Solution 13.1  Intervall Trees.

a) The set of the used interval boundaries is \{1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13\}. A skeleton of an interval tree is a (nearly) complete binary search tree that maintains the interval boundaries as keys. Thus we obtain the following skeleton:

![Interval Tree Skeleton](image)

After inserting the intervals into this tree we obtain the following interval tree:

![Interval Tree](image)

**Notice:** The skeleton does not necessarily need to be a complete binary search tree. It is sufficient to use some balanced search tree.

b) The query for the value 5 considers the nodes 9, 4, 7 and 6. It considers the following lists. Since 9 > 5 the \(u\) list of node 9 is traversed from the left, and the interval \([1, 9]\) is outputted. The interval \([6, 13]\) is only tested; since it does not contain 5 it is not outputted. The intervals of the \(u\) list are sorted ascending by left boundary, and since 5 < 6 all the remaining intervals in this list doesn’t have to be outputted (and thus doesn’t have to be tested). After that we consider node 4. Since 4 < 5 the \(o\) list of node 4 is traversed from the left, and the intervals \([4, 7]\) and \([1, 6]\) are outputted. The interval \([2, 4]\) is only tested; since it does not contain 5 it is not outputted. Finally the \(u\) lists of the nodes 7 and 6 are tested. Since these are empty, no more intervals are outputted.

c) Let \(N\) be number of maintained intervals. An insertion operation needs time \(O(\log N)\) both in segment trees as well as in intervall trees. Also the query of a point has the same running time \(O(\log N + k)\) in both trees, where \(k\) ist the number of returned intervals.
Since interval trees store each interval in exactly one node (instead of in up to \(O(\log N)\) as segment trees do), they only need space \(O(N)\) (instead of \(O(N \log N)\) as segment trees).

**Solution 13.2  Non-Dominated Points.**

**Stopping points:** We run the scanline from left to right, and we stop if and only if a new point is found. If there is more than one point lying on the scanline, we consider them in decreasing order of their \(y\) coordinates.

**Scanline data structure:** We use only one variable to store the \(y\) coordinate that is minimum among the \(y\) coordinates of the points considered so far. Initially the variable is set to \(\infty\).

**Update:** As soon as new point is found whose \(y\) coordinate is less or equal the current stored minimum, it is outputted, and the stored minimum is updated correspondingly.

**Extracting the solution:** Obviously all non-dominated points are outputted directly once they are found.

**Running time:** Sorting \(n\) points can be achieved in time \(\Theta(n \log n)\). The cost for updating is constant at each stopping point. Since there exist \(n\) stopping points we obtain an overall running time of \(\Theta(n \log n)\).

**Solution 13.3  Piercing Orthogonal Rectangles.**

**Stopping points:** We run the scanline from left to right, and we stop every time that a rectangle starts or ends. For this purpose, we sort the \(x\) coordinates of the left and right sides of the rectangles in ascending order and use them as stopping points.

**Scanline data structure:** As currently active objects, we store the intervals resulting from the intersection of the rectangles with the scanline. For this purpose we use a modified segment tree. However in each node we don’t store the intervals itself but only a counter that indicates the maximum number of overlapping intervals in the corresponding subtree. This number can easily be computed recursively: if the two children of a node \(v\) contain a maximum of \(m_1\) and \(m_2\) overlapping intervals, then the maximum number of overlapping intervals below \(v\) is exactly the sum of the number of intervals represented at \(v\) and \(\max\{m_1, m_2\}\) (all the intervals represented by \(v\) overlap with the intervals represented by its children, so the maximum number

![Example](image)

**Example:** If the scanline intersects the rectangles on the left, then we obtain the intervals \([2, 6], [1, 5], [3, 4]\) (shown in the middle, f.l.t.r.). On the right a part of the scanline data structure is shown. The maximum number of overlapping intervals is indicated on the right of each node. Notice that the interval \([2, 6]\) is stored in the nodes \([2, 3], [3, 5]\) and \([5, 6]\). Thus \([2, 3]\) stores the counter 1, and \([3, 5]\) stores 2. An elementary interval that is overlappend by a maximum number of intervals stored below \([1, 5]\) is \([3, 4]\).
of overlapping intervals must be increased accordingly). Additionally, at each node \( v \) we store an elementary interval that is covered by a maximum number of intervals in the segment tree rooted at \( v \).

Notice that the \( y \) coordinates of the rectangles may be rational, but usually a segment tree can only maintain intervals with integer boundaries. Thus we sort the \( y \) coordinates of the rectangles in advance, remove duplicates and obtain the possible coordinates \( y_1, \ldots, y_k \) where \( k \leq 2n \). Now we change the \( y \) coordinates of each rectangle: if a rectangle \( R \) has the vertices \((x, y_i), (x, y_j), (x', y_i) \) and \((x', y_j)\) then we replace it with a rectangle \( R' \) with the vertices \((x, i), (x, j), (x', i) \) and \((x', j)\). Using this method we can use a segment tree with interval boundaries in \( \{1, \ldots, k\} \).

**Update:** We distinguish two cases.

**Case 1: A new rectangle starts.** Let \( I \) be the interval emerging from the intersection of the scanline and the new rectangle, and let \( v_1, \ldots, v_k \) be the nodes where a usual segment tree would store \( I \). In all of these nodes the counter of maximum overlapping intervals is increased by 1. Additionally we have to update the counters of all preceding nodes on a path to the root. The values of the other nodes remain unchanged.

**Case 2: A rectangle ends.** In this case we proceed analogously to case 1, decrement the counter in the corresponding nodes of the interval by 1, and update the counters of the preceding nodes.

**Extracting the solution:** Every time the scanline data structure is updated, we check whether the overall number of maximum overlapping rectangles increases. This number is stored in the root of the segment tree. We store the maximum number of overlapping rectangles, an elementary interval that is overlapped most, and the corresponding scanline position. At the end we can use this information to compute a point that lies within the stored elementary interval at the stored scanline position.

**Running time:** The initial sorting of the stopping points can be done in time \( \Theta(n \log n) \). The same holds for the sorting of the \( y \) coordinates. Insertion and deletion in the modified segment tree can be performed in time \( \Theta(\log n) \), since we only have to update nodes along the paths visited in these operations. Since there are \( 2n \) stopping points, the resulting running time is \( \Theta(n \log n) \). The space consumption is bounded by \( \Theta(n) \) since not the intervals itself but only a counter is stored (which uses only constant space).

**Remark:** In the exercise it was explicitly stated that also points on the boundary of a rectangle might pierce a rectangle. However the above solution only finds points that are lying completely within a rectangle. We can solve this problem as follows. Let \( v \) be an arbitrary node of the modified segment tree representing an interval \([y, y']\). In \( v \), we do not only store the number \( m(v) \) of maximum overlapping intervals but also two numbers \( l(v) \) and \( r(v) \). The number \( l(v) \) indicates how many intervals in the subtree of \( v \) are pierced by the value \( y \), \( r(v) \) analogously indicates the number of intervals pierced by \( y' \). Let \( w_1 \) be the left and \( w_2 \) be the right succeeding node of \( v \), and let \( k \) be the number of intervals represented by \( v \). It is easy to see that we have

\[
m(v) = k + \max\{m(w_1), m(w_2), r(w_1) + l(w_2)\}.
\]  

(1)

Analogously the numbers \( l(v) \) and \( r(v) \) can be computed recursively. If there exists a stopping point where some rectangles start and some others end, then we first have process all starting rectangles, and only after that we can process the ending ones. Thus the algorithmic approach remains the same, only some technical details have to be changed.
Solution 13.4  Overlapping Triangles.

a) Here we can simply use the standard method for intersections of line segments. The input for the algorithm consists of the edges of each triangle as line segments. We just have to ensure that intersections of segments corresponding to the same triangle are ignored. The running time is \( \Theta(n \log n) \) where \( n \) is the number of triangles.

b) **Stopping points**: We use a vertical scanline that runs from left to right. For each triangle, we use the vertices with the smallest and the largest \( x \) coordinate as stopping points.

**Scanline data structure**: Since no two triangles intersect, the triangles can be compared according to \( y \) coordinates. Even though the triangles have \( y \) coordinates that vary in horizontal direction, the *relative* order of the triangles remains the same. Therefore, we can use an AVL tree that is ordered in \( y \) direction as scanline data structure.

**Update**: When we reach a stopping point that corresponds to the beginning of a triangle \( T \), we search for the position where to insert \( T \) in the AVL tree. To do this, we need to compare \( T \) at each node with the triangle \( T' \) stored in that node. We compare them by testing whether the vertex of \( T \) lying on the scanline lies also in \( T' \) or not. If this is the case, then \( T \) is completely contained in \( T' \) (since the triangles do not cross), and we stop. Otherwise, we determine if \( T \) is above or below \( T' \) and we proceed accordingly in the AVL tree. When we reach a leaf, we add \( T \) to the AVL tree in the usual fashion.

When we reach a stopping point corresponding to the end of a triangle we simply delete the corresponding triangle from the tree. We search for it in the same way as in the case of insertion.

**Extracting the solution**: If a triangle is contained in some other triangle, then the process terminates before the end. Otherwise, it terminates after all the stopping points were processed.

**Running time**: Let \( n \) be the number of triangles. Sorting the vertices is possible in time \( \Theta(n \log n) \). Insertion and deletion in an AVL tree each require time \( \Theta(\log n) \) for every of the \( \Theta(n) \) stopping points, because we can easily determine (by cross product or by a comparison of linear equations) in constant time whether a point is above, below or on a straight line – and thus also whether a point is above, below or inside a triangle. Thus the overall running time is \( \Theta(n \log n) \).