Running with Bunnies  
====================  
  
You and your rescued bunny prisoners need to get out of this collapsing death trap of a space station - and fast! Unfortunately, some of the bunnies have been weakened by their long imprisonment and can't run very fast. Their friends are trying to help them, but this escape would go a lot faster if you also pitched in. The defensive bulkhead doors have begun to close, and if you don't make it through in time, you'll be trapped! You need to grab as many bunnies as you can and get through the bulkheads before they close.   
  
The time it takes to move from your starting point to all of the bunnies and to the bulkhead will be given to you in a square matrix of integers. Each row will tell you the time it takes to get to the start, first bunny, second bunny, ..., last bunny, and the bulkhead in that order. The order of the rows follows the same pattern (start, each bunny, bulkhead). The bunnies can jump into your arms, so picking them up is instantaneous, and arriving at the bulkhead at the same time as it seals still allows for a successful, if dramatic, escape. (Don't worry, any bunnies you don't pick up will be able to escape with you since they no longer have to carry the ones you did pick up.) You can revisit different spots if you wish, and moving to the bulkhead doesn't mean you have to immediately leave - you can move to and from the bulkhead to pick up additional bunnies if time permits.  
  
In addition to spending time traveling between bunnies, some paths interact with the space station's security checkpoints and add time back to the clock. Adding time to the clock will delay the closing of the bulkhead doors, and if the time goes back up to 0 or a positive number after the doors have already closed, it triggers the bulkhead to reopen. Therefore, it might be possible to walk in a circle and keep gaining time: that is, each time a path is traversed, the same amount of time is used or added.  
  
Write a function of the form solution(times, time\_limit) to calculate the most bunnies you can pick up and which bunnies they are, while still escaping through the bulkhead before the doors close for good. If there are multiple sets of bunnies of the same size, return the set of bunnies with the lowest prisoner IDs (as indexes) in sorted order. The bunnies are represented as a sorted list by prisoner ID, with the first bunny being 0. There are at most 5 bunnies, and time\_limit is a non-negative integer that is at most 999.  
  
For instance, in the case of  
[  
[0, 2, 2, 2, -1], # 0 = Start  
[9, 0, 2, 2, -1], # 1 = Bunny 0  
[9, 3, 0, 2, -1], # 2 = Bunny 1  
[9, 3, 2, 0, -1], # 3 = Bunny 2  
[9, 3, 2, 2, 0], # 4 = Bulkhead  
]  
and a time limit of 1, the five inner array rows designate the starting point, bunny 0, bunny 1, bunny 2, and the bulkhead door exit respectively. You could take the path:  
  
Start End Delta Time Status  
- 0 - 1 Bulkhead initially open  
0 4 -1 2  
4 2 2 0  
2 4 -1 1  
4 3 2 -1 Bulkhead closes  
3 4 -1 0 Bulkhead reopens; you and the bunnies exit  
  
With this solution, you would pick up bunnies 1 and 2. This is the best combination for this space station hallway, so the answer is [1, 2].

-- Python cases --  
Input:  
solution.solution([ ], 1)  
Output:  
    [1, 2]  
  
Input:  
solution.solution([[0, 1, 1, 1, 1], [1, 0, 1, 1, 1], [1, 1, 0, 1, 1], [1, 1, 1, 0, 1], [1, 1, 1, 1, 0]], 3)  
Output:  
    [0, 1]

Bringing a Gun to a Guard Fight  
===============================  
  
Uh-oh - you've been cornered by one of Commander Lambdas elite guards! Fortunately, you grabbed a beam weapon from an abandoned guard post while you were running through the station, so you have a chance to fight your way out. But the beam weapon is potentially dangerous to you as well as to the elite guard: its beams reflect off walls, meaning you'll have to be very careful where you shoot to avoid bouncing a shot toward yourself!  
  
Luckily, the beams can only travel a certain maximum distance before becoming too weak to cause damage. You also know that if a beam hits a corner, it will bounce back in exactly the same direction. And of course, if the beam hits either you or the guard, it will stop immediately (albeit painfully).   
  
Write a function solution(dimensions, your\_position, guard\_position, distance) that gives an array of 2 integers of the width and height of the room, an array of 2 integers of your x and y coordinates in the room, an array of 2 integers of the guard's x and y coordinates in the room, and returns an integer of the number of distinct directions that you can fire to hit the elite guard, given the maximum distance that the beam can travel.  
  
The room has integer dimensions [1 < x\_dim <= 1250, 1 < y\_dim <= 1250]. You and the elite guard are both positioned on the integer lattice at different distinct positions (x, y) inside the room such that [0 < x < x\_dim, 0 < y < y\_dim]. Finally, the maximum distance that the beam can travel before becoming harmless will be given as an integer 1 < distance <= 10000.  
  
For example, if you and the elite guard were positioned in a room with dimensions [3, 2], your\_position [1, 1], guard\_position [2, 1], and a maximum shot distance of 4, you could shoot in seven different directions to hit the elite guard (given as vector bearings from your location): [1, 0], [1, 2], [1, -2], [3, 2], [3, -2], [-3, 2], and [-3, -2]. As specific examples, the shot at bearing [1, 0] is the straight line horizontal shot of distance 1, the shot at bearing [-3, -2] bounces off the left wall and then the bottom wall before hitting the elite guard with a total shot distance of sqrt(13), and the shot at bearing [1, 2] bounces off just the top wall before hitting the elite guard with a total shot distance of sqrt(5).  
  
Languages  
=========  
  
To provide a Java solution, edit Solution.java  
To provide a Python solution, edit solution.py  
  
Test cases  
==========  
Your code should pass the following test cases.  
Note that it may also be run against hidden test cases not shown here.  
  
-- Java cases --  
Input:  
Solution.solution([3,2], [1,1], [2,1], 4)  
Output:  
    7  
  
Input:  
Solution.solution([300,275], [150,150], [185,100], 500)  
Output:  
    9  
  
-- Python cases --  
Input:  
solution.solution([3,2], [1,1], [2,1], 4)  
Output:  
    7  
  
Input:  
solution.solution([300,275], [150,150], [185,100], 500)  
Output:  
    9

Disorderly Escape  
=================  
  
Oh no! You've managed to free the bunny prisoners and escape Commander Lambdas exploding space station, but her team of elite starfighters has flanked your ship. If you dont jump to hyperspace, and fast, youll be shot out of the sky!  
  
Problem is, to avoid detection by galactic law enforcement, Commander Lambda planted her space station in the middle of a quasar quantum flux field. In order to make the jump to hyperspace, you need to know the configuration of celestial bodies in the quadrant you plan to jump through. In order to do \*that\*, you need to figure out how many configurations each quadrant could possibly have, so that you can pick the optimal quadrant through which youll make your jump.   
  
There's something important to note about quasar quantum flux fields' configurations: when drawn on a star grid, configurations are considered equivalent by grouping rather than by order. That is, for a given set of configurations, if you exchange the position of any two columns or any two rows some number of times, youll find that all of those configurations are equivalent in that way - in grouping, rather than order.  
  
Write a function solution(w, h, s) that takes 3 integers and returns the number of unique, non-equivalent configurations that can be found on a star grid w blocks wide and h blocks tall where each celestial body has s possible states. Equivalency is defined as above: any two star grids with each celestial body in the same state where the actual order of the rows and columns do not matter (and can thus be freely swapped around). Star grid standardization means that the width and height of the grid will always be between 1 and 12, inclusive. And while there are a variety of celestial bodies in each grid, the number of states of those bodies is between 2 and 20, inclusive. The solution can be over 20 digits long, so return it as a decimal string. The intermediate values can also be large, so you will likely need to use at least 64-bit integers.  
  
For example, consider w=2, h=2, s=2. We have a 2x2 grid where each celestial body is either in state 0 (for instance, silent) or state 1 (for instance, noisy). We can examine which grids are equivalent by swapping rows and columns.  
  
00  
00  
  
In the above configuration, all celestial bodies are "silent" - that is, they have a state of 0 - so any swap of row or column would keep it in the same state.  
  
00 00 01 10  
01 10 00 00  
  
1 celestial body is emitting noise - that is, has a state of 1 - so swapping rows and columns can put it in any of the 4 positions. All four of the above configurations are equivalent.  
  
00 11  
11 00  
  
2 celestial bodies are emitting noise side-by-side. Swapping columns leaves them unchanged, and swapping rows simply moves them between the top and bottom. In both, the \*groupings\* are the same: one row with two bodies in state 0, one row with two bodies in state 1, and two columns with one of each state.  
  
01 10  
01 10  
  
2 noisy celestial bodies adjacent vertically. This is symmetric to the side-by-side case, but it is different because there's no way to transpose the grid.  
  
01 10  
10 01  
  
2 noisy celestial bodies diagonally. Both have 2 rows and 2 columns that have one of each state, so they are equivalent to each other.  
  
01 10 11 11  
11 11 01 10  
  
3 noisy celestial bodies, similar to the case where only one of four is noisy.  
  
11  
11  
  
4 noisy celestial bodies.  
  
There are 7 distinct, non-equivalent grids in total, so solution(2, 2, 2) would return 7.  
  
Languages  
=========  
  
To provide a Java solution, edit Solution.java  
To provide a Python solution, edit solution.py  
  
Test cases  
==========  
Your code should pass the following test cases.  
Note that it may also be run against hidden test cases not shown here.  
  
-- Java cases --  
Input:  
Solution.solution(2, 3, 4)  
Output:  
    430  
  
Input:  
Solution.solution(2, 2, 2)  
Output:  
    7  
  
-- Python cases --  
Input:  
solution.solution(2, 3, 4)  
Output:  
    430  
  
Input:  
solution.solution(2, 2, 2)  
Output:  
    7

Sh Sw

y1 = [1, 1] y2 = [1, 1]

y1,1 = 1 y2,1 = 1 *gcd*(y1,1, y2,1) = 1

y1,1 = 1 y2,2 = 1 *gcd*(y1,1, y2,2) = 1

y1,2 = 1 y2,1 = 1 *gcd*(y1,2, y2,1) = 1

y1,2 = 1 y2,2 = 1 *gcd*(y1,2, y2,2) = 1

exponent summation = 4

y1 = [1, 1] y2 = [2]

y1,1 = 1 y2,1 = 2 *gcd*(y1,1, y2,1) = 1

y1,2 = 1 y2,1 = 2 *gcd*(y1,2, y2,1) = 1

exponent summation = 2

y1 = [2] y2 = [1, 1]

y1,1 = 2 y2,1 = 1 *gcd*(y1,1, y2,1) = 1

y1,1 = 2 y2,2 = 1 *gcd*(y1,1, y2,2) = 1

exponent summation = 2

y1 = [2] y2 = [2]

y1,1 = 2 y2,1 = 2 *gcd*(y1,1, y2,1) = 2

exponent summation = 2