1 Orthogonal Segment Intersection

The problem is the following: Given a set $S$ of $n$ horizontal segments on the plane and a vertical query segment $s$, report the segments of $S$ intersected by $s$. We will see two different but related data structures that solve the problem.

1.1 The $SR$-tree

1.1.1 The Algorithm

The segments of $S$ are stored in a segment tree $T$ based on the $x$-coordinates of the endpoints of the segments in $S$. This supports point inclusion queries along the $x$-axis. Each node of $T$ has a range tree (the secondary structure) that supports 1-D range searching along the $y$-coordinate among the segments stored at that node.

The query is processed as follows. Let $s$ be the interval $[(x, y_1), (x, y_2)]$ with $y_1 < y_2$. $x$ is located in $T$. The nodes encountered while locating $s$ in $T$ all correspond to intervals of $S$ which span $x$. At each of these nodes a range search (in the $y$-interval $[y_1, y_2]$) is done to locate all segments in $S$ that intersect $s$.

1.1.2 Proof of Correctness

The correctness of the algorithm follows from the fact that the segment tree $T$ supports point enclosure queries along the $x$-axis. All segments stored in the nodes in the search path for $x$ contain $x$. Clearly, these are also the only segments that contain $x$. Now searching among these segments to find those whose $y$-coordinates are contained in the $y$-range of $s$ gives exactly those intervals in $S$ which are intersected by $s$. This range search is done at each node using the range tree stored there.

1.1.3 Complexity Analysis

The search path for $x$ in $T$ contains $O(\log n)$ nodes. In each node a range search is done. This range search takes $O(\log n + m)$ time per node (where $m$ is the number of segments reported at that node) since at most $n$ segments are stored at each node. Hence the time taken to answer a query is $O(\log^2 n + k)$ where $k$ is the total number of segments in $S$ intersecting $s$.

To analyze the space needed, note that a segment tree alone would use $O(n \log n)$ space. If $q_\mu$ is the space required at each node of a normal segment tree, we have $\sum_{\mu \in T} q_\mu = n \log n$. In the $SR$-tree, each node, $\mu$ of $T$ uses $q_\mu \log q_\mu$ space since a range tree over $n$ intervals uses $O(n \log n)$ space. Therefore, the space used by the $SR$-tree is $\sum_{\mu \in T} q_\mu \log q_\mu \leq \log n \sum_{\mu \in T} q_\mu$.
(since \( q_n \leq n \)) = O(n \log^2 n). The space can be reduced to \( O(n \log n) \) by noting that at each node of \( T \) we do a one-dimensional range query for which a balanced search tree (which uses \( O(n) \) storage with an input of size \( n \)) suffices.

1.2 The \( RS \) tree

1.2.1 The Algorithm

The primary data structure is a range tree \( T \) which supports range search in the \( y \)-direction. We can use this to find all segments whose \( y \)-coordinates fall inside the \( y \)-range of \( s \). At each allocation node (there are \( O(\log n) \) of them), we now need to do a point-enclosure query to find out which of the horizontal segments stored at that node contain the \( x \)-coordinate of \( s \). This is done through the secondary data structure which is a segment tree.

1.2.2 Proof of Correctness

All the segments in \( S \) whose \( y \)-coordinates are contained in the \( y \)-range of \( s \) are clearly reported by the range search performed in \( T \). Of these, we should report only those whose \( x \)-intervals contain the \( x \)-coordinate of \( s \) (since these are exactly those segments of \( S \) that intersect \( s \)). These are found by doing the point location query at each allocation node found during the range search in \( T \).

1.2.3 Complexity Analysis

We need to do a point location query at each of \( O(\log n) \) allocation nodes. A analysis similar to that employed for the \( SR \) tree shows that a query can be processed in \( O(\log^2 n + k) \) where \( k \) is the total number of segments in \( S \) intersecting \( s \). The \( O(n \log^2 n) \) space can be analyzed in an analogous manner since both the range tree and the segment tree use \( O(n \log n) \) storage on input of size \( n \). It is interesting that while we could reduce the storage requirement of the \( SR \) tree from \( O(n \log^2 n) \) to \( O(n \log n) \) by replacing the range tree by the balanced search tree as the secondary data structure, we cannot obtain a similar reduction in space here. Inputs that need \( O(n \log^2 n) \) space in the worst case can be constructed.

2 Segment Intersection

In this section we modify the problem slightly to allow the segments in \( S \) to be arbitrarily sloped. An important restriction is that these should not intersect each other. The query segment is still vertical. We will see that essentially the same data structure as before solves this problem

2.1 The Algorithm

We maintain a segment tree \( T \) on the \( x \)-coordinates of the endpoints of the segments in \( S \). \( T \) is used to locate the \( x \)-coordinate of the query interval. We know that the segments stored in the nodes in the path followed by the query in \( T \) are the only ones which can intersect \( s \).
Now we have to do a range search in each of the nodes in the query path. Each such node corresponds to a strip of the plane. Each such strip is decomposed into a set of trapezoids by the segments stored at the corresponding node of \( T \). Note that all these segments cross the strip entirely. Further they do not intersect each other. Hence as we move from left to right in the strip, the relative top-to-bottom order of the segments in that strip does not change. We can now store these segments in a balanced search tree (the secondary data structure) to do the range search.

Let us assume that in the balanced search tree the left child of a segment is a segment below it in the strip and the right child is a segment above it. We search for \( y_1 \) and \( y_2 \) in the balanced search tree. Reporting the segments at all the nodes between the two nodes the two searches end in gives all the segments of \( S \) that \( s \) intersects. Suppose that during the search for \( y_1 \), we are at a particular node of the balanced search tree. If \( y_1 \) is below the segment stored at that node, then we know that the search should proceed in the left subtree of that node else in the right subtree. This test directs our search. Whether or not a point is above or below a segment can be tested easily by projecting the point on the segment in the \( y \)-direction and comparing the \( y \)-coordinates of the point and the point of projection.

## 2.2 Proof of Correctness

By the properties of the segment tree, we know that the segments stored in the nodes encountered during the search for \( x \) in \( T \) are the only ones that can intersect \( s \). It is also clear that only segments whose \( y \)-range intersects the \( y \)-range of \( s \) can intersect \( s \). Since the segments in \( S \) do not intersect each other we use the natural top-to-bottom ordering induced on the segments that intersect any vertical strip of the plane to do a range search to determine the answer.

## 2.3 Complexity Analysis

From the analyses done in previous sections, it will be clear that the time taken to answer a query is \( O(\log n + k) \). The space requirement is \( O(n \log n) \).

The problem becomes much tougher when the segments in \( S \) are allowed to intersect themselves. The best known results are a query time of \( O(\sqrt{n} + k) \) and a space requirement of \( O(n \log^3 n) \).

## 3 Ray Shooting

Given a set \( S \) of horizontal segments on the plane and a query point \( p \) with coordinates \((x, y)\) find the first segment of \( S \) intersected by a vertical ray shooting up from \( p \).

We can solve this problem by using a \( SR \)-tree to find all segments in \( S \) that intersect a semi-infinite vertical line extending up from \( p \) and then determining which of these is first intersected by the ray shooting up from \( p \). However, this in the worst case can take \( O(n) \) time. A better solution can be obtained by organizing the \( y \)-coordinates of the segments stored at each node of the segment tree \( T \) (the primary data structure) in a binary search tree. Then at each node in search path for \( x \) in \( T \), we do a binary search for \( y \) in the binary search tree and locate the lowest segment of \( S \) stored at that node which is above \( p \). Thus
each node in the search path in $T$ yields a possible answer. The lowest among these is the required solution.

The correctness is clear from the above discussion. The space required is $O(n \log n)$. The time required is $O(\log^2 n)$ since a binary search is done at each of $O(\log n)$ nodes.

4 Orthogonal Rectangle Intersections

Given a set $R$ of orthogonal rectangles (rectangles with sides parallel to the axes), report all that intersect a query orthogonal rectangle.

Let $r$ and $s$ be two rectangles. $r$ and $s$ intersect iff one of the four following mutually exclusive cases holds:

1. the bottom-left corner of $s$ is in $r$;
2. the bottom-left corner of $r$ is in $s$;
3. the bottom side of $s$ is intersected by the left side of $r$;
4. the bottom side of $r$ is intersected by the left side of $s$.

If $s$ is the query rectangle and $r$ is a rectangle in $R$, then the four cases can be solved (as we have seen before) by using the $SS$-tree for 2-dimensional point enclosure, the $RR$-tree for 2-dimensional range search, the $SR$-tree for orthogonal segment intersection (where the query segment is horizontal and the other segments vertical) and the $RS$ tree for orthogonal segment intersection respectively. We thus use a four component data structure whose elements are obtained by the symbolic expansion of $(R + S)^2$. The query processing time is $O(\log n + k)$ where $k$ is the number of intersections reported and the space requirement is $O(n \log n)$.

This idea can be generalized to $d$-dimensions where the number of hyper-rectangles from a given set that intersect a query hyper-rectangle can be found by using the data structures determined by the symbolic expansion of $(R + S)^d$. 