**1 Instructions:**

1. Complete the implementation in starCounter.cpp. This program must process an input set of nn points in O(n)O(n) average case time. (Pay particular attention to the sections marked with //\*\*\*)
2. Use the button below to submit your starCounter.cpp file:

Top of Form

Bottom of Form

**2 Problem Descriptions:**

In massive simulations of star systems, we don’t want to have to model the gravitational effects of pairs of bodies that are too far away from each other, because that will take up excess computing power (and their effects on each other are negligible). We want to consider the effects two objects have on each other only if the Euclidean distance between the two is less than dd.

Given a list of nn points in space, how many have a distance of less than dd apart from each other?

**2.1 Input**

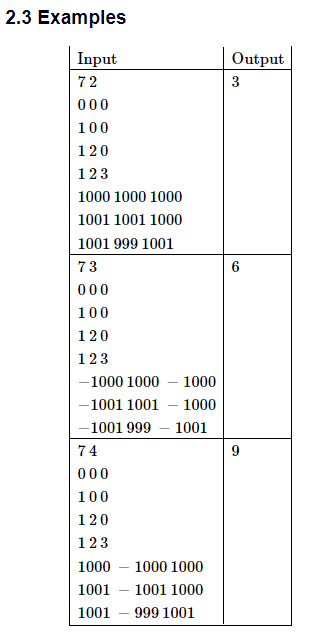
Input is accepted from the standard input (cin).

A test input will begin with a line with two integers, nn (2≤n≤100,0002≤n≤100,000) and dd (0<d≤1090<d≤109), where nn is the number of points, and dd is the desired maximum distance.

On each of the following nn lines will be three integers xx, yy and zz, (−109≤x,y,z≤109)(−109≤x,y,z≤109) which are the (x,y,z)(x,y,z) coordinates of one point. Because stars systems are sparse and not uniformly distributed through space, it is guaranteed that no more than 100,000100,000 pairs of points will be within dd of each other.

**2.2 Output**

Output is sent to standard out. The program will print a single integer indicating the number of unique pairs of points that are less than dd apart from each other.



**3 Notes**

Of, course this would be a very simple (CS150) problem if we could compare every point to every other point. But that would be an O(n2)O(n2) solution, and if we are allowing up to 100,000 points, that might be too slow.

So, how do we get this down to O(n)O(n)? Obviously, we have to touch each of the nn points at least once, so whatever we do to each point, we can’t afford to spend more then O(1)O(1) effort per point.

So we take advantage of the fact that space is very, very empty. Stars are not packed close together. (We are guaranteed that no more than 100,000 pairs of “close” points exist. Without that guarantee, we would be looking at potentially n2=1010n2=1010 pairs.

Imagine that we divide space up into cubes of size dd by dd by dd. We can index these cubes by taking the coordinates of the corner closest to (0,0,0)(0,0,0) divided by dd. For example, if d=5d=5, then cube [0,0,0] would be the cube with a diagonal running from (0,0,0)(0,0,0)to (5,5,5)(5,5,5), cube [1,0,0] would be the cube with a diagonal running from (5,0,0)(5,0,0) to (10,5,5)(10,5,5), and so on.

Obviously, each star sits in exactly one cube. Moreover, because the stars are not close together, there will be few cubes with more than one star, fewer with more than two, still fewer with more than three, and so on. In fact, since we have only 100,000 stars distributed in a space compose of as many as (2∗109)3=8∗1027(2∗109)3=8∗1027 cubes, we know that the chance of any cube being non-empty can be as small as 1058∗10271058∗1027 or roughly 0.000000000000000000001

Now, suppose that we have a star at point PP in cube [i,j,k]. Where would we look to find other stars within distance dd of PP? Obviously any other stars inside that cube *could* be within dd. (“could be”, but if you think about a pair of stars at, for example, opposite corners of the cube, it’s clear that not all pairs of stars in the cube would actually be close enough to each other.

* So we will have to compare PP against all the stars in its cube.
* It’s also possible that the cubes adjacent to [i,j,k] will have points within distance dd of PP.
  + So we will have to check the points in adjacent cubes as well.

All told, we would need to look at the points in 33=2733=27 cubes for each point PP. But, remember, most of those cubes are going to be empty. (The chances of a non-empty cube are approximately 10−2210−22.)

So, our solution looks roughly like:

1. Divide space up into cubes. We will associate a list of points with each cube. Initially, these lists are empty.
2. Make a pass over all the points, adding each to a list of points in the appropriate cube.
3. Make a second pass over all the points. For each point PP, compare PP to every other point in its own cube and in the the 26 cubes that touch PP’s cube. Count the number of these that are actually within distance dd of PP.

We could imagine doing this with a massive, 3-dimensional array of cubes. But there would be too many (possibly more than to 10271027) cubes to actually store in memory. And most of that storage would be “wasted” because the vast majority of the cubes will be empty.

Hash tables are very good at handling “sparse” data structures. We can use a hash-based multimap to map a cube [i,j,k] index onto a list of points in that cube.