Задача 1. Ферзь

Чтобы набрать 40 баллов достаточно написать решение, перебирающее все клетки, на которые можно поставить ферзя. Затем нужно посчитать количество клеток, которые бьёт этот ферзь. Для этого также переберём все оставшиеся клетки доски, и проверим, находятся ли они в одной горизонтали, вертикали или диагонали с выбранной клеткой. Запомним наибольшее количество клеток, которое может бить ферзь. Сложность такого решения будет $O(n^2m^2)$. В примере такого решения строки и столбцы доски нумеруются для удобства с нуля.

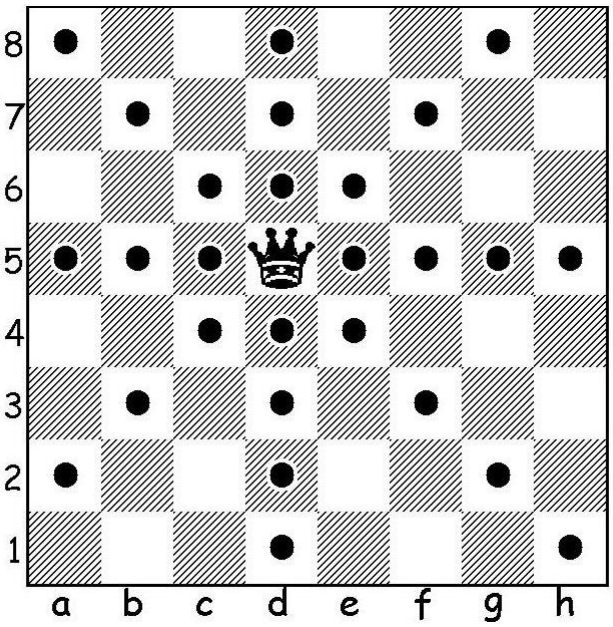
```
n = int(input())
m = int(input())
ans = 0
for x in range(n):
    for y in range(m):
        count = 0
        for xx in range(n):
            for yy in range(m):
                if x==xx or y==yy or x-xx==y-yy or x-xx==yy-y:
                    count += 1
        ans = max(ans, count - 1)
print(ans)
```

Чтобы набрать 80 баллов, нужно уменьшить сложность решения до $O(nm)$. Для этого можно находить количество клеток, которые бьёт ферзь, без цикла, то есть за $O(1)$. Или, наоборот, заметить, что ферзя нужно поставить в центр доски, и найти циклами количество клеток, которые он бьёт. Пример второго решения, где ферзь ставится в клетку с координатами $(\lfloor n/2 \rfloor, \lfloor m/2 \rfloor)$.

```
n = int(input())
m = int(input())
x = n // 2
y = m // 2
count = 0
for xx in range(n):
    for yy in range(m):
        if x==xx or y==yy or x-xx==y-yy or x-xx==yy-y:
            count += 1
print(count - 1)
```

Полное решение имеет сложность $O(1)$. Здесь нужно заметить, что ферзь контролирует наибольшее количество клеток, находясь в центре доски, и посчитать их количество, без использования циклов.

Пусть $n \leqslant m$, то есть доска "вытянута" по горизонтали, иначе поменяем значения n и m. Тогда в одной горизонтали с ферзём находятся $m-1$ клетка, а в одной вертикали — $n-1$ клетка. Диагонали, проходящие через ферзя, также могут содержать не более n клеток (поэтому в каждой из них не более $n-1$ клетки, за вычетом клетки, в которой стоит ферзь), поэтому ответ будет равен $(m-1)+3\cdot(n-1)$. Но есть одно исключение: на квадратной доске, сторона которой имеет чётную длину, одна из диагоналей будет короче на одну клетку. Это, например, случай доски 8×8 (первый пример из условия), для которой ответ равен 27, а не 28. Почему так происходит, можно видеть на рисунке.



В этом случае просто вычтем 1 из ответа. Пример такого решения.

```
n = int(input())
m = int(input())
if n > m:
    n, m = m, n
ans = 3 * n + m - 4
if n % 2 == 0 and m == n:
    ans -= 1
print(ans)
```

Задача 2. Рамка для рисунка

Начнём с переборных решений, набирающих частичные баллы. Пусть n_1 и n_2 — количество палочек длины 1 и 2 соответственно.

40 баллов можно набрать, если перебирать две стороны прямоугольника a и b. Проверим, можно ли сложить из имеющихся палочек прямоугольник $a \times b$. Должны выполняться два условия: общая длина всех палочек $n_1 + 2n_2$ должна быть не меньше периметра прямоугольника, равного $2a + 2b$, и каждая сторона нечётной длины должна содержать хотя бы одну палочку длины 1, поэтому значение n_1 должно быть не меньше количества нечётных чисел среди сторон a, b, a, b. Пример такого решения.

```
n1 = int(input())
n2 = int(input())
max_side = (n1 + 2 * n2) // 2
ans = 0
for a in range(1, max_side + 1):
    for b in range(1, max_side + 1):
        if n1 + 2 * n2 >= 2 * a + 2 * b and n1 >= 2 * (a % 2 + b % 2):
            ans = max(ans, a * b)
print(ans)
```

Чтобы набрать 60 баллов нужно перебирать только одну сторону, а не две. Пусть это сторона a. Максимальное значение одной стороны, как и в предыдущем решении, равно $\lfloor \frac{n_1+2n_2}{2} \rfloor$ (целочисленное частное от деления суммы длин всех палочек на 2). Тогда у нас будет две стороны длины a. Проверим условие, что n_1 не меньше, чем $2(a \bmod 2)$, то есть если a — нечётное, то найдётся хотя бы две палочки длины 1.

Теперь определим наибольшее подходящее значение b для данного значения a. Для этого посчитаем длину оставшихся палочек $n_1 + 2n_2 - 2a$ и поделим её на 2. При этом могло оказаться, что $n_1 < 2(a \bmod 2 + b \bmod 2)$, то есть значение n_1 оказалось меньше, чем число нечётных чисел среди значений a, b, a, b, то нам не хватит палочек длины 1 для того, чтобы собрать нечётные отрезки длины a, b, a, b. Но ранее мы проверили, что нам хватает палочек длины 1 для того, чтобы собрать только отрезки a и a, поэтому это возможно только в случае нечётного b. В этом случае уменьшим значение b на 1. Так мы определяем наибольшее значение второй стороны b для ранее выбранной стороны a. Запомним наибольшее из значений площадей прямоугольников $a \times b$.

```
n1 = int(input())
n2 = int(input())

max_side = (n1 + 2 * n2) // 2

ans = 0
for a in range(1, max_side + 1):
    if n1 < 2 * (a % 2):
        continue
    b = (n1 + 2 * n2 - 2 * a) // 2
    if n1 < 2 * (a % 2 + b % 2):
        b -= 1
    ans = max(ans, a * b)
print(ans)
```

Чтобы написать полное решение, нужно избавится и от перебора всех возможных значений одной стороны. Нужно заметить, что среди всех прямоугольников с одинаковым периметром максимальная площадь будет у квадрата или у прямоугольника, стороны которого различаются на 1. Действительно, пусть прямоугольник имеет стороны $a \times b$, при этом $a + 1 < b$. Рассмотрим прямоугольник $(a + 1) \times (b - 1)$ с таким же периметром. Его площадь будет равна $ab + b - a - 1$, то есть больше площади ab.

Поэтому для максимизации периметра в качестве значения одной из сторон нужно выбрать $\frac{1}{4}$ от максимально возможного периметра. Для этого в качестве значения минимальной стороны a возьмём значение $\lfloor \frac{n_1+2n_2}{4} \rfloor$, а значение b подберём наибольшее подходящее значение, как в предыдущем решении. Но если значение a оказалось нечётным, то для формирования сторон длины a понадобится минимум 2 палочки длины 1, и, возможно, нам не хватит палочек длины 1 для достижения максимально возможного значения b. Поэтому необходимо рассмотреть как чётное, так и нечётное значение a, то есть нужно взять не только $a = \lfloor \frac{n_1+2n_2}{4} \rfloor$, но и значение на 1 меньше — $a = \lfloor \frac{n_1+2n_2}{4} \rfloor$, так как одно из них будет чётным. Для каждого из этих значений a выберем подходящее b и найдём наибольшее значение $a \times b$.

Пример такого решения.

```
n1 = int(input())
n2 = int(input())
ans = 0
side = (n1 + 2 * n2) // 4
for a in range(side - 1, side + 1):
    if n1 < 2 * (a % 2):
        continue
    b = (n1 + 2 * n2 - 2 * a) // 2
    if n1 < 2 * (a % 2 + b % 2):
        b -= 1
    ans = max(ans, a * b)
print(ans)
```

Есть и другие способы решения задачи. Например, можно попробовать конструктивно построить

решение так, чтобы две стороны прямоугольника $a \times b$ оказались максимально большими и при этом близки друг к другу. Для этого сначала разложим палочки длины 2 поровну на все стороны. У нас останется 0, 1, 2 или 3 палочки длины 2. Если осталось хотя бы две палочки длины 2, то увеличим на 2 длины двух сторон a. Если после этого осталась хотя бы одна палочка длины 2 и ещё одна палочка длины 2 или две палочки длины 1, то также можно сторону b увеличить на 2. Иначе попробуем используя палочки длины 1 выровнять длины сторон. После выравнивания длин сторон все оставшиеся палочки длины 1 разложим поровну по всем сторонам. После этого останется не более трёх палочек длины 1, если их две или три — то можно длины двух сторон увеличить на 1.

Сложность реализации такого решения в том, что нужно аккуратно рассмотреть все случаи, не пропустив ни одного.

```
n1 = int(input())
n2 = int(input())

a = 2 * (n2 // 4)
b = 2 * (n2 // 4)
n2 %= 4

if n2 >= 2:
    n2 -= 2
    a += 2
if n2 == 1 and n1 >= 2:
    b += 2
    n2 = 0
    n1 -= 2
while a < b and n1 >= 2:
    a += 1
    n1 -= 2
while a > b and n1 >= 2:
    b += 1
    n1 -= 2
a += n1 // 4
b += n1 // 4
n1 %= 4
if n1 >= 2:
    a += 1
print(a * b)
```

Задача 3. Телефонный справочник

Обозначим через s — исходную строку, n — её длину, а a — мощность алфавита, то есть количество возможных символов. Для английского алфавита $a = 26$.

У этой задачи есть много разных идей решения. Начнём с самой простой — будем перебирать пары символов $i < j$ двумя вложенными циклами и менять символы строки s_i и s_j местами. Можно дополнительно проверять, что $s_i > s_j$, то есть при перестановке символов строка станет меньше в лексикографическом порядке. Из всех полученных таким образом строк выберем минимальную. Такое решение имеет сложность $O(n^3)$, так как пар символов $O(n^2)$, а сравнение строк выполняется за $O(n)$.

```
s = input()
ans = s
for i in range(len(s)):
    for j in range(i + 1, len(s)):
        if s[i] > s[j]:
            s1 = s[:i] + s[j] + s[i + 1:j] + s[i] + s[j + 1:]
```

```
            if s1 < ans:
                ans = s1
print(ans)
```

Можно улучшить это решение, если не сравнивать строки явно и даже не строить строку с ответом. Для этого заметим, что пусть у нас есть несколько способов выполнить перестановку двух символов. Например, в строке “ccddaabb” мы можем поменять одну из букв “c” или “d” с одной из букв “a” или “b”. Пусть есть два способа это сделать, например, поменять символ s_{i_1} и s_{j_1} или поменять s_{i_2} и s_{j_2}.

Первый способ даст меньшую строку, если i_1 меньше i_2, то есть мы уменьшим тот символ, который стоит раньше. Если $i_1 = i_2$, то первый способ будет меньше, если $s_{j_1} < s_{j_2}$, то есть на место первого заменяемого символа ставится меньший символ. В случае равенства символов первый способ будет меньше при $j_1 > j_2$, т.к. второй участвующий в обмене символ увеличивается, и мы должны выбрать его как можно позже. Например, для строки “ccddaabb” наилучшим будет ответ “acddacbb”.

Поэтому для построения решения сложности $O(n^2)$ можно перебирать пары переставляемых символов i и j и запоминать такую пару, которая даст лучший ответ (с минимальным значением i, при равенстве — с минимальным значением s_j, при равенстве — с максимальным значением j).

Пример такого решения. В этом решении в переменной *ans* хранится тройка чисел $(i, s_j, -j)$. Перед значением j поставили знак минус, потому что кортежи сравниваются в лексикографическом порядке, а нам необходимо, чтобы при минимальных i и s_j был выбран тот ответ, у которого значение j будет больше, поэтому мы будем хранить $-j$. Запомним наименьший из всех подходящих кортежей, потом выведем ответ, переставив два символа, если был найден подходящий ответ.

```
s = input()
ans = (len(s), '', 0)
for i in range(len(s)):
    for j in range(i + 1, len(s)):
        if s[i] > s[j]:
            ans = min(ans, (i, s[j], -j))
if ans[0] != len(s):
    i = ans[0]
    j = -ans[2]
    s = s[:i] + s[j] + s[i + 1:j] + s[i] + s[j + 1:]
print(s)
```

Для дальнейшего улучшения этого решения заметим, что если рассматриваем символ s_j, который мы хотим поменять с каким-то предыдущим символом, то можно рассматривать не все символы s_i, а только самые первые вхождения какой-либо буквы, например, самое первое вхождение буквы “z”, самое первое вхождение буквы “y” и т.д. Запомним для каждой буквы алфавита от “a” до ”z” её первое вхождение.

Затем переберём символы s_j, которые мы рассматриваем, как правый символ из двух переставляемых. Но в качестве левого символа s_i будем рассматривать не все символы, а только первые вхождения тех символов, значение которых больше, чем s_j. В примере решение ниже это цикл по переменной *c*. Идея выбора наилучшего ответа аналогично предыдущему решению. Такое решение имеет сложность $O(an)$.

```
s = input()
n = len(s)
first = [n] * 26
for i in range(len(s)):
    c = ord(s[i]) - ord('a')
    if first[c] == n:
        first[c] = i
ans = (n, '', 0)
for j in range(n):
```

```
    for c in range(ord(s[j]) - ord('a') + 1, 26):
        i = first[c]
        if i < j:
            ans = min(ans, (i, s[j], -j))
if ans[0] != n:
    i = ans[0]
    j = -ans[2]
    s = s[:i] + s[j] + s[i + 1:j] + s[i] + s[j + 1:]
print(s)
```

Следующим шагом будет запоминание не только первого вхождения каждого символа, но и последнего вхождения, потому что для каждой пары символов c и d, где $c > d$ нужно переставлять первое вхождение c с последним вхождением d. После нахождения первой и последней позиции (первый цикл) переберём все пары символов $c > d$ и рассмотрим ответ, который получается перестановкой первого вхождения c (переменная i) и последнего вхождения d (переменная j). Такое решение будет иметь сложность $O(n + a^2)$, и оно уже набирает 100 баллов.

```
s = input()
n = len(s)
first = [n] * 26
last = [-1] * 26
for i in range(len(s)):
    c = ord(s[i]) - ord('a')
    if first[c] == n:
        first[c] = i
    last[c] = i

ans = (n, '', 0)

for c in range(26):
    i = first[c]
    for d in range(c):
        j = last[d]
        if i < j:
            ans = min(ans, (i, d, -j))

if ans[0] != n:
    i = ans[0]
    j = -ans[2]
    s = s[:i] + s[j] + s[i + 1:j] + s[i] + s[j + 1:]
print(s)
```

Есть и решение сложности $O(n)$, то есть не зависящее от мощности алфавита. Для этого поймём, как будет устроена строка, которую нельзя уменьшить перестановкой двух символов. В такой строке символы будут идти в порядке неубывания, например, “aaacdddfffkkkm”. Пропустим префикс строки, состоящий из неубывающих символов. Пусть после этого мы встретили символ, который меньше предыдущего, например, после символа “m” пусть идёт символ “h”. Этот символ можно переставить вперёд, при этом мы найдём самый первый символ, с которым его можно переставить. Это будет первый символ на префиксе, который больше его, то есть в этом примере это будет первое вхождение “k”. Найти этот символ мы можем обратным циклом, в котором индекс символа будет уменьшаться. Запомним эти символы в ответе.

Если дальше мы встретим такой же символ, то, поскольку он будет позже, нужно обновить позицию второго символа в ответе. А если мы встретим меньший символ, то мы сможем переставить его с символом, который находится раньше первого запомненного символа. В этом случае опять

запустим цикл, идущий к началу строки, в поисках первого символа, который больше данного. Пример такого решения.

```
s = input()
i = 0
while i < len(s) - 1 and s[i] <= s[i + 1]:
    i += 1
ans_i = i
ans_j = i
ans_c = s[i]
for j in range(i + 1, len(s)):
    if s[j] == ans_c:
        ans_j = j
    elif s[j] < ans_c:
        ans_c = s[j]
        ans_j = j
        while ans_i > 0 and s[ans_i - 1] > ans_c:
            ans_i -= 1
if ans_i != ans_j:
    s = s[:ans_i] + s[ans_j] + s[ans_i + 1:ans_j] + s[ans_i] + s[ans_j + 1:]
print(s)
```

Задача 4. Задачи на печать!

Будем рассматривать задачи последовательно, определяя, в каком режиме должна быть напечатана очередная задача. Например, если в задаче 2 страницы, то её условие должно быть напечатана только в двустороннем режиме. А если в задаче 1 страница, то всё зависит от того, в каком режиме была напечатана предыдущая страница. Если это был односторонний режим, то мы расширим предыдущий диапазон печати на новую задачу, а если двусторонний — то эту одну страницу можно напечатать в двустороннем режиме, однако, следующая страница обязательно должна стать началом нового диапазона печати, который может быть как односторонним, так и двусторонним.

Заведём переменную *mode*, в которой будет храниться текущий режим печати — 1 для односторонней печати и 2 для двусторонней. Значение 0 означает, что очередная страница должна стать началом нового диапазона печати, который может быть любым. В переменной *ans* хранится общее число диапазонов печати. В переменной p хранится количество страниц в текущей задаче.

Далее нужно аккуратно разобрать все случаи. Если $p = 2$ то мы обязательно переходим в режим 2, при этом если ранее режим был другим, то к ответу прибавляем 1.

Если $p = 1$, то в режиме 1 не нужно делать ничего, в режиме 2 эта страница печатается в двустороннем режиме, но нужно перейти в режим 0 для начала нового диапазона со следующей страницы (потому что на обороте этой страницы ничего нельзя печатать), а в режиме 0 нужно перейти в режим 1, начав новый диапазон, то есть добавив к ответу 1.

Наконец, разберём случай $p = 3$. Если до этого был режим 1, то мы печатаем одну страницу односторонней печатью, а ещё две страницы — двусторонней, поэтому нужно перейти в режим 2. Если мы были в режиме 2, то все три страницы можно напечатать в двустороннем режиме, потом нужно перейти в режим 0, потому что придётся начать новый диапазон. Аналогично поступим, когда режим был равен 0 — начнём новый двусторонний режим, напечатаем три страницы, а затем придётся начать новый диапазон, то есть в этом случае нужно просто увеличить значение *ans* на 1, сохранив значение *mode* равным 0.

Пример такого решения.

```
ans = 0
mode = 0
n = int(input())
for i in range(n):
    p = int(input())
```

```
    if p == 1:
        if mode == 2:
            mode = 0
        elif mode == 0:
            mode = 1
            ans += 1
    if p == 2:
        if mode != 2:
            ans += 1
            mode = 2
    if p == 3:
        if mode == 1:
            mode = 2
            ans += 1
        elif mode == 2:
            mode = 0
        else:
            ans += 1
print(ans)
```

Частичные решения можно получить используя полный перебор вариантов.

Задачу можно решить и динамическим программированием, в котором целевой функцией $f(i)$ будет количество диапазонов, необходимое для печати первых i задач. При этом придется ввести и второй параметр, аналогичный по смыслу переменной *mode*.

Задача 5. Вечер кёрлинга

Будем говорить о матчах, как об отрезках на прямой, у которых левый конец — момент начала матча, правый конец — момент окончания матча. Также будем говорить о том, что два отрезка не пересекаются, если левый конец одного отрезка не меньше правого конца другого отрезка. Концы могут и совпадать, но это допускается в условии этой задачи.

Без условия о наличии перерыва эта задача решается при помощи жадного алгоритма. Отсортируем все отрезки по правому концу, и будем перебирать их в порядке неубывания правого конца. Если левый конец очередного рассматриваемого отрезка не меньше, чем правый конец последнего выбранного отрезка, то добавляем отрезок в множество выбранных. То есть мы выбираем очередной отрезок, как отрезок с минимальным правым концом, который не пересекается с последним ранее выбранным отрезком.

Заметим, что этот алгоритм также находит способ выбрать i отрезков так, чтобы они попарно не пересекались, а правый конец последнего выбранного отрезка был как можно меньше.

Если этот алгоритм применить ещё раз, но только от конца к началу, то можно решить задачу "с конца": для каждого i мы найдём способ выбрать i отрезков так, чтобы они не пересекались и начало первого из них был как можно больше.

Теперь научимся обрабатывать перерыв. Для этого будем перебирать количество матчей, которые Василиса просмотрит до перерыва. Мы уже нашли минимальное время, за которое Василиса может просмотреть эти матчи. Добавим к этому времени продолжительность перерыва. Затем, используя ранее найденные значения, найдём какое максимальное количество матчей можно просмотреть за оставшееся время. Для этого будем использовать метод двух указателей: если какому-то количеству i матчей, которые Василиса просмотрит до перерыва, соответствует j матчей, которые можно просмотреть после перерыва, то при увеличении i значение j будет уменьшаться. Поэтому сложность такого алгоритма после сортировки будет $O(n)$, а вместе с сортировкой — $O(n \log n)$.

```
n = int(input())
t = int(input())
games = []
for i in range(1, n + 1):
```

```
    l, r = map(int, input().split())
    games.append([i, l, r])

games_sorted_end = sorted(games, key = lambda elem: elem[2])
min_end_time = [0]
min_end_game = [0]
prev_game = [0] * (n + 1)
for i, l, r in games_sorted_end:
    if l >= min_end_time[-1]:
        prev_game[i] = min_end_game[-1]
        min_end_time.append(r)
        min_end_game.append(i)

INF = 2 * 10**9
games_sorted_beg = sorted(games, key = lambda elem: elem[1], reverse=True)
max_begin_time = [INF]
max_begin_game = [0]
next_game = [0] * (n + 1)
for i, l, r in games_sorted_beg:
    if r <= max_begin_time[-1]:
        next_game[i] = max_begin_game[-1]
        max_begin_time.append(l)
        max_begin_game.append(i)

ans_m = -1
game_before_break = 0
game_after_break = 0
j = len(max_begin_time) - 1
for i in range(1, len(min_end_time)):
    while max_begin_time[j] - min_end_time[i] < t:
        j -= 1
    if j > 0 and i + j > ans_m:
        ans_m = i + j
        game_before_break = min_end_game[i]
        game_after_break = max_begin_game[j]

print(ans_m)
if ans_m > 0:
    ans = []
    curr = game_before_break
    while curr:
        ans.append(curr)
        curr = prev_game[curr]
    ans = ans[::-1]
    curr = game_after_break
    while curr:
        ans.append(curr)
        curr = next_game[curr]
    print("␣".join(map(str, ans)))
```

Возможные частичные неэффективные решения могут использовать перебор или динамическое программирование.

В решении с перебором при $n \leqslant 15$ нужно отсортировать все матчи, зачем перебрать 2^n под-

множеств выбранных матчей и проверить, что они удовлетворяют условию непересечения матчей и наличия перерыва, затем выбрать подходящее подмножество, содержащее наибольшее число элементов.

В решении динамическим программированием можно рассмотреть целевую функцию $f(k, b)$ — минимальное время, за которое можно просмотреть k матчей. Значение b будет равно 0 или 1 и означает отсутствие или наличие перерыва в выбранной последовательности из k матчей. Такое решение будет иметь сложность $O(n^2)$.