

Разбор задач

Задача 1. Лёша-путешественник

В этой задаче нужно заметить, что каждое из двух условий (что перед Лёшей находится не менее A вагонов, а за ним — не более B вагонов) задаёт суффикс вагонов, в которых может находиться Лёша. Из первого условия следует, что Лёша должен находиться в $N - A$ последних вагонах, а второе условие говорит, что Лёша должен находиться в $B + 1$ последних вагонах. Таким образом, ответом является $\min(N - A, B + 1)$.

```
n = int(input())
a = int(input())
b = int(input())
ans = min(n - a, b + 1)
print(ans)
```

Задача 2. Ход слона

Рассмотрим решение, работающее при $R = C$. В таком случае слон обязан находиться на диагонали, соединяющей левый нижний угол с правым верхним. Тогда слон обязательно бьёт все клетки данной диагонали за исключением той клетки, на которой стоит он сам.

Также слон бьёт клетки на ещё одной диагонали, которая параллельна главной (идущей из левого верхнего угла в правый нижний). Количество клеток, которые он бьёт, тем больше, чем ближе слон к центру доски. Рассмотрим количество клеток, которые слон бьёт на этой диагонали в зависимости от его координат, при $N = 6$:

- 0
- 2
- 4
- 4
- 2
- 0

Видно, что по мере приближения к центру количество атакуемых клеток увеличивается с шагом 2. Получаем следующий вариант решения:

```
n = int(input())
r = int(input())
c = int(input())
ans = n - 1
if r <= (n + 1) // 2:
    ans += r + c - 2
else:
    ans += 2 * n - r - c
print(ans)
```

Решение, работающее при $N \leq 100$, построить несколько проще, чем предыдущее. Достаточно воспользоваться весьма простым свойством — если две клетки расположены на одной диагонали, то модуль разности между номерами их строк равен модулю разности между номерами их столбцов. Остаётся только перебрать все клетки и посчитать, для какого количества клеток справедливо данное равенство с клеткой, где стоит слон, не забыв учесть, что для вышеупомянутой клетки данное равенство тоже выполнится.

```
n = int(input())
r = int(input())
c = int(input())
ans = 0
for row in range(1, n + 1):
    for col in range(1, n + 1):
        if abs(row - r) == abs(col - c):
            ans += 1
print(ans - 1)
```

Для того чтобы получить полное решение, необходимо по координатам слона определять длины диагоналей, на которых он находится.

Рассмотрим диагонали, параллельные главной. Длина главной диагонали равна N , соседних ей диагоналей – $N - 1$, соседних соседним – $N - 2$ и так далее. Следовательно, длина таких диагоналей будет равна разнице N и расстояния до главной диагонали, что приводит нас к формуле $N - \text{abs}(R + C - 1 - N)$, где $R + C - 1$ номер диагонали.

Подобные размышления применимы и к диагоналям, которые параллельны побочной диагонали, с той лишь разницей, что номер диагонали считается по формуле $N - R + C$, а значит, итоговая формула для вычисления длины диагонали будет иметь вид $N - \text{abs}(C - R)$.

Ответом на задачу является сумма длин диагоналей, на которых лежит клетка со слоном, уменьшенная на два, чтобы учесть занятую слоном клетку. Итоговое решение на языке Python получится таким:

```
n = int(input())
r = int(input())
c = int(input())
len_diag_main = n - abs(r + c - 1 - n)
len_diag_sub = n - abs(c - r)
print(len_diag_main + len_diag_sub - 2)
```

Задача 3. Озеленение

Разобьём решение на 2 части:

1) Пусть мы зафиксировали какие-то стороны участка и их ориентацию, обозначим их как A и B . Заметим, что по стороне длины N мы можем разместить не более $N//A$ участков, а по стороне длины B – не более $M//B$. Тогда всего получится разместить не более $(N//A) * (M//B)$ участков. Заметим, что если поменять A и B местами, то формула примет вид $(N//B) * (M//A)$. Таким образом, по заданным A и B мы научились за $O(1)$ искать наибольшее количество участков, которые удастся разместить на прямоугольнике $N \times M$.

2) Теперь научимся быстро перебирать A и B . Вспомним, что $A * B = S$, заметим, что можно перебирать только A , а $B = S/A$. Теперь воспользуемся следующим фактом: если $A * B = S$, то $\min(A, B) \leq \sqrt{S}$. Достаточно перебрать только числа, которые меньше или равны \sqrt{S} , а второй делитель будет находиться однозначно.

Таким образом, мы получили решение за $O(\sqrt{S})$

```
n = int(input())
m = int(input())
s = int(input())
ans = 0
d = 1
while d * d <= s:
    if s % d == 0:
        a = d
        b = s // d
```

```
ans = max(ans, (n // a) * (m // b), (n // b) * (m // a))
d += 1
print(ans)
```

Задача 4. Заплыв

При значениях $N \leq 1000$ можно решить задачу, последовательно проверяя для каждой пары буйков i и j ($i < j$), способен ли Петя, начав заплыв от буйка i , доплыть до буйка j и вернуться назад за время, не превосходящее T . Такое решение содержит два цикла: для i от 1 до $N - 1$ и для j от $i + 1$ до N , в рамках которых будет подсчитываться суммарное расстояние между буйками i и j . Асимптотика этого решения $O(N^2)$, и тесты, содержащие большие значения N , оно не пройдёт из-за ограничений по времени.

Вместе с тем несложно заметить, что в описанном решении многократно выполняются одни и те же вычисления. Если удастся избежать этих повторений, работа программы существенно ускорится.

Можно поступить следующим образом. Для каждого буйка i подсчитаем, какое расстояние отделяет его от самого первого буйка. Ради удобства будем использовать традиционную для большинства языков программирования нумерацию буйков с 0. Результаты подсчёта сохраним в списке a . Ниже показан код на Python, который это делает.

```
a = [0] * n
for i in range(n - 1):
    a[i + 1] = a[i] + s[i]
```

Теперь, чтобы посчитать расстояние между двумя буйками с номерами i и j ($i < j$), достаточно вычислить разность $a[j] - a[i]$. При этом, если для некоторых i и j оказалось, что расстояние ещё не превышает половину T , можно попробовать изменить правую границу и рассмотреть пару с номерами i и $j + 1$.

Если же выяснилось, что расстояние уже превышает допустимое с точки зрения времени, нет необходимости проверять следующие значения j . Стоит перейти сразу к паре буйков с номерами $i + 1$ и j (меньшее количество нас не интересует, а технически проще делать именно такой переход а не, к примеру, к $i + 1$ и $j + 1$).

Разумеется, в ходе описанных вычислений следует сохранять наилучшее значение для количества буйков (ans) и соответствующие ему номера буйков (ans_l и ans_r). Код, выполняющий эти действия, приведён ниже.

```
k = t // 2
ans = 1
ans_l = 0
ans_r = 0
l = 0
r = 0
while r < n:
    if a[r] - a[l] <= k and r - l + 1 > ans:
        ans = r - l + 1
        ans_r = r
        ans_l = l
    if a[r] - a[l] <= k:
        r += 1
    else:
        l += 1
print(ans_l + 1, ans_r + 1)
```

При выводе ответа восстанавливается нумерация с единицы.

Асимптотика такого решения составляет $O(N)$, на каждом шаге увеличивается либо левая (l), либо правая (r) граница рассматриваемого отрезка из буйков.

Существуют и другие решения, отличающиеся как технически, так и идейно.

Задача 5. Перекрёсток

Эту задачу проще всего решать непосредственным подсчётом.

Для каждого из углов определим минимальное время, за которое его достигнет Петя. Если $TJ \leq GJ$, Петя может прийти на любой угол Y двумя способами.

Если же $TJ > GJ$, можно либо заменить значение RJ каким-то очень большим числом, заведомо превышающим самое большое время, которое Петя потратит на переходы (если учитывать, что никакие времена в задаче не превосходят 10^6 , подойдёт число 10^8), а также присвоить TJ какое-либо значение, меньшее или равное GJ ; можно использовать отрицательное значение TJ как маркер подобной ситуации (допустимы разные реализации).

Поскольку в задаче гарантировано, что способ перейти существует всегда, такой модифицированный переход окажется заведомо более долгим, нежели реальный.

Удобно определить функцию, которая по номеру дороги num и моменту времени $prevtime$, в который к этой дороге подошёл Петя, определяет, в какой момент он завершит переход этой дороги. Опишем, что делает эта функция, и приведём её код на Python.

Нужно узнать, удастся ли Пете начать переход дороги сразу же в момент $prevtime$ или понадобится ждать. Для этого возьмём остаток от деления rem времени $prevtime$ на время, составляющее полный цикл светофора f (суммарное время, в течение которого горят красный и зелёный сигналы). По истечении целого количества k полных циклов по оставшемуся времени узнаем цвет сигнала на светофоре. Если горит красный, то необходимо просто дождаться его окончания, если же зелёный, то либо оставшегося времени $(f - rem)$ окажется достаточно для перехода, либо придётся подождать в течение этого времени и времени, на протяжении которого будет гореть красный сигнал.

В данной реализации в случае невозможности перехода TJ заменяется отрицательным значением, а в качестве результата функции выводится число заведомо большее, чем время, необходимое на переход.

```
MYINF = 10*1000*1000
```

```
def cross_road(prevtime, num):
    f = r[num] + g[num]
    k = prevtime // f
    rem = prevtime % f
    fint = f * k
    if f - rem < t[num]:
        fint += (f + r[num])
    else:
        fint += max(r[num], rem)
    res = fint + t[num]
    if t[num] < 0:
        res = MYINF
    return res
```

Теперь, когда в нашем распоряжении имеется такая функция, можно вычислить минимальное время для каждого из углов. Удобно сохранить эти значения в словаре. Ниже приводится соответствующий фрагмент кода; для нумерации дорог используются числа от 0 до 3 (вместо нумерации от 1 до 4).

Также важно заметить, что угол 34 достигается либо как $12 - 23 - 34$, либо как $12 - 41 - 34$. Поскольку оба пути необходимы для дальнейших вычислений, мы вводим дополнительный элемент словаря для угла 43, сохраняя в нём второй путь.

```
fwd = dict()
fwd[23] = cross_road(0, 1)
```

```
fwd[41] = cross_road(0, 0)
fwd[34] = cross_road(fwd[23], 2)
fwd[43] = cross_road(fwd[41], 3)

fwd[23] = min(fwd[23], cross_road(fwd[43], 2))
fwd[41] = min(fwd[41], cross_road(fwd[34], 3))
fwd[34] = min(fwd[34], fwd[43])
```

Теперь остается только вывести в качестве ответа $fwd[y]$.

Конечно, существуют и другие способы решения этой задачи, в том числе использующие алгоритмы на графах.