

А. Повторяем таблицу умножения

Подгруппа 1 ($n \leq 5$)

В этой подгруппе число n может быть равно только 1, 2, 3, 4 или 5. Для каждого из этих случаев можно просто написать 10 строк с выводом таблицы умножения.

Например, если $n = 2$, то выводим «2 x 1 = 2», «2 x 2 = 4», «2 x 3 = 6» и так далее.

Для других n делаем то же самое. Так как вариантов всего пять, можно в программе проверить, чему равно n , и вывести нужный столбик.

Пример кода на Python:

```
n = int(input())
if n == 1:
    print("1 x 1 = 1")
    print("1 x 2 = 2")
    print("1 x 3 = 3")
    print("1 x 4 = 4")
    print("1 x 5 = 5")
    print("1 x 6 = 6")
    print("1 x 7 = 7")
    print("1 x 8 = 8")
    print("1 x 9 = 9")
    print("1 x 10 = 10")
elif n == 2:
    print("2 x 1 = 2")
    print("2 x 2 = 4")
    print("2 x 3 = 6")
    print("2 x 4 = 8")
    print("2 x 5 = 10")
    print("2 x 6 = 12")
    print("2 x 7 = 14")
    print("2 x 8 = 16")
    print("2 x 9 = 18")
    print("2 x 10 = 20")
elif n == 3:
    print("3 x 1 = 3")
    print("3 x 2 = 6")
    print("3 x 3 = 9")
    print("3 x 4 = 12")
    print("3 x 5 = 15")
    print("3 x 6 = 18")
    print("3 x 7 = 21")
    print("3 x 8 = 24")
    print("3 x 9 = 27")
    print("3 x 10 = 30")
# для n = 4 и n = 5 аналогично
```

Подгруппа 2 (полные ограничения)

Здесь n может быть любым от 1 до 100. Писать 100 вариантов длинными списками было бы слишком долго. Лучше использовать цикл. Нужно вывести 10 строк, в каждой из которых написано « $n \times i = n \cdot i$ » для i от 1 до 10. Цикл повторяется 10 раз, на каждом шаге вычисляется произведение и выводится строка в нужном формате.

Пример кода на Python:

```
n = int(input())
for i in range(1, 11):
    print(n, "x", i, "=", n * i)
```

В. Сетевой протокол

Подгруппа 1 ($n \leq 10$)

В этой подгруппе n небольшое, не больше десяти. Можно просто вычислить 5 в степени n , а потом взять остаток от деления на 7. В Python это делается с помощью операторов `**` (возведение в степень) и `%` (остаток от деления). Или можно вручную посчитать ответ для каждого n .

Пример кода на Python:

```
n = int(input())
print((5 ** n) % 7)
```

Подгруппа 2 ($n \leq 10^7$)

Здесь n может быть до десяти миллионов. Если пытаться вычислить 5 в степени n напрямую, то мы не успеем это сделать, вычисления будут крайне долгими. Обратим внимание, что нам нужен только остаток от деления на 7. В арифметике есть полезное свойство: при умножении двух чисел остаток от деления их произведения на 7 равен остатку от деления произведения их остатков на 7. То есть $(a * b) \% 7 = ((a \% 7) * (b \% 7)) \% 7$. Поэтому можно изначально приравнять ответ к единице, потом n раз умножать его на 5, каждый раз беря остаток от деления на 7. Числа при этом никогда не станут больше 6, и программа выполнится за разумное время (около 10 миллионов шагов).

Пример кода на Python:

```
n = int(input())
result = 1
for i in range(n):
    result = (result * 5) % 7
print(result)
```

Подгруппа 3 (полные ограничения)

Теперь n может быть огромным (до 10^{18}). Цикл из стольких шагов выполнить невозможно. Нужно обнаружить следующую закономерность. Посчитаем несколько значений $5^n \bmod 7$:

- $n = 1 \rightarrow 5 \bmod 7 = 5$
- $n = 2 \rightarrow 25 \bmod 7 = 4$
- $n = 3 \rightarrow 125 \bmod 7 = 6$
- $n = 4 \rightarrow 625 \bmod 7 = 2$
- $n = 5 \rightarrow 3125 \bmod 7 = 3$

- $n = 6 \rightarrow 15625 \bmod 7 = 1$
- $n = 7 \rightarrow 78125 \bmod 7 = 5$
- $n = 8 \rightarrow 390625 \bmod 7 = 4$
- $n = 9 \rightarrow 1953125 \bmod 7 = 6$
- $n = 10 \rightarrow 9765625 \bmod 7 = 2$
- $n = 11 \rightarrow 48828125 \bmod 7 = 3$
- $n = 12 \rightarrow 244140625 \bmod 7 = 1$
- $n = 13 \rightarrow 1220703125 \bmod 7 = 5$

Мы видим, что остатки повторяются каждые 6 шагов: 5, 4, 6, 2, 3, 1, а затем снова 5, 4, ... Поэтому достаточно найти остаток от деления n на 6 и по нему определить ответ:

- если остаток 1 \rightarrow ответ 5
- если остаток 2 \rightarrow ответ 4
- если остаток 3 \rightarrow ответ 6
- если остаток 4 \rightarrow ответ 2
- если остаток 5 \rightarrow ответ 3
- если остаток 0 \rightarrow ответ 1

Пример кода на Python:

```
n = int(input())
r = n % 6
if r == 1:
    print(5)
elif r == 2:
    print(4)
elif r == 3:
    print(6)
elif r == 4:
    print(2)
elif r == 5:
    print(3)
else:
    print(1)
```

С. Прямоугольники

Подгруппа 1 (A = B, C = D)

В этой подгруппе два квадрата со сторонами A и C . Их можно объединить в прямоугольник, если $A = C$, сдвинув к друг другу. Квадрат из двух квадратов никак не получить.

Пример кода на Python:

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

if a == c:
    print('RECTANGLE')
else:
    print(':(')
```

Подгруппа 2 (A = B)

В этой подгруппе у Алисы квадрат со стороной A , а у Боба прямоугольник со сторонами C на D . Их можно объединить в прямоугольник, если $A = C$ или $A = D$, сдвинув прямоугольник соответствующей стороной к квадрату. Квадрат из квадрата и прямоугольника никак не получить.

Пример кода на Python:

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

if c == a or d == a:
    print('RECTANGLE')
else:
    print(':(')
```

Подгруппа 3 (A = C)

В этой подгруппе два прямоугольника со сторонами A на B и C на D , где $A = C$. Эти прямоугольники гарантированно можно объединить в прямоугольник, сдвинув их сторонами, которые соответствуют длинам A и C .

Как получить квадрат?

Если мы сдвинем прямоугольники сторонами, которые соответствуют длинам A и C , и $A = C = B + D$, то мы получим квадрат. В этом несложно убедиться,

порисовав примеры прямоугольников, которые удовлетворяют этим требованиям.

Аналогично, если сдвинуть прямоугольники сторонами, которые соответствуют длинам A и D , и $A = D = B + C$, то мы также получим квадрат.

Аналогично, мы можем получить квадрат из прямоугольников, которые удовлетворяют требованию $B = C = A + D$ или $B = D = A + C$.

Пример кода на Python:

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

if (a == c and b + d == a) or (a == d and b + c == a) \
    or (b == c and a + d == b) or (b == d and a + c == b):
    print('SQUARE')
else:
    print('RECTANGLE')
```

Подгруппа 4 (полные ограничения)

В этой подгруппе два любых прямоугольника. Получить из них квадрат можно при тех же условиях, что и в предыдущем решении. Получить из них прямоугольник можно, если возможно сдвинуть их к друг другу сторонами с равными длинами ($A = C$ или $A = D$, или $B = C$, или $B = D$).

Пример кода на Python:

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

if (a == c and b + d == a) or (a == d and b + c == a) \
    or (b == c and a + d == b) or (b == d and a + c == b):
    print('SQUARE')
elif a == c or a == d or b == c or b == d:
    print('RECTANGLE')
else:
    print(':(')
```

D. Настольная игра

Подгруппа 1 ($n = 1$)

В этой подгруппе Боб кидает только один кубик. На кубике может выпасть число от 1 до 6. Нужно посчитать, сколько исходов дают сумму не меньше X . Для одного кубика сумма равна выпавшему числу. Значит, нужно посчитать, сколько чисел от 1 до 6 больше или равны X . Можно пробежать циклом по всем возможным значениям от 1 до 6 и, если значение $\geq X$, прибавлять 1 к ответу.

Пример кода на Python:

```
n = int(input())
x = int(input())

c = 0
for i in range(1, 7):
    if i >= x:
        c += 1
print(c)
```

Подгруппа 2 ($n \leq 3$)

Здесь кубиков не больше трёх. Можно перебрать все возможные исходы с помощью вложенных циклов. Для двух кубиков – два цикла, для трёх – три цикла. Внутри самого глубокого цикла считаем сумму всех выпавших значений. Если сумма $\geq X$, то прибавляем 1 к ответу. Так как n маленькое, количество итераций не превышает $6^3 = 216$, что очень быстро.

Пример кода на Python:

```

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

c = 0
if n == 1:
    for i1 in range(1, 7):
        if i1 >= x:
            c += 1
elif n == 2:
    for i1 in range(1, 7):
        for i2 in range(1, 7):
            if i1 + i2 >= x:
                c += 1
else:
    for i1 in range(1, 7):
        for i2 in range(1, 7):
            for i3 in range(1, 7):
                if i1 + i2 + i3 >= x:
                    c += 1

print(c)

```

Подгруппа 3 ($n \leq 8$)

Теперь кубиков может быть до восьми. Продолжаем идею вложенных циклов. Для каждого значения n от 1 до 8 пишем свой блок с соответствующим количеством циклов. Для $n = 8$ это восемь вложенных циклов. Такой перебор возможен, потому что $6^8 \approx 1,7$ миллиона итераций, что укладывается в одну секунду.

Перебор всех исходов можно организовать не в виде вложенных циклов, а в виде рекурсии (перебираем внутри функции значение, которое выпало на i -м кубике, и для каждого значения вызываем эту же функцию, которая будет перебирать значения на $(i + 1)$ -м кубике), но скорее всего такое решение на Python не пройдет по времени.

Пример кода на Python (версия с вложенными циклами):

```

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

c = 0
if n == 1:
    for i1 in range(1, 7):
        if i1 >= x:
            c += 1
elif n == 2:
    for i1 in range(1, 7):
        for i2 in range(1, 7):
            if i1 + i2 >= x:
                c += 1
# ...
# аналогично для 3 <= n <= 7
# ...
elif n == 8:
    for i1 in range(1, 7):
        for i2 in range(1, 7):
            for i3 in range(1, 7):
                for i4 in range(1, 7):
                    for i5 in range(1, 7):
                        for i6 in range(1, 7):
                            for i7 in range(1, 7):
                                for i8 in range(1, 7):
                                    if i1 + i2 + i3 + i4 + i5 + i6 + i7 + i8 >= x:
                                        c += 1
print(c)

```

Подгруппа 4 ($X \geq 6 \cdot n - 2$)

В этой подгруппе X очень близко к максимально возможной сумме.

Максимальная сумма, которая может выпасть на n кубиках, равна $6n$. Посчитаем, сколько исходов дают сумму, равную $6n$, $6n - 1$ и $6n - 2$.

- Сумма $6n$ получается только в одном случае: на всех кубиках выпало 6.
- Сумма $6n - 1$ получается, если на одном кубике выпало 5, а на остальных 6. Кубик с пятёркой можно выбрать n способами (первый, второй, ..., n -й). Остальные кубики все шестёрки. Значит, сумма $6n - 1$ присутствует в n исходах.
- Сумма $6n - 2$ получается двумя способами:
 1. На одном кубике выпало 4, на остальных 6. Таких исходов n (выбираем кубик с четвёркой).
 2. На двух кубиках выпало по 5, на остальных 6. Таких исходов равно числу способов выбрать два кубика из n , то есть $n \cdot (n - 1) / 2$.

Значит, сумма $6n - 2$ присутствует в $n + n \cdot (n - 1) / 2$ исходах.

Следовательно, для $X = 6n$ ответ равен 1, для $X = 6n - 1$ ответ равен $1 + n$, а для $X = 6n - 2$ ответ равен $1 + 2n + n \cdot (n - 1) / 2$.

Пример кода на Python:

```
n = int(input())
x = int(input())

if x == n * 6:
    print(1)
elif x == n * 6 - 1:
    print(1 + n)
elif x == n * 6 - 2:
    print(1 + n * 2 + n * (n - 1) // 2)
```

Подгруппа 5 (полные ограничения, $n \leq 12$)

Здесь n может быть до 12, и полный перебор всех 6^n вариантов (более 2 миллиардов) уже не проходит. Поступим так: заранее посчитаем для пяти кубиков, сколько исходов дают сумму, большую или равную каждому возможному значению S от 0 до 30. Это можно сделать пятью вложенными циклами. Затем, если $n > 5$, перебираем значения на первых $(n - 5)$ кубиках с помощью вложенных циклов (их будет не более 7, потому что $n \leq 12$). Для каждого такого набора считаем уже набранную сумму sum_first . Тогда оставшимся пяти кубикам нужно набрать сумму не меньше, чем $X - sum_first$. Если $X - sum_first \leq 0$, то подходят все исходы пяти кубиков. Если $X - sum_first > 30$, то ни один исход не подходит. Иначе прибавляем заранее посчитанное количество для суммы, большей или равной $X - sum_first$. Если $n \leq 5$, то просто перебираем все кубики вложенными циклами (как в подгруппе 3). Такой способ работает быстро, потому что число итераций для $(n-5)$ кубиков не превышает $6^7 \approx 280$ тысяч, а подсчет для пяти кубиков делается один раз.

Пример кода на Python:

```

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

precalc = [0] * 31
for i1 in range(1, 7):
    for i2 in range(1, 7):
        for i3 in range(1, 7):
            for i4 in range(1, 7):
                for i5 in range(1, 7):
                    S = i1 + i2 + i3 + i4 + i5
                    precalc[S] += 1
for i in range(29, -1, -1):
    precalc[i] += precalc[i + 1]
c = 0

# ...
# для 1 <= n <= 5 обычный перебор
# ...
# для 6 <= n <= 9 аналогично реализации для n = 10
# ...
if n == 10:
    for i1 in range(1, 7):
        for i2 in range(1, 7):
            for i3 in range(1, 7):
                for i4 in range(1, 7):
                    for i5 in range(1, 7):
                        sum_first = i1 + i2 + i3 + i4 + i5
                        if x - sum_first <= 0:
                            c += precalc[0]
                        elif x - sum_first <= 30:
                            c += precalc[x - sum_first]
# ...
# для 11 <= n <= 12 аналогично реализации для n = 10
# ...

print(c)

```

Е. PPSHneyneF4

Подгруппа 1 (n = 1)

Программа состоит из одной команды. Это команда print, так как гарантируется, что во входных данных есть хотя бы одна команда print. Выводим ОК и соответствующий символ.

Пример кода на Python:

```
n = int(input())
cmd, data = input().split()

print('OK')
print(data)
```

Подгруппа 2 (n = 2)

Программа состоит из двух команд. Нужно рассмотреть все возможные варианты, чтобы определить результат: ОК, ERROR или INFINITY.

- Если первая команда print и вторая print – выводим оба символа (ОК).
- Если первая print, вторая call 1 – выводим два раза символ первой команды (так как вызов возвращается на первую же команду print).
- Если первая print, вторая call 2 – бесконечность (вызов самой себя).
- Если первая print, вторая call с другим номером – ошибка.
- Если первая call 2, вторая print – выводим два раза символ второй команды.
- Если первая call 1 – бесконечность.
- Иначе – ошибка.

Пример кода на Python:

```

n = int(input())
cmd = [0] * n
data = [0] * n
for i in range(n):
    cmd[i], data[i] = input().split()
    if cmd[i] == 'call':
        data[i] = int(data[i])

if n == 1:
    print('OK')
    print(data[0])
else:
    if cmd[0] == 'print':
        if cmd[1] == 'print':
            print('OK')
            print(data[0] + data[1])
        elif data[1] == 1:
            print('OK')
            print(data[0] + data[0])
        elif data[1] == 2:
            print('INFINITY')
        else:
            print('ERROR')
    else:
        if data[0] == 2:
            print('OK')
            print(data[1] + data[1])
        elif data[0] == 1:
            print('INFINITY')
        else:
            print('ERROR')

```

Подгруппа 3 (только команды print)

Так как здесь все команды print, то ошибок и бесконечных процессов нет. Выводим ОК и все символы из команд.

Пример кода на Python:

```

n = int(input())
cmd = [0] * n
data = [0] * n
for i in range(n):
    cmd[i], data[i] = input().split()

print('OK')
for i in range(n):
    print(data[i], end = '')
print()

```

Подгруппа 4 (каждый call ведёт на print)

В этой подгруппе все вызовы указывают на команды, которые являются print. Сначала проверяем, нет ли ошибок (номер call вне диапазона). Если есть ошибка – выводим ERROR. Очевидно, что бесконечных процессов быть не может, поэтому выводим OK и затем для каждой команды:

- если команда print – выводим её символ;
- если команда call – выводим символ той команды print, на которую она указывает.

Пример кода на Python:

```
n = int(input())
cmd = [0] * n
data = [0] * n
for i in range(n):
    cmd[i], data[i] = input().split()
    if cmd[i] == 'call':
        data[i] = int(data[i])
        if data[i] < 1 or data[i] > n:
            print('ERROR')
            exit()

print('OK')
for i in range(n):
    if cmd[i] == 'print':
        print(data[i], end = '')
    else:
        print(data[data[i] - 1], end = '')
print()
```

Подгруппа 5 (процесс не бесконечный)

Гарантируется, что в программе нет бесконечных циклов, но могут быть цепочки вызовов. Сначала проверяем ошибки (номер call вне диапазона). Затем выводим OK и для каждой команды по порядку определяем, какой символ она в итоге выведет. Для этого:

- начинаем с текущей команды;
- пока команда является call, переходим к команде, на которую она ссылается;
- когда доходим до команды print, выводим её символ.

Такой способ работает, потому что цепочки вызовов конечны и ведут к print.

Пример кода на Python:

```

n = int(input())
cmd = [0] * n
data = [0] * n
for i in range(n):
    cmd[i], data[i] = input().split()
    if cmd[i] == 'call':
        data[i] = int(data[i])
        if data[i] < 1 or data[i] > n:
            print('ERROR')
            exit()

print('OK')
for i in range(n):
    x = i
    while cmd[x] == 'call':
        x = data[x] - 1
    print(data[x], end = '')
print()

```

Подгруппа 6 (полные ограничения)

Здесь нужно обработать все возможные случаи: ошибки, бесконечность и нормальное выполнение. Сначала считываем команды и проверяем ошибки (call с номером вне диапазона). Затем для каждой команды, которая является call, проверяем, не приведёт ли она к бесконечному циклу. Для этого:

- создаём массив *used*, отмечаем текущую команду;
- идём по цепочке вызовов: переходим к команде, на которую ссылается текущая;
- если эта команда уже отмечена в *used* – значит, мы попали в бесконечный цикл, выводим INFINITY и завершаем программу;
- если команда print – останавливаем проверку, бесконечного цикла нет;
- иначе продолжаем по цепочке.

Если ни одного цикла не найдено, то выводим OK и результат. Результат получаем так же, как в подгруппе 5: для каждой команды идём по цепочке вызовов до print и выводим его символ.

Пример кода на Python:

```

n = int(input())
cmd = [0] * n
data = [0] * n
for i in range(n):
    cmd[i], data[i] = input().split()
    if cmd[i] == 'call':
        data[i] = int(data[i])
        if data[i] < 1 or data[i] > n:
            print('ERROR')
            exit()

for i in range(n):
    if cmd[i] == 'call':
        used = [False] * n
        used[i] = True
        j = data[i]
        while True:
            j -= 1
            if used[j]:
                print('INFINITY')
                exit(0)
            used[j] = True
            if cmd[j] == 'print':
                break
            j = data[j]

print('OK')
for i in range(n):
    x = i
    while cmd[x] == 'call':
        x = data[x] - 1
    print(data[x], end = '')
print()

```

Ф. Лабиринт

Подгруппа 1 (если путь есть, то не более 2 перемещений)

В этой подгруппе горшочек с золотом находится очень близко к стартовой позиции. Нужно проверить все возможные способы добраться до него за один или два шага. При этом важно, чтобы на пути не было стен.

Сначала найдём координаты А и G. Если они стоят рядом (по вертикали или горизонтали), то это один шаг – сразу YES. Если они стоят по диагонали (например, А в клетке (r, c) , G в $(r + 1, c + 1)$), то можно пройти за два шага через клетку $(r, c + 1)$ или $(r + 1, c)$. Нужно проверить, что хотя бы одна из этих клеток свободна. Если А и G находятся на расстоянии 2 по вертикали (А в (r, c) , G в $(r + 2, c)$), то можно пройти через среднюю клетку $(r + 1, c)$, если она свободна. То же самое для расстояния 2 по горизонтали. Во всех остальных случаях за два шага не добраться.

Пример кода на Python:

```

H, W = map(int, input().split())
lab = []
for i in range(H):
    lab.append(input())

# Находим координаты A и G
ax = ay = gx = gy = -1
for i in range(H):
    for j in range(W):
        if lab[i][j] == 'A':
            ax, ay = i, j
        elif lab[i][j] == 'G':
            gx, gy = i, j

# Проверка соседних клеток (расстояние 1)
if abs(ax - gx) + abs(ay - gy) == 1:
    print("YES")
# Проверка диагонали
elif abs(ax - gx) == 1 and abs(ay - gy) == 1:
    if lab[ax][gy] == '.' or lab[gx][ay] == '.':
        print("YES")
    else:
        print("NO")
# Проверка двух шагов по вертикали
elif abs(ax - gx) == 2 and ay == gy:
    midx = (ax + gx) // 2
    if lab[midx][ay] == '.':
        print("YES")
    else:
        print("NO")
# Проверка двух шагов по горизонтали
elif abs(ay - gy) == 2 and ax == gx:
    midy = (ay + gy) // 2
    if lab[ax][midy] == '.':
        print("YES")
    else:
        print("NO")
else:
    print("NO")

```

Подгруппа 2 (если путь есть, то не более 6 перемещений)

Теперь горшочек может быть чуть дальше, но не больше чем на 6 шагов. Можно просто перебрать все возможные пути с помощью рекурсии, но не углубляться больше чем на 6 шагов. Рекурсия пробует идти вверх, вниз, влево, вправо, если клетка не является стеной. Если на каком-то шаге мы попали в клетку с G, то выводим YES. Если глубина стала равна 6, то дальше не идём. Если ни один путь не привёл к G, то NO.

Пример кода на Python:

```
H, W = map(int, input().split())
lab = []
for i in range(H):
    lab.append(input())

# Находим A
ax = ay = -1
for i in range(H):
    for j in range(W):
        if lab[i][j] == 'A':
            ax, ay = i, j

found = False

def rec(r, c, depth):
    global found
    if found:
        return
    if lab[r][c] == 'G':
        found = True
        return
    if depth == 6:
        return
    # 4 направления: вниз, вверх, вправо, влево
    for dr, dc in [(1,0), (-1,0), (0,1), (0,-1)]:
        nr, nc = r + dr, c + dc
        if 0 <= nr < H and 0 <= nc < W and lab[nr][nc] != '#':
            rec(nr, nc, depth + 1)

rec(ax, ay, 0)
print("YES" if found else "NO")
```

Подгруппа 3 (полные ограничения)

Здесь горшочек может быть в любом месте лабиринта. По условию, лабиринт окружён стенами, а если путь из A в G существует, то он единственный. Можно воспользоваться правилом «обхода вдоль стены»: например, всё время держаться левой руки, касаясь стены. Начинаем из A и идём, например, сначала вверх (если не стена), а затем каждый раз поворачиваем налево относительно текущего направления. Если стены нет – идём прямо, иначе поворачиваем направо, и так далее. Такой обход гарантированно пройдёт по всем доступным коридорам и либо наткнётся на G, либо вернётся в A, сделав полный круг. Если G не найден, то до него невозможно добраться.