Разбор задач

Задача 1. Доставка в Простоквашино

Сначала рассмотрим переборное решение. Будем перебирать возможное начальное количество энергии в переменной ans, начиная с нуля. Для каждого значения проверим, сможет ли робот доехать до дома дяди Фёдора. Для этого будем моделировать его движение: сначала уменьшим запас энергии на a, а если он не стал отрицательным, увеличим его на b. Если робот смог совершить все k поездок между соседними зданиями, то текущее значение ans и будет ответом.

Пример такого решения.

```
a = int(input())
b = int(input())
k = int(input())

ans = 0
while True:
    cc = ans
    for i in range(k):
        cc -= a
        if cc < 0:
            break
        cc += b
    else:
        print(ans)
        break
    ans += 1
```

Это решение перебирает все возможные значения ответа и для каждого из них моделирует движение робота. Если ответ равен x, то время работы составляет $O(xk)$. В худшем случае это слишком долго, поэтому попробуем улучшить алгоритм.

Заметим, что для проверки начального заряда не обязательно моделировать движение робота отдельно для каждого возможного ответа. Вместо этого можно один раз проследить за изменением его энергии, считая, что изначально она равна нулю. При этом в ходе передвижения робота значение его энергии может стать отрицательным, разрешим роботу передвигаться с отрицательной энергией, запоминая минимальное (отрицательное) значение энергии, которое было у робота в ходе поездки.

Введём переменную cnt, которая будет показывать, как изменился запас энергии относительно начального значения. При каждом перемещении робота на расстояние между соседними зданиями будем уменьшать cnt на a, а затем запоминать минимальное значение этой переменной в opt. После этого увеличим cnt на b, поскольку робот может воспользоваться зарядной станцией у достигнутого дома.

Начальный заряд должен быть не меньше модуля минимального значения opt. Действительно, если в некоторый момент значение cnt равно $-x$, то для сохранения неотрицательного запаса энергии к этому моменту необходимо было изначально иметь не менее x единиц энергии.

Пример такого решения.

```
a = int(input())
b = int(input())
k = int(input())

cnt = 0
opt = 0
for i in range(k):
    cnt -= a
    opt = min(opt, cnt)
    cnt += b
```

```
print(-opt)
```

В этом решении каждый шаг моделирования выполняется за $O(1)$, поэтому общая сложность составляет $O(k)$. Такое решение проходит тесты с ограничением $k \leqslant 5 \cdot 10^5$, но для полного решения при $k \leqslant 10^9$ его недостаточно.

Рассмотрим теперь решение за $O(1)$.

Если $b \geqslant a$, то после подзарядки запас энергии робота увеличивается или остаётся прежним. Поэтому достаточно изначально зарядить робота на a единиц энергии: он сможет доехать до первого дома, а затем после каждой подзарядки у него будет достаточно энергии для следующего перемещения. Если $a = 0$, робот может добраться до любого дома без начального заряда, что также соответствует этой формуле.

Теперь рассмотрим случай $b < a$. После каждого перемещения и последующей подзарядки запас энергии уменьшается на $a - b$ единиц, то есть заряд робота будет уменьшаться при передвижении к последующему дому. Нам нужно так зарядить робота в начале, чтобы он смог доехать до последнего дома, значит, нужно рассмотреть случай, когда робот доезжает до последнего дома с нулевым зарядом. Суммарно на передвижение ему понадобится ka единиц энергии, но при этом он $k - 1$ раз зарядится на b единиц энергии. Если его первоначальный заряд был равен x, то тогда $x - ka + (k - 1)b = 0$, откуда $x = ka - (k - 1)b$.

Итак, ответ можно вычислить по формуле

$$\text{ans} = \begin{cases} a, & b \geqslant a, \\ ka - (k-1)b, & b < a. \end{cases}$$

Пример полного решения.

```
a = int(input())
b = int(input())
k = int(input())

if b >= a:
    print(a)
else:
    print(k * a - (k - 1) * b)
```

Это решение использует постоянное число арифметических операций, поэтому его временная сложность составляет $O(1)$, а дополнительная память — $O(1)$. По условию $ak \leqslant 2 \cdot 10^9$, поэтому вычисления укладываются в заданные ограничения и решение набирает 100 баллов.

Задача 2. Места в автобусе

В каждом ряду находится четыре места, причём проход разделяет их на две пары. Номера мест в первом ряду — 1, 2, 3, 4, во втором — 5, 6, 7, 8 и так далее. Также заметим, что у прохода находятся места с чётными номерами, а места с нечётными номерами находятся у окна.

Будем считать, что ряды нумеруются с нуля. Тогда номер ряда, в котором находится место с номером x, можно вычислить по формуле (x − 1) // 4, где // — целочисленное деление. Аналогично вычислим номер ряда для места y.

В начале выполнения программы удобно сделать так, чтобы $x < y$. Если изначально $x > y$, поменяем значения переменных местами.

Теперь рассмотрим все четыре категории расположения мест.

1. Идеально удобно.

В этом случае друзья сидят рядом, с одной стороны от прохода. Такие места имеют номера 1 и 2, 5 и 6, 9 и 10 и так далее либо 3 и 4, 7 и 8, 11 и 12 и так далее.

Поскольку $x < y$, для первой пары мест в ряду номер x должен быть нечётным, а номер y должен равняться $x + 1$. Кроме того, оба места должны находиться в одном ряду.

2. Довольно удобно.

В этом случае друзья сидят в одном ряду по разные стороны от прохода, причём оба места находятся у прохода. У прохода находятся места с чётными номерами, поэтому оба значения x и y должны быть чётными, а также места должны находиться в одном ряду.

3. Можно играть.

В этом случае друзья сидят на соседних рядах с одной и той же стороны от прохода. Например, места 1 и 5, а также места 4 и 8.

В этом случае нам неважно, находятся ли места у прохода или у окна, они только должны находиться в соседних рядах и по одну сторону от прохода. Рассмотрим места с номерами x + x % 2 и y + y % 2, где % — операция взятия остатка от деления. В результате преобразования нечётный номер места сменится на номер соседнего с ним места у прохода. То есть места x + x % 2 и y + y % 2 будут находиться оба у прохода, и они будут либо совпадать с местами x и y, либо будут соседними с ними.

Два места у прохода будут находиться друг за другом (то есть в соседних рядах по одну сторону от прохода), если их номера отличаются на 4. То есть в третьем случае нужно проверить условие x + x % 2 + 4 == y + y % 2.

Если ни одно из перечисленных условий не выполняется, расположение мест относится к четвёртой категории.

Полное решение выглядит следующим образом.

```
x = int(input())
y = int(input())
if x > y:
    x, y = y, x
row1 = (x - 1) // 4
row2 = (y - 1) // 4
if row1 == row2 and x % 2 == 1 and y == x + 1:
    print(1)
elif row1 == row2 and x % 2 == 0 and y % 2 == 0:
    print(2)
elif x + x % 2 + 4 == y + y % 2:
    print(3)
else:
    print(4)
```

Задача 3. Три Кирилла и их рост

Сначала рассмотрим решение полным перебором. Будем перебирать все возможные значения роста первого Кирилла a, второго Кирилла b и третьего Кирилла c. При этом сразу учтём условие $a \leqslant b \leqslant c$, чтобы не рассматривать лишние варианты. Если сумма ростов равна n, увеличим счётчик ответа.

```
n = int(input())
ans = 0
for a in range(1, n + 1):
    for b in range(a, n + 1):
        for c in range(b, n + 1):
            if a + b + c == n:
                ans += 1
print(ans)
```

Каждый из трёх циклов может выполнить до n итераций. Следовательно, временная сложность решения составляет $O(n^3)$. При больших значениях n такое решение работает слишком долго.

Попробуем избавиться от одного цикла. Если мы уже выбрали рост первого Кирилла a и рост второго Кирилла b, то рост третьего Кирилла однозначно определяется условием на сумму $c = n - a - b$.

Остаётся убедиться в том, то рост третьего Кирилла не меньше роста второго, то есть $c \geqslant b$. Если это условие выполняется, найден ещё один подходящий вариант.

```
n = int(input())
ans = 0
for a in range(1, n + 1):
    for b in range(a, n + 1):
        c = n - a - b
        if c >= b:
            ans += 1
print(ans)
```

Теперь используются только два вложенных цикла, поэтому временная сложность составляет $O(n^2)$. Это решение быстрее предыдущего, но при $n \leqslant 10^9$ его также недостаточно.

Рассмотрим следующую оптимизацию. Заметим, что при фиксированном росте второго Кирилла b можно непосредственно посчитать количество подходящих значений роста первого Кирилла a.

Из условия монотонности ростов следует, что $a \leqslant b$.

Кроме того, поскольку $c = n - a - b$ и $c \geqslant b$, получаем $n - a - b \geqslant b$, откуда $a \leqslant n - 2b$.

Таким образом, рост первого Кирилла может принимать любое целое значение от 1 до $\min(b, n - 2b)$.

Если эта величина положительна, количество подходящих значений равно ей. Если же она неположительна, подходящих значений нет. Поэтому для каждого значения b можно увеличить ответ на $\max(0, \min(b, n - 2b))$.

Получаем следующее решение.

```
n = int(input())
ans = 0
for b in range(1, n):
    ans += max(0, min(b, n - 2 * b))
print(ans)
```

В этом решении один цикл, поэтому временная сложность составляет $O(n)$. Такое решение проходит тесты с ограничением $n \leqslant 10^6$, но для полного решения необходимо избавиться и от цикла.

Для этого заметим, что слагаемые в полученной сумме имеют достаточно простой вид.

Если $b \leqslant \lfloor n/3 \rfloor$, то $n - 2b \geqslant b$, поэтому количество подходящих значений роста первого Кирилла равно b. Обозначим $m = \left\lfloor \frac{n}{3} \right\rfloor$.

Сумма значений всех b от 1 до m равна $1 + 2 + \ldots + m = \frac{m(m+1)}{2}$.

Осталось рассмотреть значения $b > m$. Для них количество подходящих значений равно $n - 2b$, так как в этом случае $n - 2b < b$, если эта величина положительна, и нулю в противном случае.

При увеличении b на единицу величина $n - 2b$ уменьшается на два. Следовательно, оставшиеся положительные слагаемые образуют последовательность чисел одинаковой чётности, начиная с $k = n - 2(m + 1)$. Необходимо складывать числа k, $k - 2$, $k - 4$, ... до тех пор, пока они остаются положительными.

Количество членов этой арифметической прогрессии равно $k/2$, округлённому вверх, а последний член этой арифметической прогрессии имеет такую же чётность, что и число k, поэтому он равен $2 - (k \bmod 2)$. Сумма прогрессии может быть вычислена по формуле $\frac{(k+2-(k \bmod 2))\lfloor (k+1)/2 \rfloor}{2}$.

Таким образом, можно вычислить ответ без перебора всех значений b.

```
n = int(input())
m = n // 3
ans = m * (m + 1) // 2
m += 1
k = n - 2 * m
```

```
ans += (k + 2 - k % 2) * ((k + 1) // 2) // 2
print(ans)
```

Сложность такого решения составляет $O(1)$.

Задача 4. Очередь на заправку

30 баллов можно получить, моделируя процесс поминутно. Сложность такого решения $O(|t_1| + |t_2| + \ldots + |t_n|)$, и оно не представляет интереса, поэтому сразу перейдём к полному решению.

Будем последовательно рассматривать машины в порядке их расположения в очереди. Для каждой машины необходимо определить, когда она сможет начать заправку.

Поскольку на заправке две колонки, достаточно хранить временную отметку момента освобождения каждой из них. Будем использовать для этого список free из двух элементов. Изначально обе колонки свободны, поэтому оба элемента равны нулю. Колонки будем нумеровать с нуля, как и элемент списка.

Если время заправки очередной машины положительно, то её горловина расположена справа, и она должна воспользоваться колонкой 0. Если время отрицательно, то горловина расположена слева, и машине нужна колонка 1. Для удобства будем хранить номер подходящей колонки в переменной i, а продолжительность заправки вычислять как модуль числа t.

Пусть curr_time — время, когда предыдущая машина начала заправку. Очередная машина не может начать заправку раньше этого момента. Кроме того, выбранная колонка должна быть свободна. Поэтому время начала заправки равно максимуму из curr_time и времени освобождения соответствующей колонки: curr_time= max(curr_time, free[i]).

После этого обновим время освобождения колонки. Если машина заправляется $|t|$ минут, то колонка освободится через $|t|$ минут после начала заправки: free[i] = curr_time + |t|.

Таким образом, для каждой машины достаточно выполнить несколько операций. После обработки всех машин ответом будет максимальное время освобождения двух колонок, поскольку необходимо дождаться окончания заправки всех машин.

Полное решение выглядит следующим образом.

```
n = int(input())
free = [0, 0]
curr_time = 0
for _ in range(n):
    t = int(input())
    if t > 0:
        i = 0
    else:
        i = 1
    curr_time = max(curr_time, free[i])
    free[i] = curr_time + abs(t)
print(max(free))
```

Каждая машина обрабатывается ровно один раз, причём на её обработку требуется постоянное число операций. Поэтому временная сложность алгоритма составляет $O(n)$.

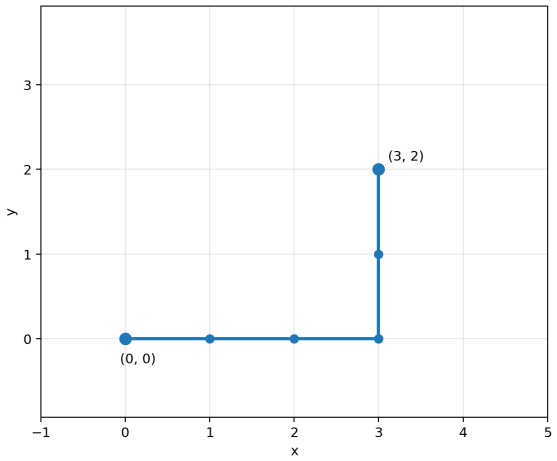
Задача 5. Мост из отрезков

Задачу можно сформулировать по-другому: необходимо построить несамопересекающийся маршрут длины n из точки $(0;0)$ в точку $(x;y)$, проходящий по линиям целочисленной сетки. Каждый отрезок маршрута соединяет две соседние целочисленные точки и имеет единичную длину. Поэтому за один шаг ровно одна координата меняется на 1 или -1.

Чтобы добраться из $(0,0)$ в (x,y), необходимо сделать не менее $|x|$ горизонтальных и $|y|$ вертикальных шагов. Следовательно, минимальное число отрезков равно $|x| + |y|$. Если $n < |x| + |y|$, построить мост невозможно.

Кроме того, каждый шаг меняет чётность суммы координат. Поэтому чётность суммы координат конечной точки должна совпадать с чётностью числа n. Если это условие не выполняется, мост также построить нельзя. Именно эти две проверки выполняются в начале программы.

Если $n = |x| + |y|$, то мост необходимо построить по кратчайшему пути. Достаточно пройти $|x|$ шагов по горизонтали в сторону координаты x, а затем $|y|$ шагов по вертикали в сторону координаты y. Пример такого пути показан на рисунке.

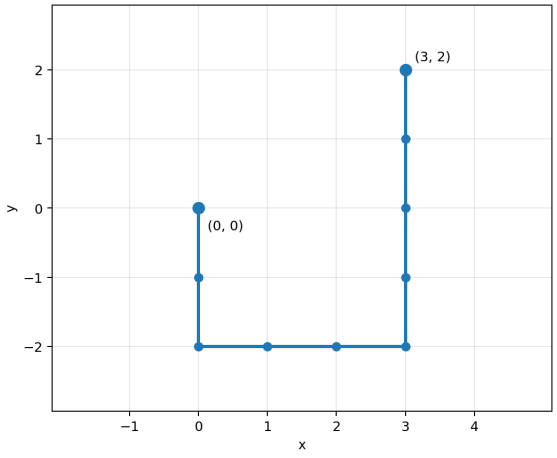


Если $n > |x| + |y|$, то необходимо добавить к кратчайшему пути дополнительные отрезки. Поскольку чётность n и $x + y$ должна совпадать, разность $n - |x| - |y|$ чётна. Значит, можно добавить отрезки парами, количество пар равно $\text{extra} = (n - |x| - |y|)/2$.

Добавлять дополнительные отрезки будем так. Сначала выполняются extra шагов в направлении, противоположном вертикальному направлению к конечной точке. Затем выполняются все горизонтальные шаги в направлении x, после чего выполняются $\text{extra} + |y|$ вертикальных шагов в направлении конечной точки.

Первые дополнительные шаги компенсируются последними: на вертикальном участке путь сначала отходит от конечной точки, а затем возвращается. Это позволяет использовать все n отрезков, при этом получится несамопересекающаяся ломаная.

На рисунке показано построение для $n = 9$, $x = 3$, $y = 2$. Здесь $\text{extra} = \frac{9-3-2}{2} = 2$.



В примере решения функция sign возвращает 1, если аргумент неотрицателен, и -1 в противном случае. Значение функции sign соответствует движению в направлении точки, а для движения в противоположном направлении необходимо взять противоположное значение. Переменные dir1x и dir1y задают направление горизонтальных шагов, а dir2x и dir2y — направление вертикальных шагов. Количества шагов в этих направлениях хранятся в cnt1 и cnt2.

Затем программа выводит начальную точку $(0, 0)$ и координаты каждой следующей вершины:

1. выполняет extra шагов против направления второго участка;
2. выполняет cnt1 шагов в направлении первого участка;
3. выполняет extra + cnt2 шагов в направлении второго участка.

Но если горизонтального участка нет, то первоначальное движение в противоположном направлении вдоль оси Y и последующее движение к конечной точке пересекутся. Это произойдёт, если $x = 0$. Вместо того чтобы отдельно рассматривать этот случай, просто поменяем горизонтальные и вертикальные участки местами, то есть сначала будем передвигаться вдоль оси X, затем — вдоль оси Y на ненулевое значение (потому что числа x и y не могут быть равны 0 одновременно), затем снова вдоль оси X в другом направлении.

Пример решения. Это решение работает и в случае $n = |x| + |y|$, так как тогда extra $= 0$ и дополнительные шаги не совершаются.

```
import sys

n = int(input())
x = int(input())
y = int(input())

if (x + y) % 2 != n % 2 or n < abs(x) + abs(y):
    print(-1)
    sys.exit(0)

def sign(a):
    if a >= 0:
        return 1
    else:
        return -1

dir1x = sign(x)
dir1y = 0
cnt1 = abs(x)
dir2x = 0
dir2y = sign(y)
cnt2 = abs(y)

if x == 0:
    dir1x, dir2x = dir2x, dir1x
    dir1y, dir2y = dir2y, dir1y
    cnt1, cnt2 = cnt2, cnt1

extra = (n - abs(x) - abs(y)) // 2

x = 0
y = 0
print(x, y)
for i in range(extra):
    x -= dir2x
    y -= dir2y
    print(x, y)
for i in range(cnt1):
    x += dir1x
    y += dir1y
```

```
    print(x, y)
for i in range(extra + cnt2):
    x += dir2x
    y += dir2y
    print(x, y)
```