1. The algorithm works as follows. It uses a scanline starting at $y = \infty$. It also has two balanced binary trees that keep track of the x-coordinates of the left and right sides respectively of the squares intersecting the scanline. Every node has a size value which keeps track of the amount of nodes that are part of it's subtree. Whenever the scanline comes across the top of a rectangle we simply add the left and right x-coordinate to their respective trees. Whenever the scanline reaches a point we calculate the depth as the size of the left or right tree (they have the same size) minus the amount that is left of x on the right tree and the amount that is right of x on the left tree. If our scanline reaches the end of a square we simply remove the left and right x-coordinate from the two respective trees.

   For every $R \in \mathcal{R}$ we will denote the left and right x-coordinate wit R_{x_1} and R_{x_2} respectively and R_{y_1} and R_{y_2} denote the bottem and top y-coordinate respectively.

   Lemma 1. *The algorithm computes $d_{\mathcal{R}}(p)$ for all $p \in P$ correctly.*

   Proof. Remember that the priority of an event is given by its y-coordinate, and that because no points or vertices are co-linear it can never happen that that vertex of a rectangle shares a y-coordinate with a point. We shall prove the lemma by induction over the events in the queue.

   Let q be the newest event in the queue. Assume as a induction hypothesis that all points with a higher y-coordinate have been computed correctly and that the left and right tree consist exactly of the squares intersecting the previous sweepline. The induction basis is true, because when $y = \infty$ no squares intersect the sweep line and no points have been passed.

   Assume that the next queue element is a point $p \in P$. Because the queue is sorted there has not been a bottem or top of a square since last iteration, hence the left tree and right tree represent the set of left x-coordinates and right x-coordinate intersecting sweep line going through p. We therefore know that the squares containing p form a subset of what is in the trees. We know that p is in rectangle R if and only if $R_{x_1} \leq p_x \leq R_{x_2}$ or equivalently not $R_{x_1} > p_x$ or $p_x > R_{x_2}$. Because $R_{x_1} < R_{x_2}$ we see that both sets of squares are disjoint. Therefore the amount of squares that contain p are the amount of squares currently in the tree minus the amount squares where $R_{x_1} > p_x$ or $p_x > R_{x_2}$ hold. These are exactly the squares that are right to p_x in left tree or are left to p_x in the right tree, hence we have correctly calculated $d_{\mathcal{R}}(p)$ and the induction hypothesis remains true.

   Assume that the queue event is the top of a rectangle. Because no vertices of two different square can be the same, we see that when this square is added to the trees that the trees contain exactly the squares that are on the current sweep line

   If the queue event is the bottem of a rectangle then this rectangle is no longer intersected by the sweep line, hence by removing it from the trees ensures that the trees contain exactly the the squares that are on the current sweep line.

   We conclude that when the sweep line has passed all of the events, that the property in the induction hypothesis still holds, hence we have proven that the algorithm computes $d_{\mathcal{R}}(p)$ for all $p \in P$ correctly. □

   Lemma 2. *The algorithm takes $o((n+m)log(n+m))$ time.*

   Proof. In the beginning all points, tops of squares and bottem of squares are sorted. This takes $o((n+m)log(n+m))$-time. After that it has to pass all $o(n+m)$ events. For each event it has to either add an element to the binary trees, do a binary search or remove an element from the binary trees. All These operations take $o(\log m)$-time, hence handling the queue takes $o(n+m)o(\log m)$-time. Thus our algorithm takes $o((n+m)log(n+m))$-time. □

   Because of lemma 1 and 2 we can conclude the following:

   Theorem 3. *Calculating the depth of n points given m rectangles takes $o((n+m)\log(n+m))$-time.*

2. Because all triangles have one edge that is aligned to one of the axes, we have four types of triangles to work with. In the answer I will only look at the cases that an edge is aligned with x-axis and that this edge is on the top of the triangle. This is possible because we can handle the types of triangles seperately. For every type we rotate and/or flip the the plane such that we get the types of triangles as stated earlier. This process takes linear time. The process for the specified type of triangles will be very similar with exercise 1. Just as in exercise we we will have an event que consisting of the points, top triangles and bottem of

triangles. Whenever the scanline reaches the top of a triangle it's leftmost point and rightmost point are added to left and right tree respectively. The difference is how different points are compared. We cannot simply compare the x-coordinates, because the edges do not run vertically anymore. Now we also have to take the y-coordinate in consideration. In the left tree we will say that point (x_1, y_1) is left of (x_2, y_2) if and only if $x_1 \cos 30° + y_2 \sin 30° < x_2 \cos 30° + y_2 \sin 30°$. In right tree we will say that (x_1, y_1) is left of (x_2, y_2) if and only if $x_1 \cos 30° - y_2 \sin 30° < x_2 \cos 30° - y_2 \sin 30°$

3. Our method for squares and rectangles will not work for arbitrary triangles. This is because whenever we add a triangle to our binary trees, the order of edges could change in any way in the future. Hence we have to look at possible intersections, hence the time-complexity when using our method cannot simply depend on the amount of triangles and points.