Разбор задач

Задача 1. Долгая тренировка

Чтобы определить общую длительность всех подходов в секундах, необходимо выразить в секундах длительность одного подхода (умножив количество минут M на 60 и прибавив к этому количество секунд S) и умножить на количество подходов N.

Между подходами будет $N-1$ перерыв, каждый из которых длится P секунд, поэтому общая длительность перерывов находится по формуле $(N-1) \cdot P$.

Исходя из вышеизложенного, общая длительность тренировки в секундах составит $N \cdot (M \cdot 60 + S) + (N-1) \cdot P$. Результат можно выразить в минутах/секундах, поделив общее количество секунд на 60. Число минут будет равно целочисленной части результата деления, число секунд — остатку от деления

Описанные рассуждения запишем в виде следующего кода на языке программирования Python.

```
n = int(input())
m = int(input())
s = int(input())
p = int(input())
full_time = n * (m * 60 + s) + (n - 1) * p
print(full_time // 60)
print(full_time % 60)
```

Задача 2. Переключая каналы

Рассмотрим сначала случай $P < U$ и разберём все принципиально различающиеся случаи переключения с канала P на канал U. Ради краткости для обозначения нажатия и удерживания кнопки переключения будем использовать термин «длинное нажатие». Также используем обозначения // для целочисленного деления и % для взятия остатка от деления (как в языке Python).

Для решения задачи в случае, когда $P > U$, достаточно поменять местами значения P и U, так как все операции увеличения и уменьшения номера канала симметричны.

Есть следующие способы достижения из канала P канала U.

1. Сначала выполняем длинные, затем короткие нажатия на кнопку «+». В этом случае наилучший вариант — выполнить $(U-P)//K$ длинных нажатий и $(U-P)\%K$ коротких.

2. Сначала выполняем длинные нажатия на кнопку «+», затем короткие нажатия на кнопку «−». В этом случае нужно «перепрыгнуть» через канал U, выполнив на одно длинное нажатие больше, чем в предыдущем случае, затем вернуться назад, выполнив $K-(U-P)\%K$ коротких нажатий.

3. Сначала при помощи длинных нажатий на кнопку «−» достигнем канала номер 1, для чего понадобится $\lceil \frac{P-1}{K} \rceil$ нажатий (частное, округлённое вверх), а затем нужно, начав с канала 1, достичь канала U, что можно сделать одним из двух способов, описанных ранее.

4. Сначала при помощи длинных нажатий на кнопку «+» достигнем канала номер N, для чего понадобится $\lceil \frac{N-P}{K} \rceil$ нажатий (частное, округлённое вверх), а затем нужно, начав с канала N, достичь канала U, что можно сделать двумя возможными способами.

Пример решения на языке Python. В этом решении функция `solve` возвращает ответ для первых двух случаев, меняя местами p и u в случае $p > u$. Далее в основной программе рассматриваются все варианты, при этом результат для случаев 1 и 2 находится вызовом `solve(p, u)`, третий случай — `(p - 1 + k - 1) // k + solve(1, u)`, четвёртый случай — `(n - p + k - 1) // k + solve(n, u)`. Из всех полученных значений выбирается наименьшее.

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

```
p = int(input())
u = int(input())

def solve(p, u):
    if p > u:
        p, u = u, p
    dist = (u - p)
    long_presses = dist // k
    ans1 = long_presses + dist % k
    ans2 = (long_presses + 1) + (k - dist % k)
    return min(ans1, ans2)

ans = min(solve(p, u),
          (p - 1 + k - 1) // k + solve(1, u),
          (n - p + k - 1) // k + solve(n, u))
print(ans)
```

Для небольших значений N задача решается с помощью алгоритма поиска в ширину или динамического программирования. Основная идея состоит в том, чтобы установить связи между каналами: два канала считаются связанными, если их отделяет друг от друга ровно одно нажатие — длинное или короткое (де-факто можно говорить о построении графа). Далее, выбрав в качестве стартовой точки канал P, следует пройти по связям, находя кратчайшие пути до каждого канала (в том числе и до канала U). Однако такие решения для больших N потребуют слишком больших затрат времени и памяти.

Пример решения, использующего алгоритм обхода графа в ширину (BFS):

```
n = int(input())
k = int(input())
s = int(input())
f = int(input())
INF = n + 1
dist = [INF] * (n + 1)
dist[s] = 0
q = [s]
while dist[f] == INF:
    u = q.pop(0)
    for v in (u + 1, u - 1, u + k, u - k):
        if v < 1:
            v = 1
        if v > n:
            v = n
        if dist[v] == INF:
            dist[v] = dist[u] + 1
            q.append(v)
print(dist[f])
```

Задача 3. Кладоискатель

Рассмотрим решение первой подзадачи, когда входные данные не превосходят 100. Из условия следует, что искомое значение x находится в следующих границах: его минимальное значение не меньше наименьшего из чисел: $-r_1$ и $a - r_2$, а максимальное значение не больше наибольшего из чисел r_1 и $a + r_2$.

Глубина залегания монетки может принимать значения от 0 до наименьшего из r_1 и r_2.

Переберём все точки плоскости (x, y) в этих границах и найдём, для какой из точек выполнятся оба условия $x^2 + y^2 = r_1^2$ и $(a - x)^2 + y^2 = r_2^2$ (они являются уравнениями двух окружностей).

Пример такого решения.

```
a = int(input())
r1 = int(input())
r2 = int(input())
min_x = min(-r1, a - r2)
max_x = max(r1, a + r2)
for x in range (min_x, max_x + 1):
    for y in range(0, -max(r1, r2), -1):
        if x ** 2 + y ** 2 == r1 ** 2 and (x - a) ** 2 + y ** 2 == r2 ** 2:
            print(x)
            print(y)
```

Для решения второй подзадачи (входные числа не превосходят 10^5) избавимся от вложенного цикла по значению y. По-прежнему перебираем x в пределах, описанных в решении для первой подзадачи.

Для выбранного x значение y^2 равно $r_1^2 - x^2$ с одной стороны и $r_2^2 - (a-x)^2$ с другой стороны. Если эти два значения совпали — мы нашли подходящее решение, в качестве y нужно взять $y = -\sqrt{r_1^2 - x^2}$. Пример такого решения.

```
a = int(input())
r1 = int(input())
r2 = int(input())
x_min = min(-r1, a - r2)
x_max = max(r1, a + r2)
for x in range(x_min, x_max):
    y = r1 ** 2 - x ** 2
    if r2 ** 2 - (a - x) ** 2 == y:
        print(x)
        print(-int(y ** 0.5))
```

Полное решение имеет сложность $O(1)$. Пусть (x, y) – искомые координаты.

Из системы уравнений

$$\begin{cases} x^2 + y^2 = r_1^2, \\ (a-x)^2 + y^2 = r_2^2 \end{cases}$$

следует, что (выразим y^2 из первого уравнения и подставим во второе)

$$(a-x)^2 + r_1^2 - x^2 = r_2^2,$$

$$2ax = r_1^2 - r_2^2 + a^2,$$

откуда

$$x = \frac{r_1^2 - r_2^2 + a^2}{2a}.$$

Теперь выразим y:

$$y = -\sqrt{r_1^2 - x^2}.$$

Пример такого решения.

```
a = int(input())
r1 = int(input())
r2 = int(input())
```

```
x = (r1 ** 2 - r2 ** 2 + a ** 2) // (2 * a)
y = -int((r1 ** 2 - x ** 2) ** 0.5)
print(x)
print(y)
```

Задача 4. Обработка заявлений

Для $n \leqslant 1000$ или $n \leqslant 10^5$ задача решается простым моделированием сложности $O(n^2)$ и $O(n)$ соответственно. Нужно создать список из всех заявлений, затем в цикле с первым заявлением из списка выполнять одну из трёх операций. Отличие решений по сложности заключается в том, как реализовать удаление первого элемента из списка. Если в языке Python для этого использовать метод `pop(0)`, то одно такое удаление будет выполняться за $O(n)$, а общая сложность будет $O(n^2)$. Для уменьшения сложности до $O(n)$ нужно использовать структуру данных «очередь» или «дек» или реализовать «ленивое удаление», то есть не удалять элемент, а просто увеличивать на 1 индекс элемента, который является первым в списке.

Пример решения сложности $O(n)$ с «ленивым удалением».

```
n = int(input())
k = int(input())
a = [i + 1 for i in range(n)]
i = 0
while i < len(a):
    if a[i] == k and i % 3 == 0:
        print("Yes")
        print(i + 1)
        break
    elif a[i] == k and i % 3 == 1:
        print("No")
        print(i + 1)
        break
    elif i % 3 == 2:
        a.append(a[i])
    i += 1
```

Идея полного решения заключается в том, что примерно для $n/3$ заявлений мы сразу же можем дать ответ, что данное заявление будет подписано (и это случится на k-м шаге), ещё примерно $n/3$ заявлений будет отброшено, и примерно $n/3$ заявлений будет переложено, в этом случае нужно решить задачу для нового n, уменьшенного в три раза. Причём «моделирование» обработки всей стопки заявлений можно делать за $O(1)$ операций. Нужно только аккуратно разобраться, как происходит сведение задачи от n к $n/3$.

Рассмотрим задачу в более общем виде — дана тройка (n, k, op), где:

- n — количество заявлений,
- k — порядковый номер искомого заявления,
- op — какую *последнюю* операцию мы выполнили над заявлением (0 — переложить вниз стопки, 1 — подписать, 2 — отбросить). Эти операции повторяются по циклу: 1, 2, 0, 1, 2, 0 и т.д. Поскольку мы начинаем обрабатывать заявления с операции 1, то можно считать, что последней выполненной операцией до этого была операция 0.

Определим, какая операция будет производиться над k-м по порядку заявлением. Это $(op + k) \bmod 3$ (если последней была выполнена операция op, то с первым заявлением в стопке будет выполнена операция $op + 1$, затем — $op + 2$ и т.д. с учётом зацикливания). Если значение $(op + k) \bmod 3$ равно 1 или 2, то добавляем к ответу k, выводим ответ и завершаем программу. Если

же $(op+k) \bmod 3 = 0$, то добавим n к количеству шагов и перейдём к такой же точно задаче, только меньшего размера. Для этого нужно определить новые значения n, k и op.

Вычислим новое n — то есть сколько заявлений переместится вниз стопки.

Если $op = 0$, то переместятся $\lfloor n/3 \rfloor$ заявлений — мы перекладываем каждое третье заявление, то есть нам нужно найти количество нулей в последовательности 1, 2, 0, 1, 2, 0, ..., из n элементов.

Если $op = 1$, то нам также достаточно посчитать количество нулей в последовательности 2, 0, 1, 2, 0, 1, ..., это такая же последовательность, но сдвинутая на 1, поэтому ответ будет равен $\lfloor (n+1)/3 \rfloor$.

Если $op = 2$, то тогда нужно посчитать количество нулей в последовательности 0, 1, 2, 0, 1, 2, ..., это такая же последовательность, но сдвинутая на 2, поэтому ответ будет равен $\lfloor (n+2)/3 \rfloor$.

Заметим, что все три варианта можно записать одной формулой: $n' = (n + op)//3$.

Вычислим новое k — то есть каким по порядку будет искомое заявление после рассмотрения всех заявлений и перекладывания части из них вниз стопки. Другими словами, нужно найти количество заявлений с порядковыми номерами до k (включительно), которые переместятся вниз. Это значение также зависит от op (с какого заявления началась обработка). По аналогии с предыдущими рассуждениями получаем: $k' = (k + op)//3$.

Наконец, вычислим новое op — то есть какая операция была выполнена над последним обработанным заявлением: $op' = (op + n) \bmod 3$.

В итоге мы пришли к аналогичной задаче с параметрами (n', k', op'), для решения которой снова повторяем вышеописанные действия.

Поскольку каждый раз остаётся примерно треть заявлений от предыдущего количества, то сложность решения $O(\log n)$.

Пример решения сложности $O(\log n)$.

```
BOTTOM, SIGN, DROP = 0, 1, 2
n = int(input())
k = int(input())
op = BOTTOM
steps = 0
while (op + k) % 3 == BOTTOM:
    steps += n
    n, k, op = (n + op) // 3, (k + op) // 3, (n + op) % 3
print("Yes" if (op + k) % 3 == SIGN else "No")
steps += k
print(steps)
```

Задача 5. Осторожно, злые числа!

Для решения подзадачи $n \leqslant 10^5$ переберём все числа от 1 до n и проверим выполнение двух условий (чётность и количество единиц в двоичной записи). Если оба условия выполнены – увеличим ответ на 1.

```
n = int(input())
ans = 0
for i in range(1, n + 1):
    if i % 2 == 0 and bin(i).count('1') % 2 == 0:
        ans += 1
print(ans)
```

Для решения второй подзадачи, когда n – степень двойки, можно заметить (например, проверив маленькие степени двойки), что между 2^p и $2^{p+1} - 1$ у нас получается ровно 2^{p-2} очень злых чисел.

Этот факт доказуем. Представим все числа из интервала $[2^p, 2^{p+1} - 1]$ в двоичной системе счисления. Все они имеют одинаковую длину $p + 1$, у всех чисел первая цифра 1, у всех очень злых чисел последняя цифра будет 0. Тогда мы должны найти количество способов расставить нечетное количество единиц (одна уже стоит в самом начале числа) среди $p - 1$ позиций. Возникает ком-

бинаторная формула суммы числа сочетаний $\sum_{i=1}^{p-1} C_{p-1}^{i}$, где i принимает все нечётные значения, не превышающие $p-1$.

По свойствам сумм числа сочетаний биномиальных коэффициентов и их знакопеременной сумме можно доказать, что это число равно 2^{p-2}.

Заметим, что само число $n = 2^p$ злым никогда не будет (в его двоичном представлении одна единица на первой позиции). Для ответа на вопрос второй подзадачи задачи для $n = 2^p$ нам останется сложить все степени двойки до $p-3$ включительно. Эта сумма равна $2^0+2^1+2^2+...+2^{p-3} = 2^{p-2}-1$.

Пример решения в этом случае.

```
n = int(input())
ans = 0
p = 0
while n % 2 == 0:
    n //= 2
    p += 1
if n == 1:
    ans = 2 ** (p - 2) - 1
print(ans)
```

Полное решение: рассмотрим четвёрки последовательных целых чисел: (0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11) и т.д. Числа в этой четвёрке в двоичной системе счисления оканчиваются цифрами 00, 01, 10 и 11. Из них два нечётных числа не подходят, а два оставшихся чётных числа в двоичной записи отличаются ровно одной предпоследней цифрой, поэтому среди этих чисел ровно одно будет очень злым.

Исключением является только первая четвёрка, в которой чётными числом, содержащим чётное число единиц в двоичной записи, является число 0, однако, оно не подходит, потому что очень злое число должно быть положительным.

Таким образом, для получения ответа нужно посчитать полное число четвёрок в числе n, кроме первой четвёрки, а оставшиеся числа, которые не попали в полную четвёрку, перебрать и для каждого из них непосредственно проверить нужные условия.

Пример решения на языке Python.

```
n = int(input())
ans = (n - 4) // 4
for i in range(n // 4 * 4, n + 1):
    if i % 2 == 0 and bin(i).count('1') % 2 == 0:
        ans += 1
print(ans)
```