Задача 1. Всё могут короли!

Первая подзадача (полный перебор). При наиболее плотной «упаковке» короли будут располагаться в левых нижних клетках квадратов 2×2, которые замостят доску, начиная с левого нижнего угла. При этом сами короли будут располагаться исключительно на полях, обе координаты которых — нечётные числа. Переберём вложенным циклом все поля доски $n \times n$ и посчитаем количество подходящих.

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

Данное решение имеет сложность $O(n^2)$.

Вторая подзадача (сокращение перебора). Заметим, что короли располагаются в квадратной сетке и что по вертикали будет столько же королей, сколько по горизонтали. Значит, можно просто возвести в квадрат это количество.

```
n = int(input())
ans = 0
for i in range(1, n + 1):
    if i % 2 == 1:
        ans += 1
ans = ans ** 2
print(ans)
```

Данное решение имеет сложность $O(n)$

Полное решение

Будем размещать королей, начиная с левого нижнего угла. Тогда их количество на нижнем крае доски будет равно $\lceil \frac{n}{2} \rceil$, где скобки означают округление вверх до целой части. Такое же количество королей окажется на левом крае доски, а значит, их общее число составит $\lceil \frac{n}{2} \rceil^2$.

```
n = int(input())
ans = ((n + 1) // 2) ** 2
print(ans)
```

Данное решение имеет сложность $O(1)$.

Задача 2. Натуральный ряд

Первая подзадача (полный перебор). Промоделируем описанную ситуацию: для каждого натурального числа установим, будет оно вычеркнуто или нет (последнее возможно, если число имеет ненулевой остаток при делении как на 2, так и на 3). Заведём счётчик подходящих чисел и будем его увеличивать, пока не наберём нужное количество.

```
n = int(input())
count = 0
number = 0
while count < n:
    number += 1
    if number % 2 > 0 and number % 3 > 0:
        count += 1
print(number)
```

Данное решение имеет сложность $O(n)$.

Полное решение: из каждых шести чисел, идущих подряд, останется два (имеющих остатки 1 и 5 при делении на 6), поэтому каждое увеличение n на 2 вызовет увеличение ответа на 6. Определим количество полных пар для числа $n-1$ и скорректируем ответ в зависимости от чётности самого n.

ans = (n - 1) // 2 * 6.

Если n нечётное, то к ответу добавим 1, иначе – 5.

```
n = int(input())
ans = (n - 1) // 2 * 6
if n % 2:
    ans += 1
else:
    ans += 5
print(ans)
```

Данное решение имеет сложность $O(n)$.

Задача 3. Городки

Первая подзадача: $n \leqslant 3$. Рассмотрим простые случаи.

При $n = 1$ есть только палочки единичной длины. Их длина и количество будет ответом, поскольку распилить такие палочки на две части нельзя (формально можно, но получатся части длиной 0 и 1).

При $n = 2$ есть палочки единичной длины и длины 2. Пусть у нас x палочек длины 1 и y длины 2. Попытаемся максимизировать количество палочек длины 1: их можно сделать $x + 2 \times y$ (добавляем распиленные пополам палочки длины 2). Попытаемся максимизировать количество палочек длины 2: их можно оставить только y. Первый результат лучше для любых неотрицательных значений. Поэтому ответом будет пара чисел $x + 2 \times y$ и 1.

При $n = 3$ рассуждения аналогичны. Пусть у нас x палочек длины 1, y длины 2 и z длины 3. Попытаемся максимизировать количество палочек длины 1 – это $x + 2 \times y + z$ (добавляем распиленные пополам палочки длины 2 и получившиеся при распиле палочек длины 3 кусочки длины 1). Попытаемся максимизировать количество палочек длины 2 – это $y + z$. Попытаемся максимизировать количество палочек длины 3: их можно оставить только z. Первый результат опять лучше для любых неотрицательных значений. Поэтому ответом будет пара чисел $x + 2 \times y + z$ и 1.

```
n = int(input())
if n == 1:
    x = int(input())
    print(x)
if n == 2:
    x = int(input())
    y = int(input())
    print(x + 2 * y)
if n == 3:
    x = int(input())
    y = int(input())
    z = int(input())
    print(x + 2 * y + z)
print(1)
```

Первая подзадача: $n \leqslant 1000$. Полный перебор. Создадим вспомогательный массив всех возможных размеров, для каждого размера переберём все возможные способы сделать конкретное значение палочек внутренним циклом. Заполнив данный массив, найдём в нём максимальный элемент.

```
n = int(input())
L = [0]
for i in range(n):
    L.append(int(input()))
```

```
M = [L[i] for i in range(n + 1)]
for i in range(1, n + 1):
    for j in range(i + 1, n + 1):
        if j % 2 == 1:
            if j // 2 == i:
                M[i] += L[j]
            if j // 2 + 1 == i:
                M[i] += L[j]
        elif j // 2 == i:
            M[i] += 2 * L[j]
best_ans = 1
best_count = M[1]
for i in range(2, n + 1):
    if M[i] > best_count:
        best_count = M[i]
        best_ans = i

print(best_count)
print(best_ans)
```

Полное решение: Сокращение перебора, поиск максимума за один проход по массиву.

Для каждого i у нас есть $L[i]$ исходных заготовок.

Если $2 \times i - 1 \leqslant n$, то есть возможность распилить все палочки длины $2 \times i - 1$ и получить дополнительно $L[2 \times i - 1]$ палочек длины i;

Если $2 \times i \leqslant n$, то есть возможность распилить все палочки длины $2 \times i$ и получить дополнительно $2 \times L[2 \times i - 1]$ палочек длины i;

Если $2 \times i + 1 \leqslant n$, то есть возможность распилить все палочки длины $2 \times i + 1$ и получить дополнительно $L[2 \times i + 1]$ палочек длины i.

Пройдём по массиву и для каждого i определим количество палочек данного размера, которые возможно получить из палочек исходной длины и палочек длин $2 \times i - 1$, $2 \times i$ и $2 \times i + 1$, выберем наибольшее из всех полученных значений.

```
n = int(input())
L = [0]

for i in range(n):
    L.append(int(input()))

best_ans = 1
best_count = L[1]
if n > 1:
    best_count += 2 * L[2]
if n > 2:
    best_count += L[3]
for i in range(2, n + 1):
    count_cur = L[i]
    if 2 * i - 1 <= n:
        count_cur += L[2 * i - 1]
    if 2 * i <= n:
        count_cur += 2 * L[2 * i]
    if 2 * i + 1 <= n:
        count_cur += L[2 * i + 1]
    if count_cur > best_count:
        best_count = count_cur
```

```
        best_ans = i

print(best_count)
print(best_ans)
```

Это же решение можно записать короче, если вместо проверки выхода за границу массива добавить в массив ещё $n + 1$ дополнительный элемент, соответствующий палочкам большей длины, присвоив им нулевые значения.

```
n = int(input())
a = [0] + [int(input()) for i in range(n)] + [0] * (n + 1)
best_ans = 1
best_count = a[1] + 2 * a[2] + a[3]
for i in range(2, n + 1):
    count = a[i] + a[2*i-1] + 2 * a[2*i] + a[2*i+1]
    if count > best_count:
        best_count = count
        best_ans = i
print(best_count)
print(best_ans)
```

Задача 4. Сон Пифагора

Неполное решение при $n \leqslant 10^5$ – моделирование ситуации (определение последней ненулевой цифры и её вычитание из числа).

```
n = int(input())
ans = 0
while n:
    m = n
    while m % 10 == 0:
        m //= 10
    n -= m % 10
    ans += 1
print(ans)
```

Полное решение:

Ответим на следующие вопросы:

1) Сколько операций нужно, чтобы обнулить последнюю цифру числа n?

Ответ: 1, если она не равна нулю, 0 в противном случае.

2) Сколько после этого операций нужно, чтобы обнулить предпоследнюю цифру числа n?

Ответ: Поскольку на уменьшение этой цифры на 1 требуется 2 операции, то значение цифры нужно умножить на 2.

3) Сколько после этого операций нужно, чтобы обнулить третью с конца цифру числа n?

Ответ: Поскольку на уменьшение этой цифры на 1 требуется 20 операций (нам придётся 10 раз уменьшить предпоследнюю цифру числа), значение цифры нужно умножить на 20.

И так далее. Чтобы обнулить k-ю цифру с конца, нам потребуется $2 \times 10^{k-1}$ операций.

Такое решение уже наберёт полный балл.

```
n = int(input())
ans = 0
if n % 10:
    ans += 1
n //= 10
p = 2
while n:
```

```
    ans += n % 10 * p
    p *= 10
    n //= 10
print(ans)
```

Но можно сделать ещё одно умозаключение и понять, что всё описанное выше – это просто удвоение числа. Поэтому алгоритм можно записать значительно короче: чтобы обнулить все цифры исходного числа, достаточно отбросить от него последнюю цифру, оставшееся число удвоить и прибавить к результату 1 (но только если отброшенная цифра не равнялась нулю).

Более строгое математическое обоснование можно провести по индукции:

Пусть дано число $\overline{a_n a_{n-1} \dots a_1 a_0}$. Разобьём процесс на этапы, в начале k-е число оканчивается на $k-1$ ноль.

Индукция: требуется $2 \times \overline{a_k a_{k-1} \dots a_1} + 1$ операций на зануление непоследней цифры и всех последующих.

База для $k = 1$ работает.

Рассмотрим a_k. После вычитания a_k все следующие цифры, кроме последней, – девятки, поэтому потребуется $1 + 2 \times 99 \dots 9 + 1 = 2 \times (10^{k-1} - 1) + 2 = 2 \times 10^{k-1}$ операций.

Проделав это a_k раз, получим как раз прибавление к общему числу $2 \times \overline{a_k 00 \dots 0}$, что и требовалось доказать.

```
n = int(input())
ans = 2 * (n // 10)
if n % 10 > 0:
    ans += 1
print(ans)
```

Задача 5. Геометрическая игра на планшете

Рассмотрим некоторые частичные решения данной задачи.

Подзадача 1. Если входные числа не превосходят 1000, то задачу можно решить квадратичным моделированием набора заполненных ячеек. Заведём массив из n элементов, каждый будет содержать заполнение очередной ячейки. Пронумеруем цвета в порядке их следования следующим образом: жёлтый — 0, зелёный — 1, красный — 2, синий — 3, и если в ячейке хранится один треугольник, её значение равно номеру этого цвета. Пустую ячейку, будем кодировать числом -10, полностью заполненную ячейку будем кодировать числом 10. Заметим, что систему кодирования можно придумать и другую.

Далее запускаем цикл, который будет на каждом шаге под номером i пытаться поместить треугольник цвета $i\%4$ в какую-нибудь ячейку. Само собой, это нужно делать, если фигурки цвета $i\%4$ ещё не закончились. Если это так, то запустим внутренний цикл по переменной j, перебирающий все ячейки. Если в ячейке номер j ещё пусто, помещаем в эту ячейку текущий треугольник цвета $i\%4$ и заканчиваем текущий шаг внешнего цикла. Если же пустых ячеек не встретили, но есть ячейка с парным текущему цветом (это можно проверить, посчитав расстояние между текущим цветом и содержимым ячейки, оно должна быть равным 2), то помещаем текущий цвет в эту ячейку, назначаем ей значение 10. При этом уменьшаем количество экземпляров текущего цвета и увеличиваем счётчик уже размещённых треугольников этого цвета.

Осталось заметить, что для полного размещения всех треугольников внешнему циклу достаточно $8 * n$ шагов. За каждые четыре последовательных шага будет заполняться хотя бы одна половина ячейки (если фигурки и места ещё не закончились). Можно показать, что эта оценка понижается до $4 * n$.

Далее приведём код на Python, моделирующий это решение. Состояние цветов будем хранить в массивах из четырёх элементов v (количество ещё не перемещённых треугольников соответствующего цвета) и ans (количество уже размещенных в ячейках треугольников соответствующего цвета). В массиве msk моделируем маску занятости ячеек.

```
v = []
```

```
for i in range (4):
    v.append(int(input()))
n = int(input())

ans = [0 for i in range(4)]
msk = [-10 for i in range(n)]

for i in range(4 * n):
    tcol = i % 4
    if v[tcol] > 0:
        pos1 = -1
        pos2 = -1
        for j in range(n):
            if msk[j] == -10:
                pos1 = j
                break
            if abs(tcol - msk[j]) == 2:
                pos2 = j
        if pos1 != -1:
            msk[pos1] = tcol
            ans[tcol] += 1
            v[tcol] -= 1
            continue
        if pos2 != -1:
            msk[pos2] = 10
            ans[tcol] += 1
            v[tcol] -= 1

for i in range(4):
    print(ans[i])
```

Это решение набирает 30 баллов.

Подзадача 2. Разобьём весь процесс раскладки на два этапа: на первом этапе раскладываем треугольники в пустые ячейки по одному и, если те ещё не закончились, на втором этапе пытаемся добавить их к парным цветам.

Если общее число треугольников не превосходит 10^6, то, независимо от числа ячеек, можно перебирать по циклу цвета и пытаться расположить их по одному в пустые ячейки во время первого этапа. При этом, как обычно, будем учитывать в четырёх соответствующих переменных (а лучше в массиве из четырех элементов) число уже размещённых и оставшихся фигурок (как и ранее, это массивы ans и v, но теперь массив ans будет хранить только количество треугольников, размещённых в пустые ячейки на первом этапе процесса).

Берём очередной цвет, и если соответствующие треугольники и пустые ячейки ещё не закончились (последние храним просто в переменной n, в которую считываем общее число ячеек), то уменьшаем количество неперемещённых и увеличиваем число перемещённых фигурок текущего цвета и уменьшаем число пустых ячеек. Удобно это оформить в виде бесконечно работающего цикла, завершающегося в случаях, когда закончатся пустые ячейки или четыре раза подряд будут отсутствовать треугольники текущего цвета.

Второй этап моделировать не будем, нам уже достаточно данных для вывода ответа. Для каждого цвета i определим парный ему $pcol = (i+2)\%4$. Тогда число размещённых в ячейках треугольников цвета i будет равно $ans[i]$ (то, что разместили на первом этапе в пустые ячейки) плюс минимум из того, что осталось в этом цвете $v[i]$ и числа размещённых на первом этапе парных текущему $ans[pcol]$.

Код этого решения на Python:

```
v = []
for i in range (4):
    v.append(int(input()))
n = int(input())

ans = [0 for i in range(4)]

tcol = -1
cnt = 0
while True:
    cnt += 1
    tcol = (tcol + 1) % 4
    if v[tcol] > 0 and n > 0:
        v[tcol] -= 1
        ans[tcol] += 1
        n -= 1
        cnt = 0
    if cnt == 4 or n == 0:
        break

for i in range(4):
    pcol = (i + 2) % 4
    print(ans[i] + min(v[i], ans[pcol]))
```

Это решение набирает 60 баллов.

Полное решение. Из предыдущего решения мы заметили, что не обязательно моделировать второй этап, если нам уже известно число размещённых треугольников тех или иных цветов на первом этапе. Для полного решения нужно научиться получать результат первого этапа также без перебора. Можно видеть, что если у нас достаточно много ячеек, то на первом этапе сначала цвета будут раскладываться по четыре различных(таких четвёрок будет d штук), далее, когда синие треугольники закончатся, — по три различных (таких троек будет $c-d$ штук), затем по два (таких пар будет $b-c$ штук) и, наконец, оставшиеся треугольники жёлтого цвета будут раскладываться друг за другом ($a-b$ штук). Однако, так как n может быть меньше суммы $d+c+b+a$, требуется выяснить, в какой момент пустые ячейки закончатся и начнётся второй этап с добавлением треугольников к парным им цветам.

Для начала подсчитаем, что будет размещено на первом этапе. Для этого введём такие переменные:

$B_1 = 4 * d$,

$B_2 = 4 * d + 3 * (c - d)$,

$B_3 = 4 * d + 3 * (c - d) + 2 * (b - c)$,

$B_4 = 4 * d + 3 * (c - d) + 2 * (b - c) + 1 * (a - b)$, обозначающие границы упомянутых событий.

Если $n \leqslant B1$, то на первом этапе будут размещены $n/4$ четвёрок цветов (целая часть от деления на 4). Для $n\%4$ первых цветов будет размещён еще один дополнительный треугольник.

Если $B_1 < n \leqslant B2$, то на первом этапе будут размещены d четвёрок цветов и $(n - B_1)/3$ троек из первых трех цветов. Для $(n - B_1)\%3$ первых цветов будет размещён ещё один дополнительный треугольник.

Если $B_2 < n \leqslant B3$, то на первом этапе будут размещены d четвёрок цветов, $c-d$ троек из первых трёх цветов и $(n-B_2)/2$ пар из первых двух цветов. Для $(n-B_2)\%2$ первых цветов будет размещён ещё один дополнительный треугольник.

Если $B_3 < n \leqslant B4$, то на первом этапе будут размещены d четвёрок цветов, $c-d$ троек из первых трёх цветов, $b-c$ пар из первых двух цветов и $(n-B_3)$ треугольников первого цвета.

Наконец, если $B_4 < n$, то на первом этапе будут размещены все треугольники и второй этап для размещения не понадобится.

При подсчёте размещённых на втором этапе треугольников мы воспользуемся рассуждением из предыдущей подзадачи. Для каждого цвета определим парный ему. Тогда число размещённых в ячейках фигурок заданного цвета будет равно числу размещённых на первом этапе плюс минимум из того, что осталось в этом цвете, и числа размещённых на первом этапе треугольников цвета, парного текущему.

Для лучшего понимания этой идеи приведём решение на языке Python без дополнительных циклов и массивов.

```
a = int(input())
b = int(input())
c = int(input())
d = int(input())
n = int(input())

pa = 0
pb = 0
pc = 0
pd = 0

B1 = 4 * d

pa += min(n // 4, d)
pb += min(n // 4, d)
pc += min(n // 4, d)
pd += min(n // 4, d)

if n <= B1:
    pa += (n % 4 >= 1)
    pb += (n % 4 >= 2)
    pc += (n % 4 >= 3)

B2 = B1 + 3 * (c - d)

if B1 < n:
    pa += min((n - B1) // 3, c - d)
    pb += min((n - B1) // 3, c - d)
    pc += min((n - B1) // 3, c - d)

if B1 < n and n <= B2:
    pa += ((n - B1) % 3 >= 1)
    pb += ((n - B1) % 3 >= 2)

B3 = B2 + 2 *(b - c)

if B2 < n:
    pa += min((n - B2) // 2, b - c)
    pb += min((n - B2) // 2, b - c)

if B2 < n and n <= B3:
    pa += ((n - B2) % 2 >= 1)

B4 = B3 + (a - b)

if B3 < n:
```

```
    pa += min(n - B3, a - b)


print(pa + min(pc, a - pa))
print(pb + min(pd, b - pb))
print(pc + min(pa, c - pc))
print(pd + min(pb, d - pd))
```

Эта программа набирает 100 баллов.

Рассмотрим ещё один подход к решению задачи:

Заведём два массива: unused (неиспользованные) и used (использованные) длиной 4 – количество треугольников различных цветов. Далее в цикле, пока мы не расставили все треугольники и есть свободные позиции для них, моделируем установку в свободную ячейку треугольника текущего цвета (изменяя состояние двух массивов) и переходим командой $i = (i + 1)\%4$ к следующему цвету.

Когда этот цикл завершится, у нас в $used[i]$ будет храниться число установленных треугольников i-го цвета. Однако возможно установить фигурки этого цвета ещё и в ячейки, занятые треугольниками, «дополняющими до квадрата» треугольники текущего цвета. Для цвета 0 это цвет 2 (и наоборот), для цвета 1 это цвет 3 (и наоборот). Для определения такого номера для текущего i можно воспользоваться битовой операцией (i XOR 2), возвращающей соответствующие значения. Тогда дополнительно мы сможем установить наименьшее из двух значений: unused[i] или used[i ^ 2] (неиспользованных текущего цвета и использованных дополняющего цвета). Здесь операция ^ — это операция побитового исключающего или, её использование приведёт к тому, что число увеличится на 2 или уменьшится на 2 в зависимости от того, входит ли в число бит, значение которого равно 2.

```
unused = [int(input()) for i in range(4)]
sum_unused = sum(unused)
n = int(input())
used = [0] * 4
i = 0
while sum_unused > 0 and n > 0:
    if unused[i]:
        unused[i] -= 1
        used[i] += 1
        sum_unused -= 1
        n -= 1
    i = (i + 1) % 4
for i in range(4):
    print(used[i] + min(unused[i], used[i ^ 2]))
```

Это решение набирает 60 баллов.

Полный балл можно набрать, если перейти от расстановки треугольников по одному к расстановке по большим группам. Сначала мы будем размещать фигурки всех четырёх цветов. Максимальное количество групп, которое мы можем разместить, не может превосходить целую часть числа $n/4$ или количество синих треугольников, которых меньше всего.

```
unused = [int(input()) for i in range(4)]
n = int(input())
used = [0] * 4
group4 = min(n // 4, unused[3])
for i in range(4):
    unused[i] -= group4
    used[i] += group4
n -= 4 * group4
```

Также уменьшим число n на количество размещённых треугольников. Теперь либо нет четырёх свободных клеточек, либо закончился последний цвет. Элементы первых трёх цветов будут

размещаться по три. Аналогично посчитаем, сколько таких троек можно разместить, и уменьшим значение n.

```
group3 = min(n // 3, unused[2])
for i in range(3):
    unused[i] -= group3
    used[i] += group3
n -= 3 * group3
```

Теперь либо нет двух свободных клеточек, либо закончился третий цвет. Посчитаем количество пар треугольников первых двух цветов, которые будут размещены.

```
group2 = min(n // 2, unused[1])
for i in range(2):
    unused[i] -= group2
    used[i] += group2
n -= 2 * group2
```

Теперь либо нет одной свободной клеточки, либо закончился второй цвет. Наконец, все оставшиеся места будут заняты треугольниками первого цвета. В конце также нужно учесть, что некоторые неразмещённые фигурки ещё можно расположить в клетках, частично занятыми другими треугольниками.

```
group1 = min(n, unused[0])
unused[0] -= group1
used[0] += group1
n -= 1 * group1

print(used[0] + min(unused[0], used[2]))
print(used[1] + min(unused[1], used[3]))
print(used[2] + min(unused[2], used[0]))
print(used[3] + min(unused[3], used[1]))
```

Данную программу можно сократить за счёт использования внешнего цикла, так как там совершаются однотипные операции. Внешний цикл перебирает размер группы от 4 до 1.

```
unused = [int(input()) for i in range(4)]
n = int(input())
used = [0] * 4

for group_size in range(4, 0, -1):
    groups = min(n // group_size, unused[group_size - 1])
    for i in range(group_size):
        unused[i] -= groups
        used[i] += groups
    n -= group_size * groups

for i in range(4):
    print(used[i] + min(unused[i], used[i ^ 2]))
```

Эта программа набирает 100 баллов.