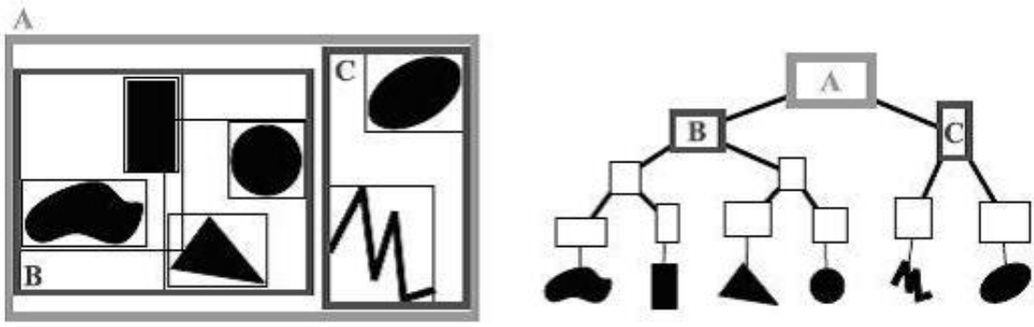
A **bounding volume hierarchy (BVH)** is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree.



(An example of a bounding volume hierarchy using rectangles as bounding volumes)

BVH can be used as an accelerating structure in ray tracing algorithms. In this problem, you're to implement a static BVH to answer ray-triangle intersection queries efficiently (by static we mean the scene does not change). Note that if the ray hits more than one triangle, only the closest one is output.

For people who have no idea how the BVHs are constructed, here is a simple way: compute the *axis-aligned bounding box* (AABB) of all the objects, and then sort the objects by the coordinates of the "widest" dimension. After that, build the left sub-tree with the first half of objects and the right sub-tree with the remaining, recursively. Note that the AABBs of the objects in the left sub-tree and the right sub-tree might be overlapping, so when traversing the BVH, you need to visit both sub-trees if the ray intersects with the AABBs of both halves of objects. When the number objects is small, we do not split them, so the node becomes a leaf of BVH.

The detail above is just one way of implementing BVH. Another way is to split a node with an axis-aligned *plane*. Sorting is avoided (for each object, it takes constant time to determine which sub-tree it should go to), but you may need to place a single object in both sub-trees. Feel free to employ your own ideas in your program if you like. As long as your program is both correct and fast, it will be accepted.

Input

There will be at most 10 test cases. Each case begins with an integers p ($3 \leq p \leq 40,000$), the number of vertices, Each of the next p lines contains 3 integers, the coordinates of each vertex. The next line is an integer t ($1 \leq t \leq 80,000$), the number of triangles. Each of the next t lines contains 3 integers, the vertex indices of each triangle. Vertices and triangles are both numbered sequentially from 0.

The next line contains an integer q ($1 \leq q \leq 10,000$), the number of queries. Each of the next q lines contains 6 integers, the positions of origin and target of a ray. The origin is guaranteed to be strictly outside any triangle. Note that we're talking about rays, not segments, so "target" is actually just a point that the ray passed through. The coordinates are real numbers with small absolute values. The last test case is followed by a line with $p = 0$, which should not be processed.

The triangle meshes are extracted from real-world models.

Output

For each ray, print the ID of the triangle that the ray is hit, and the position of the hit point. If nothing is hit, print a single '`-1`'. We've carefully designed the judge I/O, but to further reduce the impact of floating-point errors, the *percentage* of incorrect output lines can be up to 5%, but please do not omit or add any line, since the comparison is done sequentially.

Sample Input

```
4
0 0 0
9 0 0
3 4 0
4 2 7
4
0 1 2
0 1 3
0 2 3
1 2 3
5
-3 0 0 4 2 3
3 -1 -2 4 2 3
6 -3 5 4 2 3
12 7 5 4 2 3
4 2 8 5 3 9
0
```

Sample Output

```
2 1.741935 1.354839 2.032258
0 3.400000 0.200000 0.000000
1 4.410256 0.974359 3.410256
3 4.568889 2.355556 3.142222
-1
```