Problem 2

Given that the M obstacles are constant and only the N drones are moving, we can construct a kd-tree (2d-tree in this case) to calculate the nearest obstacle to each of the drones.

Tree Construction

Consider O = list of obstacles (length M) and D = list of drones (length N).

```
class KDNode(x,y, x_flag)
{
	x, y,
	x_flag,
	leftChild = NULL,
	rightChild = NULL,
	selection_flag = False
}

ConstructKDTree(O, x_flag = True):
	pivot = new KDNode(O[0].x, O[0].y, x_flag)
	if len(O) == 1:
		return pivot
	leftTree = [ ], rightTree = [ ]
	if x_flag == True:
		For each point p in O:
			if p.x < pivot.x:
				Add p to leftTree
			else:
				Add p to rightTree
	else:
		For each point p in O:
			if p.y < pivot.y:
				Add p to leftTree
			else:
				Add p to rightTree
	pivot.leftChild = ConstructKDTree(leftTree, !x_flag)
	pivot.rightChild = ConstructKDTree(rightTree, !x_flag)
```

Height of tree (h) = O(log M) (avg. case)
Thus, search time complexity = O(h) (avg. case)

```
FindNearestNeighbor(Drone d, KDTree root):
	min_dist = infty
	min_point = None
	if root.leftChild == NULL and root.rightChild == NULL:
		if not root.selection_flag and dist(root,d) < min_dist:
			min_point = root
			min_dist = dist(root, d)
		return (min_point, min_dist)
	if not root.selection_flag:
		min_point = root
		min_dist = dist(root, d)
	if root.axis_flag:
		if d.x < root.x:
			min_point1, min_dist1 = Find Nearest Neighbor(d, root.leftChild)
			if min_dist1 < min_dist:
				min_dist = min_dist1
				min_point = min_point
			if abs(d.x - root.x) < min_dist:
				min_point1, min_dist1 = Find Nearest Neighbor(d, root.rightChild)
				if min_dist1 < min_dist:
					min_dist = min_dist1
					min_point = min_point1
		else:
			min_point1, min_dist1 = Find Nearest Neighbor(d, root.rightChild)
			if min_dist1 < min_dist:
				min_dist = min_dist1
				min_point = min_point
			if abs(d.x - root.x) < min_dist:
				min_point1, min_dist1 = Find Nearest Neighbor(d, root.leftChild)
				if min_dist1 < min_dist:
					min_dist = min_dist1
					min_point = min_point1
	else:
		if d.y < root.y:
			min_point1, min_dist1 = Find Nearest Neighbor(d, root.leftChild)
			if min_dist1 < min_dist:
				min_dist = min_dist1
				min_point = min_point
			if abs(d.y - root.y) < min_dist:
				min_point1, min_dist1 = Find Nearest Neighbor(d, root.rightChild)
				if min_dist1 < min_dist:
					min_dist = min_dist1
					min_point = min_point1
```

```
        else:
                min_point1, min_dist1 = Find Nearest Neighbor(d, root.rightChild)
                if min_dist1 < min_dist:
                        min_dist = min_dist1
                        min_point = min_point
                if abs(d.y - root.y) < min_dist:
                        min_point1, min_dist1 = Find Nearest Neighbor(d, root.leftChild)
                        if min_dist1 < min_dist:
                                min_dist = min_dist1
                                min_point = min_point1
    return min_dist, min_point
```

Problem 3

In order to find the K-Nearest neighbors, we run the above FindNearestNeighbor algorithm K times. Note that we set the min_point.selection_flag to True once it is selected. This prevents the algorithm from selecting the same node again and again. This FindKNearestNeighbors algorithm can be run for all the drones.

```
FindKNearestNeighbors(Drone d, KDTree root, int k):
    knn = [ ]
    for i in range(k):
        min_dist, min_point = FindNearestNeighbors(d, root)
        knn.append((min_dist, min_point))
        min_point.selection_flag = True
    return knn
```

Time complexity = O(k.h), (avg case) where k = number of neighbors required and h = height of the tree

Problem 4

Similar to OctTree, but with a slight modification. Given a space S we first create a random cuboid C inside the space given to us, and then, we create 8 such non-intersecting subspaces whose union is S - C and the intersection of any two subspaces is NULL.

Let a cuboid be defined by its beginning and ending coordinates

```
class Cuboid
{
```

```
        x1, y1, z1, x2, y2, z2;
};
```

Let us define an Octree as

```
class Octree
{
        Cuboid c,
        Octree child[8]
};


GenerateNCuboids(int n):
        list = [ ]
        S = Cuboid(0,0,0,10,10,10)
        subsps = { S }
        for i in range(n):
                S = extract random subsps from S and remove it
                Let X1, X2, Y1, Y2, Z1, Z2 be the coordinates of S
                C = random cuboid inside S
                Let x1, x2, y1, y2, z1, z2 be the coordinates of C
                Add C to list
                subspace s1 = Cuboid(x2, X2, y2, Y2, z2, Z2)
                subspace s2 = Cuboid(X1, x1, Y1, y1, z1, Z1)
                subspace s3 = Cuboid(x2, X2, Y1, y2, z2, Z2)
                subspace s4 = Cuboid(X1,x2, y1, Y2, z2, Z2)
                subspace s5 = Cuboid(X1, x2, Y1, y1, z1, Z2)
                subspace s6 = Cuboid(x1, X2, Y1, y2, Z1, z1)
                subspace s7 = Cuboid(x1, X2, y2, Y2, Z1, z2)
                subspace s8 = Cuboid(X1, x1, y1, Y2, Z1, z2)
                Add the new subspaces to subsps
```