2 декабря 2012 года в г. Москве проходил второй (муниципальный, также называемый окружной) этап всероссийской олимпиады школьников по информатике.

В данной олимпиаде приняло участие 1578 учащихся 8-11 классов. Участникам олимпиады было предложено пять задач. Каждая задача оценивалась из 10 баллов, в зависимости от количества пройденных тестов. В задачах A, B, C было 5 тестов по 2 балла каждый, в задачах D и E было 10 тестов по 1 баллу каждый.

Макисмальный балл по всем задачам набрало 58 участников, больше нуля баллов набрал 1301 участник.

Статистика решения задач:

Баллы	A	B	C	D	E
10	468	733	388	427	96
9				34	6
8	214	79	57	13	5
7				37	12
6	267	35	36	48	10
5				10	38
4	195	19	33	9	16
3				14	8
2	93	63	134	58	97
1				17	51
0	341	649	930	911	1239

Ниже приведены условия задач, разбор их решений и примеры правильных решений. Все примеры приводятся на языке программирования Python версии 3, который наиболее подходит для записи алгоритмов ввиду свой простоты и лаконичности.

Во всех задачах входные данные вводятся с клавиатуры (стандартного ввода), результат выводится на экран (стандартный вывод).

Ограничение по времени работы программы во всех задачах: 1 секунда.

Задача A. Пирожные

Для праздничного чаепития необходимо купить n пирожных. В магазине продается всего два вида пирожных, причем пирожных одного вида осталось a штук, а пирожных другого вида осталось b штук. Пирожные одного вида считаются одинаковыми. Сколькими способами можно купить ровно n пирожных?

Формат входных данных

В первой строке входных данных записано число n — количество пирожных, которое нужно купить, во второй и третьей строке записаны числа a и b — количество пирожных каждого из двух видов, которые есть в магазине. Все числа — целые, от 1 до 100.

Формат выходных данных

Программа должна вывести одно целое число — количество различных способов купить n пирожных.

Примеры

Ввод	Вывод
5 3 10	4

Примечание

В примере из условия купить 5 пирожных можно 4 способами: 0 пирожных первого вида и 5 пирожных второго вида, 1 пирожное первого вида и 4 пирожных второго вида, 2 пирожных первого

вида и 3 пирожных второго вида, 3 пирожных первого вида и 2 пирожное второго вида. Больше способов нет, так как в магазине есть только 3 пирожных первого вида.

Разбор

Решение заключается в разборе различных случаев значений чисел n, a и b и может быть реализовано с использованием условных инструкций. Отдельно следует учесть случай, который вызвал затруднения у многих участников: когда сумма $a+b < n$, то есть в магазине недостаточно пирожных для совершения покупки. В этом случае программа должна вывести число 0 — хотя это не было явно прописано в условии, но количество способов совершить покупку равно 0. Участники, которые не разобрали этот случай, получили 8 баллов за эту задачу.

Дальнейшие случаи проще разобрать следующим образом. Если $a > n$, то это означает, что пирожных вида a в магазине больше, чем требуется, и мы можем считать, что их в точности n, выполнив присваивание $a = n$. Это не повлияет на ответ задачи, так как больше n пирожных первого вида купить нельзя. Аналогично присвоим $b = n$ если $b > n$.

Теперь посчитаем, какое минимальное и максимальное число пирожных первого вида может быть куплено. Максимальное число равно a (так как $a+b \geqslant n$, то при покупке a пирожных первого вида в магазине останется достаточное число пирожных второго вида, чтобы купить оставшиеся $n-a$ пирожных). А минимальное число равно $n-b$ (в этом случае покупаются все пирожные второго вида, а оставшиеся пирожные покупаются первого вида). Общее число вариантов равно $a-(n-b)+1 = a+b-n+1$.

Вот пример такого решения на языке Питон:

```
n = int(input())
a = int(input())
b = int(input())
if a + b < n:
    print(0)
else:
    if a > n:
        a = n
    if b > n:
        b = n
    print(a + b + 1 - n)
```

Но поскольку все входные числа в этой задаче были не очень большими, задача допускала и более простое решение с использованием полного перебора вариантов. Просто переберем все возможные способы купить пирожные первого вида в цикле по переменной $i = 0...a$ и вложенным циклом по переменной $j = 0...b$ переберем все возможные способы приобрести пирожные второго вида. Если $i + j = n$, то увеличим ответ на 1.

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

Задача B. Переливания

Имеется 10 колб с водой и известен объем воды в каждой из них. За одно «касание» можно взять одну колбу и часть воды (или всю воду) из этой колбы разлить по одной или нескольким другим

колбам в любом количестве. За какое наименьшее количество «касаний» можно уравнять объемы воды во всех колбах? Каждая колба может вместить любой объем воды.

Формат входных данных

Программа получает на вход 10 целых чисел a_i, каждое записанное в отдельной строке — объем воды в каждой из колб. Все числа — целые, от 0 до 100.

Формат выходных данных

Выведите одно целое число — минимальное количество «касаний», за которое можно уравнять объемы воды во всех колбах.

Примеры

Ввод	Вывод
30 26 2 3 4 5 6 7 8 9	2

Примечание

В примере можно из первой колбы перелить 20 во вторую, оставляя в первой колбе 10. Затем из второй колбы разлить воду по всем остальным колбам так, чтобы в каждой из колб оказалось по 10.

Разбор

Считаем входные данные, запишем объемы воды в колбах в массив `V` и подсчитаем среднее значение всех объемов в колбах `Average`. Если в какой-то колбе налито больше `Average` воды, то обязательно придется дотронуться до этой колбы, чтобы вылить из нее излишки воды. Оказывается, что достаточно дотронуться только до таких колб, если воду из этих колб разливать по оставшимся колбам так, чтобы объем в них не превышал значения `Average`. Таким образом, задача сводится к нахождению среднего арифметического значений элементов массива `V` и подсчета числа элементов массива `V`, которые больше среднего арифметического `Average`.

```
N = 10
# Считывание входных данных в список V
# и подсчет среднего арифметического
V = [0] * N
Sum = 0
for i in range(N):
    V[i] = int(input())
    Sum += V[i]
Average = Sum / N
ans = 0
for i in range(N):
    if V[i] > Average:
        ans += 1
print(ans)
```

Поскольку массив содержит ровно 10 элементов, то участники олимпиады, не знакомые с массивами, могли использовать 10 различных переменных для хранения объемов воды и 10 условных инструкций вместо цикла.

Задача C. Чехарда

Дорожка замощена плитками в один ряд, плитки пронумерованы числами от 1 до 1000. На плитках с номерами A, B и C ($A < B < C$) сидят три кузнечика, которые играют в чехарду по следующим правилам:

1. На одной плитке может находиться только один кузнечик.

2. За один ход один из двух крайних кузнечиков (то есть с плитки A или с плитки C) может перепрыгнуть через среднего кузнечика (плитка B) и встать на плитку, которая находится ровно посередине между двумя оставшимися кузнечиками (то есть между B и C или A и B соответственно). Если между двумя оставшимися кузнечиками находится чётное число плиток, то он может выбрать любую из двух центральных плиток.

Например, если кузнечики первоначально сидели на плитках номер 1, 5, 10, то первым ходом кузнечик с плитки номер 10 может перепрыгнуть на плитку номер 3 (она находится посередине между 1 и 5), или кузнечик с плитки номер 1 может перепрыгнуть на плитку номер 7 или 8 (эти две плитки находятся посередине между плитками 5 и 10).

Даны три числа: A, B, C. Определите, какое наибольшее число ходов может продолжаться игра.

Формат входных данных

Программа получает на вход три целых числа A, B и C ($1 \leqslant A < B < C \leqslant 1000$), записанных в отдельных строках.

Формат выходных данных

Выведите одно число — наибольшее количество ходов, которое может продолжаться игра.

Примеры

Ввод	Вывод
1 4 6	2

Примечание

В примере сначала кузнечик с плитки №6 прыгает на плитку №3. Затем кузнечик с плитки №4 прыгает на плитку №2.

Разбор

Эта задача решается при помощи так называемого “жадного” алгоритма: кузнечики должны прыгать так, чтобы расстояние между ними оставалось максимально возможным. На первом шаге расстояния между соседними кузнечиками равны $B-A$ и $C-B$. Соответственно, если $B-A > C-B$, то должен прыгать третий кузнечик, иначе должен прыгать первый кузнечик.

Таким образом, на первом шаге необходимо выбрать наибольший из двух отрезков длины $B-A$ и $C-B$. Запишем длину этого отрезка в переменную d. Далее на каждом шаге третий кузнечик прыгает в середину отрезка длины d, поэтому значение d сокращается в два раза. При этом если d — четное, то d просто делится на 2, а если нечетное, то отрезок делится на две части, длины которых отличаются на 1 и необходимо выбрать большую из этих частей. То есть необходимо поделить d на 2 с округлением вверх (в случае нечетного d это как раз даст значение большей из двух частей).

В языке Питон для целочисленного деления (с отбрасыванием дробной части, то есть с округлением вниз, это аналог операции `div` в Паскале) используется оператор `//`. Для деления числа на 2 с округлением вверх можно использовать следующее присваивание: `d = (d + 1) // 2`.

Будем делить значение d на 2 с округлением вверх, подсчитывая количество выполненных делений. Цикл заканчивается при $d = 1$, что означает, что после очередного прыжка два кузнечика сидят на соседних клетках (расстояние между ними равно 1).

```
a = int(input())
b = int(input())
c = int(input())
d = max(b - a, c - b)
ans = 0
while d > 1:
    ans += 1
    d = (d + 1) // 2
print(ans)
```

Также можно заметить, что поскольку в цикле значение d делится на 2, то количество итераций цикла равно логарифму числа d по основанию 2, что позволяет записать решение без цикла, но с вычислением логарифма.

```
from math import log
a = int(input())
b = int(input())
c = int(input())
d = max(b - a, c - b)
if d == 1:
    print(0)
else:
    print(int(log(d - 1, 2) + 1))
```

Задача D. Телефонные номера

Телефонные номера в адресной книге мобильного телефона имеют один из следующих форматов:
+7<код><номер>
8<код><номер>
<номер>
где <номер> — это семь цифр, а <код> — это три цифры или три цифры в круглых скобках. Если код не указан, то считается, что он равен 495. Кроме того, в записи телефонного номера может стоять знак "-" между любыми двумя цифрами (см. пример).

На данный момент в адресной книге телефона Васи записано всего три телефонных номера, и он хочет записать туда еще один. Но он не может понять, не записан ли уже такой номер в телефонной книге. Помогите ему!

Два телефонных номера совпадают, если у них равны коды и равны номера. Например, +7(916)0123456 и 89160123456 — это один и тот же номер.

Формат входных данных

В первой строке входных данных записан номер телефона, который Вася хочет добавить в адресную книгу своего телефона. В следующих трех строках записаны три номера телефонов, которые уже находятся в адресной книге телефона Васи. Гарантируется, что каждая из записей соответствует одному из трех приведенных в условии форматов.

Формат выходных данных

Для каждого телефонного номера в адресной книге выведите YES (заглавными буквами), если он совпадает с тем телефонным номером, который Вася хочет добавить в адресную книгу или NO (заглавными буквами) в противном случае.

Примеры

Ввод	Вывод
8(495)430-23-97 +7-4-9-5-43-023-97 4-3-0-2-3-9-7 8-495-430	YES YES NO

Разбор

В этой задаче требовалось реализовать несложный алгоритм обработки текстовых данных. Так как телефонные номера сравниваются только по совпадению трехзначного кода и семизначного номера, то удобно каждый номер хранить в виде строки из 10 символов: 3 цифры кода и 7 цифр номера. Тогда номера можно будет сравнивать просто как строки.

Реализуем функцию `normalize`, которая по данному телефонному номеру получает его представление в виде строки из 10 символов. Функция будет получать исходную строку в виде параметра и возвращать строку, являющуюся "нормализованной" формой телефонного номера.

Функция будет устроена так. Сначала она из исходной строки оставляет только символы, являющиеся цифрами. Для этого заводится новая строка `ans` и перебираются все символы исходной строки. Если символ является цифрой, то он добавляется к строке `ans`. В результате получится строка из 11 цифр (если номер содержал код, тогда первая цифра это либо 7, либо 8) или из 7 цифр. Если получилось 11 цифр, то необходимо удалить из строки первый символ и вернуть результат, а если получилось 7 цифр, то необходимо в начало строки добавить строку с кодом `'495'`.

Основная часть программы считывает первый номер, вызывает для него функцию `normalize` и сохраняет результат в переменной. Далее в цикле три раза считывается новый телефонный номер, для него вызывается функция `normalize` и результат сравнивается с сохраненным номером. Если номера равны, то выводится `YES`, иначе выводится `NO`.

```
def normalize(s):
    ans = ''
    for d in s:
        if '0' <= d <= '9':
            ans += d
    if len(ans) == 11:
        return ans[1:]
    else:
        return '495' + ans

s = normalize(input())
for i in range(3):
    t = normalize(input())
    if s == t:
        print('YES')
    else:
        print('NO')
```

Задача E. Простой квадрат

У Пети имеется игровое поле размером 3×3, заполненное числами от 1 до 9. В начале игры он может поставить фишку в любую клетку поля. На каждом шаге игры разрешается перемещать фишку в любую соседнюю по стороне клетку, но не разрешается посещать одну и ту же клетку дважды. Петя внимательно ведет протокол игры, записывая в него цифры в том порядке, в котором фишка посещала клетки. Пете стало интересно, какое максимальное число он может получить в протоколе. Помогите ему ответить на этот вопрос.

Формат входных данных

Входной файл содержит описание поля — 3 строки по 3 целых числа, разделенных пробелами. Гарантируется, что все девять чисел различны и лежат в диапазоне от 1 до 9.

Формат выходных данных

Выведите одно целое число — максимальное число, которое могло получиться в протоколе при игре на данном поле. Ответ можно выводить не в виде числа, а в виде строки или в виде последовательности отдельных цифр (но не разделяя их пробелами).

Примеры

Ввод	Вывод
1 2 3 4 5 6 7 8 9	987456321

Разбор

Для решения этой задачи необходимо было организовать перебор возможных маршрутов, что можно слелать по-разному.

Заметим, что соблюдая правила игры на квадрате всегда можно нарисовать маршрут длины 9, поэтому ответ всегда будет 9-значным. Но поскольку для строк длины 9, составленных из цифр от 1 до 9, и для 9-значных чисел из этих же цифр лексикографический и числовой порядок сортировки совпадают, удобней работать со строками, а не с числами.

Будем хранить квадрат в виде квадратного двумерного массива `A` размера 3×3:

```
A[0][0]  A[0][1]  A[0][2]
A[1][0]  A[1][1]  A[1][2]
A[2][0]  A[2][1]  A[2][2]
```

В массиве будут храниться символы (в языке Питон это строки длины 1).

Ниже приведен вариант решения, в котором перебираются все возможные маршруты на квадрате при помощи “перебора с возвратом” (англ. backtracking).

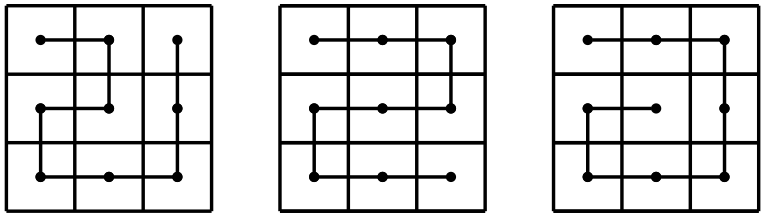
```
Answer = ''

A = [input().split() for i in range(3)]

Visited = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

def Backtracking(prefix, x, y):
    global Answer
    prefix += A[x][y]
    if len(prefix) == 9 and prefix > Answer:
        Answer = prefix
    Visited[x][y] = 1
    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
        if 0 <= x + dx <= 2 and 0 <= y + dy <= 2 and Visited[x + dx][y + dy] == 0:
            Backtracking(prefix, x + dx, y + dy)
    Visited[x][y] = 0

for x in range(3):
    for y in range(3):
        Backtracking('', x, y)
print(Answer)
```

Смысл перебора с возвратом следующий. Заводится двумерный массив `Visited` в котором делается пометка 1, если данная клетка была посещена и заходить в нее больше нельзя. Если же клетка доступна для посещения, то в ней делается пометка 0. Алгоритм перебора реализован в виде рекурсивной процедуры `Backtracking`, которой в качестве параметров передается строка `prefix` — последовательность уже выписанных чисел и координаты `x` и `y` новой клетки, которая добавляется к маршруту. Данный алгоритм добавляет к строке `prefix` цифру, записанную в клетке `A[x][y]` и проверяет, не получился ли при этом ответ больший, чем сохранен в глобальной переменной `Answer`.

Затем клетка помечается, как посещенная (`Visited[x][y] = 1`). После этого перебираются все соседние клетки в цикле по двум переменным `dx` и `dy`, пара `(dx, dy)` задает направление перемещения, то есть следующая клетка перебираемого маршрута будет иметь координаты `(x + dx, y + dy)`. Проверяется условие невыхода за границы квадрата и если следующая клетка свободна, то функция `Backtracking` запускается из следующей клетки `(x + dx, y + dy)`. Когда функция `Backtracking` выходит из клетки `(x, y)`, нужно отметить ее доступной для посещения (`Visited[x][y] = 0`).

Далее перебираются все клетки квадрата и из каждой клетки запускается функция `Backtracking` для перебора всех маршрутов, начинающихся в этой клетке.

Алгоритм полного перебора с возвратом в данном случае позволяет полностью решить задачу, но во многих случаях он оказывается неэффективен из-за слишком большого объема перебора. В данной задаче можно сократить перебор, если заметить, что существует всего три принципально различные формы маршрута, проходящего через все клетки квадрата.



Следующее решение основано на идее перебора всех этих форм маршрутов. Маршрут может начинаться либо в угловой клетке, либо в центре квадрата. Если он начинается в угловой клетке, то наилучший из всех маршрутов будет начинаться в том углу, в котором записано наибольшее из четырех чисел, записанных во всех углах. Поэтому будем поворачивать квадрат до тех пор, пока наибольшая цифра из четырех угловых клеток не окажется в левом верхнем углу (клетке `A[0][0]`). Для поворота реализуем функцию `Rotate`.

Теперь если наилучший маршрут начинается в угловой клетке, то он начинается в клетке `A[0][0]`. Следующая клетка может быть одна из двух соседних с ней `A[1][0]` или `A[0][1]`. Сравним эти два значения, и если `A[1][0]` окажется больше, отразим квадрат относительно главной диагонали при помощи функции `Mirror`. Тем самым мы добьемся того, что если наилучший маршрут начинается в угловой клетке, то он начинается в клетке `A[0][0]`, затем идет в клету `A[0][1]`. Таких маршрутов всего четыре: три изображены на рисунке, а четвертый маршрут — это маршрут на первом рисунке, но записанный в обратном порядке. Числа вдоль всех этих четырех маршрутов выпишем в переменные `P1`, `P2`, `P3`, `P4` (переменные будут строковыми).

Теперь рассмотрим маршруты, начинающиеся в центре квадрата. Такой маршрут должен из центра перейти в одну из четырех соседних клеток, причем в ту, в которой записано наибольшее число. Опять используя фунцию `Rotate` добьемся того, что наибольшее из чисел, записанных в соседних с центром клетках, будет записано на верхней стороне квадрата, то есть в клетке `A[0][1]`. Маршрутов, начинающихся в `A[1][1]`, затем проходящих через `A[0][1]` всего два — это две спирали, закрученных в разных направления. Числа вдоль этих двух маршрутов выпишем в строковые переменные `P5` и `P6`.

Затем выведем наибольшую из всех строк `P1`, `P2`, `P3`, `P4`, `P5`, `P6`.

```
A = [input().split() for i in range(3)]

# Функция поворачивает квадрат против часовой стрелки
def Rotate():
    A[0][0], A[0][2], A[2][2], A[2][0] = A[0][2], A[2][2], A[2][0], A[0][0]
    A[0][1], A[1][2], A[2][1], A[1][0] = A[1][2], A[2][1], A[1][0], A[0][1]

# Функция отражает квадрат относительно главной диагонали
def Mirror():
    A[0][1], A[1][0] = A[1][0], A[0][1]
```

```
    A[0][2], A[2][0] = A[2][0], A[0][2]
    A[1][2], A[2][1] = A[2][1], A[1][2]

# Ищем лучший путь из угла

# Максимальное значение, записанное в углах
MaxInAngles = max(A[0][0], A[0][2], A[2][2], A[2][0])

# Поворачиваем квадрат, пока наибольшее из значений в углах
# не окажется в левом верхнем углу
while A[0][0] != MaxInAngles:
    Rotate()

# Сравним соседние клетки в левом верхнем углу, при необходимости
# отразим квадрат так, чтомы следущая клетка маршрута была A[0][1]
if A[1][0] > A[0][1]:
    Mirror()

# Все 4 возможных маршрута, начинающихся в A[0][0], затем в A[0][1]
P1 = A[0][0]+A[0][1]+A[1][1]+A[1][0]+A[2][0]+A[2][1]+A[2][2]+A[1][2]+A[0][2]
P2 = A[0][0]+A[0][1]+A[0][2]+A[1][2]+A[2][2]+A[2][1]+A[1][1]+A[1][0]+A[2][0]
P3 = A[0][0]+A[0][1]+A[0][2]+A[1][2]+A[1][1]+A[1][0]+A[2][0]+A[2][1]+A[2][2]
P4 = A[0][0]+A[0][1]+A[0][2]+A[1][2]+A[2][2]+A[2][1]+A[2][0]+A[1][0]+A[1][1]

# Теперь будем искать лучший маршут из середины квадрата

# Максимальное значение, записанное на сторонах
MaxInSides = max(A[0][1], A[1][2], A[2][1], A[1][0])

# Поворачиваем квадрат, пока наибольшее из значений на сторонах
# не окажется на верхней стороне (A[0][1])
while A[0][1] != MaxInSides:
    Rotate()

# Теперь рассмотрим две спирали, начинающиеся в центре, затем
# идущих в клетку A[0][1]

P5 = A[1][1]+A[0][1]+A[0][2]+A[1][2]+A[2][2]+A[2][1]+A[2][0]+A[1][0]+A[0][0]
P6 = A[1][1]+A[0][1]+A[0][0]+A[1][0]+A[2][0]+A[2][1]+A[2][2]+A[1][2]+A[0][2]

# Выведем максимум из всех полученных маршрутов
print(max(P1, P2, P3, P4, P5, P6))
```

Подобный алгоритм перебора можно реализовать и без использования циклов и вспомогательных функций, и даже не используя двумерные массивы (можно хранить числа, записанные в клетках квадрата, в девяти различных переменных).