```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;


class Main {
    public static void main(String[] args) throws IOException, NumberFormatException{

        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        String[] info = input.readLine().split(" ");
        int N = Integer.parseInt(info[0]);
        int H = Integer.parseInt(info[1]);
        int C = Integer.parseInt(info[2]);

        Climb climb = new Climb(N);

        for (int point = 0; point < N; point++) {
            String[] line_point = input.readLine().split(" ");
            int x = Integer.parseInt(line_point[0]);
            int y = Integer.parseInt(line_point[1]);
            Point p = new Point(x,y);
            climb.addList(p);
        }

        climb.order();
        int max_reach=Integer.MIN_VALUE;
        int[] r_list = new int[C];

        for (int R = 0; R < C; R++) {
            String line_case = input.readLine();
            int reach=Integer.parseInt(line_case);
            if (max_reach<reach)
                max_reach=reach;

            r_list[R]=reach;
        }

        climb.create_graph(max_reach,N);

        for (int i = 0; i < C; i++) {

            if (r_list[i]>=H)
                System.out.println(0);

            else{
                int result=climb.BFS(r_list[i],H,N);
                if (result==Climb.INFINITY)
                    System.out.println("unreachable");
                else
                    System.out.println(result);
            }
```

```
        }
    }
}

class Point implements Comparable<Point>{
    private int px;
    private int py;

    public Point(int x, int y){
        this.px=x;
        this.py=y;
    }

    public int value_x(){
        return this.px;
    }

    public int value_y(){
        return this.py;
    }

    @Override public int compareTo(Point otherPoint){
        if(this.py<otherPoint.py)
            return -1;
        if(this.py>otherPoint.py)
            return 1;
        return 0;
    }
}

class Climb {
    private Graph G;
    private List <Point> P;
    public static final int INFINITY = Integer.MAX_VALUE;
    public static final int WHITE = 0;
    public static final int GREY = 1;
    public static final int BLACK = 2;

    public Climb(int npoints){
        this.G=new Graph(npoints);
        this.P = new ArrayList<Point>(npoints);
    }

    public void add_to_graph(int p1, int p2, double dist){
        this.G.addEdge(p2, p1, dist);
    }

    public void addList(Point p){
        P.add(p);
    }

    public void order(){
        Collections.sort(P);
    }

    public void create_graph(int Reach, int N_points){

        for (int i = 0; i < N_points-1; i++) {
```

```
            int next=i+1;
            while (next<N_points && P.get(next).value_y() - P.get(i).value_y()<=Reach){

                double d=dist(P.get(i), P.get(next));
                if (d<=Reach)
                    add_to_graph(i,next,d);

                next++;
            }
        }
    }

    public double dist(Point p1, Point p2){
        int x1=p1.value_x();
        int x2=p2.value_x();
        int y1=p1.value_y();
        int y2=p2.value_y();

        double dist = Math.sqrt(Math.pow(x2-x1, 2)+Math.pow(y2-y1, 2));

        return dist;
    }

    public int BFS(int Reach, int Height, int N_points){

        int[] color = new int[N_points];
        int[] dist_array = new int[N_points];

        for (int i = 0; i < N_points; i++) {
            color[i]=WHITE;
            dist_array[i]=INFINITY;
        }

        Queue<Integer> S = new LinkedList<Integer>();

        for (int p = 0; p <N_points && P.get(p).value_y()<=Reach; p++) {
            color[p]=GREY;
            dist_array[p]=1;
            S.add(p);
        }

        while(!S.isEmpty()){

            int u = S.remove();
            for(Edge v: this.G.adjacents[u]){
                if (color[v.dest]==WHITE && Reach >= v.weight) {
                    color[v.dest]=GREY;
                    dist_array[v.dest]=dist_array[u]+1;
                    S.add(v.dest);
                }
            }
            color[u]=BLACK;
        }

        int mindist=INFINITY;

        for (int p = N_points-1; p >= 0 && P.get(p).value_y()+Reach>=Height; p--) {
```

```
            if(mindist>dist_array[p]){
                mindist=dist_array[p];

            }
        }
        return mindist;
    }
}


class Graph {
    int nodes;
    List<Edge>[] adjacents;
    @SuppressWarnings("unchecked")
    public Graph(int nodes)
    {
      this.nodes = nodes;
      adjacents = new List[nodes];
      for (int i = 0; i < nodes; ++i)
        adjacents[i] = new LinkedList<>();
    }
    /* Adds the edge (U,V) to the graph. */
    public void addEdge(int idp1, int idp2, double dist)
    {
      adjacents[idp1].add(new Edge(idp1, idp2, dist));
      adjacents[idp2].add(new Edge(idp2, idp1, dist));
    }
}

class Edge{
    int source;
    int dest;
    double weight;

    public Edge(int source, int dest, double weight){
      this.source=source;
      this.dest=dest;
      this.weight=weight;
    }
}
```