Разбор задач

Задача 1. Качели

Первую группу тестов можно пройти при помощи переборного решения. Будем перебирать значение ответа (массу камня) в переменной ans. Для каждого значения ans проверим, смогут ли дети качаться с камнем данной массы. Переберём все возможные варианты размещения детей и камня, всего таких способов 6 (тремя способами можно выбрать одного ребёнка, который сидит на одном конце качелей, и двумя способами — конец, на который положат камень). Для каждого способа посчитаем модуль разности весов на концах качелей, если он не превосходит d, то ответ найден.

Пример такого решения.

```
a = int(input())
b = int(input())
c = int(input())
d = int(input())
ans = 0
while True:
    if (abs(a+b-c-ans) <= d or abs(a+b-c+ans) <= d
     or abs(b+c-a-ans) <= d or abs(b+c-a+ans) <= d
     or abs(a+c-b-ans) <= d or abs(a+c-b+ans) <= d):
        print(ans)
        break
    ans += 1
```

Чтобы набрать 100 баллов можно в этом решении заменить линейный поиск ответа на двоичный. Но такое решение довольно сложно, т.к. необходимо правильно определить границы для двоичного поиска. Нет нужды приводить такое решение, потому что у задачи есть более элегантное решение сложности $O(1)$.

Для того, чтобы минимизировать разницу весов на концах качелей, необходимо на одну сторону посадить самого тяжёлого ребёнка, а на другую сторону — двух других детей. Посчитаем разницу масс на концах качелей в этом случае, если она не превосходит d, то камень не нужен, и ответом будет 0. Иначе вычтем из этой разницы значение d, это и будет ответ.

Пример такого решения.

```
a = int(input())
b = int(input())
c = int(input())
d = int(input())
side1 = max(a, b, c)
side2 = a + b + c - side1
print(max(0, abs(side1 - side2) - d))
```

Задача 2. Фонари

Вывод программы различается для случаев, когда размещение фонарей возможно или невозможно. Один фонарь первого типа освещает $2x + 1$ домов, второго типа — $2y + 1$ домов. Поэтому сначала проверим, существует ли решение задачи, то есть посчитаем максимальное число домов, которые могут освещаться a фонарями первого вида и b фонарями второго вида. Если это число меньше n, то нужно вывести -1. Иначе получим ответ при помощи «жадного» алгоритма: для минимизации числа фонарей выберем фонарь, который освещает больше домов, и разместим его так, чтобы множество домов, которое он освещает, непосредственно примыкало к уже освещённым домам. Повторим этот процесс, пока все дома не станут освещены.

В приведённом ниже решении мы предполагаем, что фонари первого вида освещают большее число домов, то есть $x \geqslant y$. Если это не так, то поменяем два вида фонарей местами. Поэтому будем стараться всегда использовать фонарь первого вида. В переменной last_lighted хранится

номер последнего освещённого дома. Цикл продолжается, пока не все дома освещены, то есть пока last_lighted $<$ n. Если есть ещё фонари первого вида, то используется фонарь первого вида, и количество освещённых домов увеличивается на $2x + 1$ для фонарей первого типа и на $2y + 1$ для второго типа. При выводе координаты нового освещённого дома необходимо учесть, что координата дома в выводе не может быть больше n.

Пример решения.

```
n = int(input())
a = int(input())
x = int(input())
b = int(input())
y = int(input())

if a * (2 * x + 1) + b * (2 * y + 1) < n:
    print(-1)
else:
    if x < y:
        x, y = y, x
        a, b = b, a
    last_lighted = 0
    while last_lighted < n:
        if a > 0:
            print(min(n, last_lighted + x + 1), x)
            last_lighted += 2 * x + 1
            a -= 1
        else:
            print(min(n, last_lighted + y + 1), y)
            last_lighted += 2 * y + 1
            b -= 1
```

Задача 3. Красная Шапочка на болоте

В подгруппе $N \leqslant 15$ можно написать переборное решение. Рассмотрим все подмножества кочек, таких подмножеств будет 2^N. Пусть Красная Шапочка прыгает на выбранные кочки. Проверим, возможно ли это и сколько у неё останется энергии, потом выберем подмножество с наибольшей остаточной энергией.

В подгруппе $N \leqslant 900$ предполагается решение сложности $O(N^2)$ с использованием идеи динамического программирования. Пусть $f(i)$ — максимальное количество энергии, которое может остаться у Красной Шапочки, когда она окажется в точке с координатой i. Тогда $f(0) = E$, $f(N+1)$ — ответ на задачу. Будем последовательно вычислять $f(1), f(2), \ldots, f(N+1)$. Для вычисления $f(i)$ рассмотрим j — координату точки, из которой мы прыгнули в i. Тогда количество энергии в точке i будет равно $f(j) - (i - j) + a_i$. Переберём все j от 0 до $i - 1$ и выберем наибольшее возможное значение.

$$f(i) = \max_{0 \leqslant j < i} f(j) - (i - j) + a_i$$

При этом удобно считать, что $a_{n+1} = 0$ (поляна обрабатывается как обычная кочка с нулевой дополнительной энергией), а также необходимо проверить, что значение $f(j) - (i - j) \geqslant 0$, иначе Красная Шапочка не сможет прыгнуть из j в i.

Пример такого решения.

```
import sys
e = int(input())
n = int(input())
a = [0] + [int(input()) for i in range(n)] + [0]
f = [e] + [0] * (n + 1)
```

```
for i in range(1, n + 2):
    m = -1
    for j in range(0, i):
        m = max(m, f[j] - (i - j))
    if m < 0:
        print(-1)
        sys.exit(0)
    f[i] = m + a[i]
print(f[n + 1])
```

Для дальнейшего решения задачи заметим, что какие бы кочки Красная Шапочка ни посетила, у неё всегда уйдёт ровно $N + 1$ единица энергии на совершение всех прыжков из точки 0 в точку $N + 1$. Действительно, пусть она посетила кочки с координатами $0 < x_1 < x_2 < \cdots < x_k < N + 1$, тогда суммарные затраты энергии равны

$$(x_1 - 0) + (x_2 - x_1) + \cdots + (x_k - x_{k-1}) + (N + 1 - x_k) = N + 1 - 0 = N + 1$$

Значит, количество энергии, оставшееся у Красной Шапочки после достижения поляны, составит $E+S-(N+1)$, где S — суммарная энергия, которую Красная Шапочка получила на всех посещённых кочках. Следовательно, необходимо максимизировать значение S. Для этого достаточно посетить все такие кочки i, у которых $a_i > 0$, проверив, что у Красной Шапочки достаточно энергии, чтобы попасть в каждую из них, то есть после каждого прыжка запас энергии неотрицателен.

Ниже приведено решение на языке Python. Значения a_i можно не сохранять в массиве, а обрабатывать их сразу после считывания. Значение s сразу проинициализируем начальным значением энергии Красной Шапочки и будем добавлять к нему положительные значения a_i. Тогда для проверки того, что Красная Шапочка может попасть в точку i достаточно проверить условие $s - i \geqslant 0$, т.к. значение i будет равно количеству энергии, которое необходимо потратить на перемещение из нуля в i.

```
import sys
s = int(input()) # Сумма энергий в начале и на всех положительных кочках
n = int(input())
for i in range(1, n + 1):
    ai = int(input())
    if ai > 0:           # На этой кочке нужно остановится
        if s - i < 0:    # Количество энергии, чтобы достичь кочки i
            print(-1)    # Если s - i отрицательно, то нельзя достичь i
            sys.exit(0)
        s += ai   ;
if s - (n + 1) < 0:      # Энергия, необходимая для достижения поляны
    print(-1)            # Если она отрицательна, то нельзя достичь поляны
else:
    print(s - (n + 1))   # иначе выводим ответ
```

Задача 4. Деление шоколадки

Должен получиться большой кусок и ещё $k - 1$ маленьких кусочков, поэтому размер большого куска будет не более, чем $mn - k + 1$.

Чтобы набрать 60 баллов можно перебирать размеры большого куска $a \times b$, при этом $1 \leqslant a \leqslant m$, $1 \leqslant b \leqslant n$. Проверим, что $ab \leqslant mn - k + 1$ и запомним наибольшее подходящее значение ab.

Такое решение будет иметь сложность $O(mn)$. Пример такого решения.

```
m = int(input())
n = int(input())
k = int(input())
```

```
ans = 1
for a in range(1, m + 1):
    for b in range(1, n + 1):
        if a * b <= m * n - k + 1:
            ans = max(ans, a * b)
print(ans)
```

Чтобы набрать 100 баллов, необходимо избавиться от одного из циклов. Заметим, что при фиксированном a значение b, при котором площадь прямоугольного куска будет наибольшей, но не превосходящей $mn-k+1$ можно получить, взяв целую часть от деления $mn-k+1$ на a. Необходимо только учесть, что значение b не может превышать n, поэтому возьмём в качестве наибольшего подходящего b минимум из значений b и (m * n - k + 1) // a. Такое решение будет иметь сложность $O(m)$. Также допустимо перебирать значение длины другой стороны за $O(n)$ или взять наименьшую из двух сторон n или m.

```
m = int(input())
n = int(input())
k = int(input())
ans = 1
for a in range(1, m + 1):
    b = min(n, (m * n - k + 1) // a)
    ans = max(ans, a * b)
print(ans)
```

Мы получили наибольший по площади целочисленный прямоугольник, площадь которого не превосходит $mn - k + 1$, помещающийся внутри прямоугольника $m \times n$. Осталось доказать, что такой прямоугольник является ответом на задачу, то есть его и ещё $k - 1$ кусков можно получить разламыванием прямоугольника $m \times n$.

Рассмотрим разные значения k. При $k = 1$ кусок всего один, его площадь не превосходит mn, ответом является само значение mn и такой прямоугольник мы получим, не делая разломов.

При $k \geqslant 3$ получить большой прямоугольник можно двумя разломами — вдоль каждой из сторон шоколадки. Мы получим нужный прямоугольник и ещё два куска. Если нам необходимо получить больше двух дополнительных кусков, то есть при $k > 3$, то станем разламывать меньшие куски на части. Один дополнительный разлом увеличивает число кусков на 1. Куски удастся разламывать до тех пор, пока каждый из них не будет состоять из одной дольки, поэтому всегда можно получить нужное количество частей.

Наконец, при $k = 2$ шоколадку нужно разломить на две части, сделав одну из частей как можно больше. Отломим от целой шоколадки полоску $1 \times m$ или $1 \times n$, в зависимости от того, какое из значений m или n меньше. Тогда большой кусок будет иметь размер $(m-1) \times n$ или $m \times (n-1)$. Но именно это и есть максимальный целочисленный прямоугольник, который получится разместить в прямоугольнике $m \times n$, но имеющий меньшую площадь, то есть и в этом случае приведённое решение даст правильный ответ.

Задача 5. Кодовый замок

Общее количество возможных кодов равно b^n, так как каждая из n цифр может принимать b различных значений.

В первой подгруппе все маски цифр имеют специфический вид, они содержат ровно одну единицу. То есть каждое полученное донесение задаёт фиксированное значение для одной цифры кода. Поскольку все маски различны, каждое ограничение сокращает количество подходящих кодов в b раз, то есть при наличии k ограничений количество подходящих кодов сократится в b^k раз и станет равно b^{n-k}. И такое простое решение набирает 24 балла.

```
b = int(input())
n = int(input())
t = int(input())
```

```
print(b ** (n - t))
```

Но это решение проходит не все тесты первой группы, потому что не учитывает тот случай, когда какое-то ограничение делает все коды невозможными. Учитывая, что каждая маска состоит из одной единицы, это возможно в ситуации, при которой число в донесении больше, чем возможное значение одной цифры, то есть больше или равно b. Достаточно проверить это условие и если оно выполняется, то ответ будет равен 0. Такое решение проходит все тесты первой группы.

```
b = int(input())
n = int(input())
t = int(input())
ans = b ** n
for i in range(t):
    mask = input()
    s = int(input())
    if s >= b:
        ans = 0
    else:
        ans //= b
print(ans)
```

Дальнейшее продвижение в этой задаче связано с перебором всех возможных кодов и проверкой поступивших донесений. Если рассматривать код, как запись числа в системе счисления с основанием b, то значение такого числа может быть от 0 до b^n (не включая верхнюю границу). Будем хранить коды в виде целых чисел, при этом цифры кода мы можем получить при помощи алгоритма перевода числа в систему счисления с основанием b, то есть делением на b в цикле n раз. Затем проверим все имеющиеся донесения, для каждого из которых переберём все возможные коды, построим представление числа в системе счисления с основанием b, посчитаем сумму цифр кода, стоящих на указанных в донесении позициях и если эта сумма не соответствует донесению, то пометим код, как недопустимый. В конце перебора выведем количество допустимых кодов.

Пример такого решения. В списке good_code размера b^n хранится для каждого кода число 1, если этот код соответствует всем донесениями, или число 0 в противном случае.

```
b = int(input())
n = int(input())
good_code = [1] * b**n
t = int(input())
for j in range(t):
    mask = input()
    sum_digits = int(input())
    for code in range(b**n):
        saved_code = code
        s = 0
        for i in range(n):
            if mask[i] == '1':
                s += code % b
            code //= b
        if s != sum_digits:
            good_code[saved_code] = 0
print(sum(good_code))
```

Но поскольку число донесений может достигать $2^n - 1$, сложность такого решения будет $b^n 2^n n$, что уже очень много для максимального случая при $b = 2$, $n = 15$. Для полного решения необходимо заметить, что большинство кодов окажется отброшено уже после проверки первых нескольких условий. Поэтому сохраним в списке только те коды, которые соответствуют всем рассмотренным

донесениям, и при анализе очередного донесения будем перебирать только оставшиеся подходящие коды. Те коды, которые удовлетворяют считанному донесению, скопируем в новый список.

Такое решение набирает 100 баллов. Пример такого решения.

```
b = int(input())
n = int(input())
codes = [i for i in range(b**n)]
t = int(input())
for j in range(t):
    mask = input()
    sum_digits = int(input())
    new_codes = []
    for code in codes:
        saved_code = code
        s = 0
        for i in range(n):
            if mask[i] == '1':
                s += code % b
            code //= b
        if s == sum_digits:
            new_codes.append(saved_code)
    codes = new_codes
print(len(codes))
```