраска стены

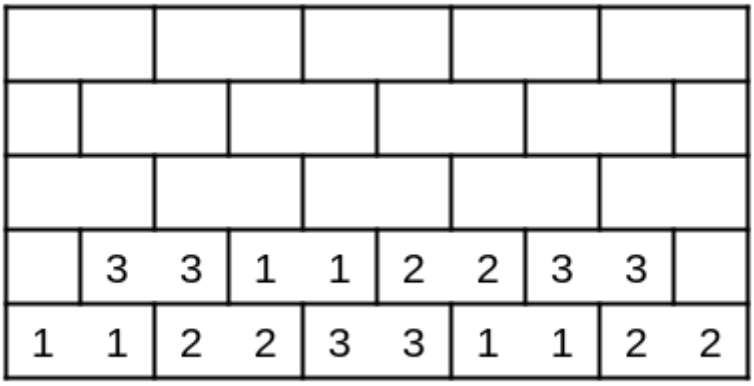
Для покраски всегда достаточно трёх цветов. Рассмотрим достаточно большой фрагмент стены и попробуем покрасить его в три цвета.



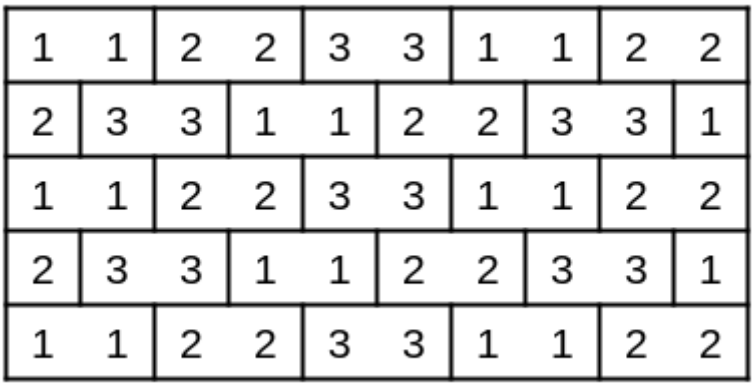
Начнём с целого кирпича в левом нижнем углу. Если три кирпича попарно соприкасаются, то они будут покрашены в разные цвета.



Продолжим, покрасив в три цвета два нижних ряда кирпичей.



Остальные ряды будут повторением двух нижних рядов.



Видим, что в нижнем и всех нечётных рядах повторяется последовательность цифр 112233, а во втором и других чётных рядах — та же последовательность, но со сдвигом на 3 цифры. Решение будет таким: создадим строку, повторив последовательность 112233 несколько раз. Для построения нечётных (считая снизу) рядов кирпичей выведем первые m символов этой строки, а в чётных рядах — m символов, но пропустив 3 начальных символа последовательности.

Есть ещё один частный случай. При $n = 1$ для раскраски достаточно двух цветов, т.к. ряд кирпичей всего один. В этом случае нужно образовать строку повторением последовательности 1122. При $n = 1$, $m = 2$ нужен только один цвет (т.к. кирпич всего один), но этот случай можно не рассматривать отдельно.

Пример правильного решения.

```
n = int(input())
m = int(input())
if n > 1:
    s = "112233" * 4
else:
    s = "1122" * 5
for i in range(n, 0, -1):
    if i % 2 == 1:
        print(s[:m])
    else:
        print(s[3:3 + m])
```

Задача 5. Проблемы логистики

Сначала рассмотрим переборные решения. Можно рассмотреть все $n!$ способов распределить контейнеры по машинам. Для каждого способа проверим выполнение ограничения стоимости перевозки одного контейнера и выберем из подходящих способов тот, который имеет наибольшую стоимость.

Для $n \leqslant 3$ допустимо перебрать все возможные перестановки непосредственно в тексте программы, и такое решение наберёт 20 баллов. Реализация алгоритма перебора всех перестановок или использование стандартной функции языка программирования (например, в Python это permutations из модуля itertools, в C++ — next_permutation из algorithms), может получить до 40 баллов. Пример такого решения.

```
import itertools

n, k = map(int, input().split())
w = list(map(int, input().split()))
p = list(map(int, input().split()))
```

```
ans = -1
for p1 in itertools.permutations(p):
    s = 0
    for i in range(n):
        if w[i] * p1[i] > k:
            break
        s += w[i] * p1[i]
    else:
        ans = max(ans, s)
print(ans)
```

Чтобы получить полное решение, сделаем наблюдение. Пусть есть два контейнера массами w_1 и w_2, при этом $w_1 > w_2$. Первый контейнер перевозят со стоимостью перевозки p_1, а второй — p_2. Если при этом $p_1 < p_2$, а $w_1 p_2 \leqslant k$, то есть можно поменять контейнеры местами, соблюдая ограничение стоимости перевозки, то стоимость перевозки увеличится.

Действительно, в первом случае стоимость перевозки равна $w_1 p_1 + w_2 p_2$, во втором случае — $w_1 p_2 + w_2 p_1$. Докажем, что

$$w_1 p_1 + w_2 p_2 < w_1 p_2 + w_2 p_1.$$

Это эквивалентно неравенству

$$(w_1 - w_2)(p_1 - p_2) < 0.$$

которое верно, т.к. $w_1 - w_2 > 0$, $p_1 - p_2 < 0$.

Таким образом, самому тяжёлому контейнеру нужно назначить самый высокий тариф из возможных с соблюдением ограничения на стоимость перевозки. Иначе, если этот тариф назначен более лёгкому контейнеру, поменяв их местами, увеличим стоимость перевозки. Поэтому для решения задачи можно перебрать все контейнеры в порядке невозрастания их масс и для каждого контейнера подобрать наибольший допустимый тариф, удаляя затем этот тариф из рассмотрения.

Пример такого решения.

```
n, k = map(int, input().split())
w = list(map(int, input().split()))
p = list(map(int, input().split()))

w.sort(reverse=True)
p.sort(reverse=True)

ans = 0
for wi in w:
    i = 0
    while i < len(p) and p[i] * wi > k:
        i += 1
    if i < len(p):
        ans += p[i] * wi
        p.pop(i)
    else:
        print(-1)
        break
else:
    print(ans)
```

В этом решении мы упорядочим по невозрастанию массы контейнеров и тарифы. Перебирая массы в порядке невозрастания, находим первый подходящий по ограничению тариф, он и является

наибольшим подходящим. Если такой тариф нашёлся, добавляем к сумме p[i] * wi и удаляем из списка найденный тариф, иначе выводим -1.

Это решение сложностью $O(n^2)$, так как поиск каждого подходящего тарифа и удаление его из списка имеет сложность $O(n)$, набирает 60 или чуть больше баллов.

Чтобы получить 100 баллов необходима оптимизация. При рассмотрении следующего контейнера множество допустимых для него тарифов увеличивается, к нему добавятся какие-то тарифы (большие, чем ранее рассмотренные). После этого из всех допустимых тарифов, включая добавленные, нужно выбрать наибольший.

Можно хранить все допустимые тарифы для рассматриваемого контейнера в отдельном списке, упорядоченном по неубыванию. Тогда новые допустимые тарифы будем добавлять в конец списка (также в порядке неубывания), а потом извлечём из этого списка последний элемент, удалив его при этом, — это и есть наибольший среди всех элементов списка. В программировании такая структура данных (где можно добавлять или удалять элементы в конце списка) называется стеком, но для реализации стека можно использовать обычный список в языке Python или vector в языке C++. Особенностью стека сложность $O(1)$ всех операций с ним.

Такое решение имеет сложность $O(n \log n)$ ввиду использования сортировки. Сложность последующих операции с проходом по элементам и работой со стеком — $O(n)$. Пример такого решения.

```
n, k = map(int, input().split())
w = list(map(int, input().split()))
p = list(map(int, input().split()))

w.sort(reverse=True)
p.sort()

ans = 0
i = 0
possible = []
for wi in w:
    while i < len(p) and p[i] * wi <= k:
        possible.append(p[i])
        i += 1
    if len(possible) > 0:
        ans += possible[-1] * wi
        possible.pop()
    else:
        print(-1)
        break
else:
    print(ans)
```

В языке C++ возможна и другая реализация эффективного решения с использованием контейнера multiset (мультимножество). Этот контейнер позволяет хранить упорядоченный набор чисел, а также быстро находить наибольший элемент, который не превосходит данный. Используем его в решении: поместим все доступные тарифы в multiset и переберём контейнеры в порядке неубывания масс. Для каждого контейнера определим максимально допустимое значение тарифа, найдём в multiset наибольшее значение, не превосходящее данное, и удалим его.

Пример такого решения.

```
#include<iostream>
#include<set>
#include<algorithm>
#include<vector>
using namespace std;
int main()
```

```
{
    long long n, k, ans, x, y;
    cin >> n >> k;
    vector<int> w;
    multiset<long long> p;
    for (int i = 0; i < n; ++i)
    {
        cin >> x;
        w.push_back(x);
    }
    for (int i = 0; i < n; ++i)
    {
        cin >> x;
        p.insert(x);
    }
    ans = 0;
    sort(w.rbegin(), w.rend());
    for (auto x : w)
    {
        auto it = p.upper_bound(k / x);
        if (it == p.begin())
        {
            cout << -1 << endl;
            return 0;
        }
        --it;
        ans += (*it) * x;
        p.erase(it);
    }
    cout << ans << endl;
}
```

В языке Python нет аналога контейнера multiset, но возможно реализовать похожее решение с использованием идей, часто возникающих в сложных алгоритмах.

1. «*Ленивое удаление*». Запишем все тарифы в неубывающий список и будем искать подходящий тариф двоичным поиском. Но вместо удаления тарифа из списка (что выполняется долго) пометим его как удалённый. Поэтому после нахождения максимального подходящего тарифа двоичным поиском, если он оказался удалённым, нужно найти наибольший неудалённый элемент слева от данного.

2. «*Ссылочная реализация*». Чтобы не просматривать все элементы списка, пропуская удалённые, для каждого элемента запишем значение prev[j] — какого-то элемента, который находится левее данного. Если элемент j не был удалён, то prev[j] == j. Иначе все элементы между prev[j] и j будут удалёнными, поэтому их не надо просматривать, и в поиске неудалённого элемента нужно просто переходить от j к prev[j]. Правда, prev[j] также может оказаться удалённым, поэтому требуется переходить в цикле от элемента j к элементу prev[j], пока не найдётся неудалённый элемент (j == prev[j]) или пока элементы не закончатся (в этом случае j == −1).

3. «*Сжатие путей*». В этой реализации поиск неудалённого элемента осуществляется долго, т.к. на самом деле при переходе вида j = prev[j] значение j будет уменьшаться на 1. Чтобы ускорить поиск можно использовать технологию сжатия путей — если мы один раз прошли по пути и нашли для какого-то элемента первый неудалённый элемент, нужно обновить значение

prev для начала этого пути, чтобы в дальнейшем не проходить по нему целиком. В примере программы ниже это делается присваиванием prev [j_found] = j − 1.

```
import bisect
import sys

n, k = map(int, input().split())
w = list(map(int, input().split()))
p = list(map(int, input().split()))
prev = [i for i in range(n)]

w.sort(reverse=True)
p.sort()

ans = 0
for wi in w:
    j_found = bisect.bisect_right(p, k // wi) - 1
    j = j_found
    while j >= 0 and prev[j] != j:
        j = prev[j]
    if j == -1:
        print(-1)
        sys.exit(0)
    ans += wi * p[j]
    prev[j_found] = j - 1

print(ans)
```

Также отметим, что во всех вариантах решений допустимо, наоборот, перебирать тарифы в порядке невозрастания и для каждого тарифа подбирать максимальный по массе допустимый контейнер.