*"""  
The logic is basically to use dynamic programming and create a matrix that stores amount of gold in the rectangle having  
corners at (0, 0) and (x, y) in element matrix[x][y]. Now a polygon can be represented as a combination of several such  
rectangles using the inclusion-exclusion principle. We have to include the area till first vertex excluded the area till  
second, include till third and so on (Refer the code). This property is very difficult to observe and that makes the  
problem challenging. The matrix itself can be computed using dynamic programming as shown. The number of queries can be  
increased to cause TLEs and failure of many other complicated algorithms that do not use the above property. The above  
property makes the complexity O(number of elements in matrix + number of queries \* number of vertices).  
"""***from** math **import** hypot, floor *# for distance and rounding down*mine = [[0 **for** \_ **in** range(100)] **for** \_ **in** range(100)] *# stores the amount of gold for each unit*m = [] *# stores meteorite coordinates***for** i **in** range(int(input(**'Enter the number of meteors and their coordinates: '**))): *# input meteorite coordinates* m.append([float(i) **for** i **in** input().split(**' '**)])  
  
**for** i **in** range(len(mine)): *# calculating the amount of gold* **for** j **in** range(len(mine)):  
 **for** k **in** m:  
 mine[i][j] += 1000 / (hypot(i - k[0] + 0.5, j - k[1] + 0.5) + 1)  
 mine[i][j] = floor(mine[i][j])  
  
a = [[-1 **for** \_ **in** range(100)] **for** \_ **in** range(100)] *# matrix for DP***def** find\_area(x, y): *# finding amount from (0, 0) to (x, y) using lazy DP* **if** a[x][y] != -1:  
 **return** a[x][y]  
 **if** x **is** 0 **or** y **is** 0:  
 a[x][y] = 0  
 **return** 0  
 *# applying inclusion exclusion principle* a[x][y] = find\_area(x - 1, y) + find\_area(x, y - 1) - find\_area(x - 1, y - 1) + mine[x - 1][y - 1]  
 **return** a[x][y]  
  
  
**def** poly\_area(): *# solving every query* ar = 0  
 o = 1  
 **for** \_ **in** range(int(input(**'Enter the number of vertices and their coordinates: '**))):  
 p, q = [int(\_) **for** \_ **in** input().split(**' '**)]  
 ar += o \* find\_area(p, q)  
 o \*= -1  
 **return** abs(ar)  
  
  
**for** \_ **in** range(int(input(**'Enter the number of queries: '**))):  
 print(poly\_area())