5.10

(a) We can do some extra work when building up the range tree. That is, for each node in the binary search tree, record the number of descendents including the node itself. This operation happens once for each node, hence will be bounded by \(O(n)\) time. Thus the construction time is still bounded in \(O(n \log n)\) time.

When doing query, we still apply Algorithm “FindSplitNode()” (Textbook, Page 97) to find the split node, and then apply Algorithm “1DRangeQuery()” (Textbook, Page 97) to find all the sub-trees we have to concern. The only difference is, we don’t call the subroutine “ReportSubtree()” (Textbook, Page 98) to report the leave nodes one by one. Instead, we just fetch out the number of descendents that we recorded before, and sum them up, thus we get the number of leaves in a query range.

To locate \(x\) and \(x'\), we need \(2 \log n\) time. To perform the summation over all concerned sub-trees, we need \(2 \log n\) time (Assuming each fundamental summation can be done is \(O(1)\) time). Therefore, the ranging counting query can be done in \(2 \log n + 2 \log n = O(\log n)\) time.

(b) The extra operation is similar to (a). We building the n-dimensional range tree recursively, just as described in Textbook, Page 109. However, for each node when building the last-level associated range tree (1-dim range tree based upon the last coordinate we consider), we record the number of descendents including the node itself.

Querying \(x\) and \(x'\) in this \(d\)-dim range tree needs \(O\left(\log^d n\right)\) time. This also means that at the last-level associated tree, we have \(O\left(\log^d n\right)\) sub-nodes to consider all together. To sum up all the descendents in these sub-nodes, we need \(O\left(\log^d n\right)\) time. Therefore, the total range query time is bounded by:

\[
O\left(\log^d n\right) + O\left(\log^d n\right) = O\left(\log^d n\right)\text{ time.}
\]

(c) As Theorem 5.11 proves, the range query for a range tree with fractional-cascading can be done in \(O\left(\log^{d-1} n + k\right)\) time. So we only have to concern how the \(O(k)\) traversal time can be saved when performing range counting.

This is simple. First, when constructing the range list, we also **record for each entry its index number in the list**. Then, taking 2-dim case for example: when doing query for \([y, y']\), we not only keep track of the entry in the associated arrays whose \(y\)-coordinate is the smallest one larger then or equal to \(y\), but also keep track of such entry for \(y'\). Then, we subtract their indexes so that get the
number of elements in this range in $O(1)$ time. Therefore, we can add up the subtraction results one by one for each of the sub-lists, which take $O(\log n)$ time. So the total query time for 2-dim case is bounded by $O(\log n)$.

This approach holds to d-dim case, in which we have to sum up the subtraction for $O(\log^{d-1} n)$ range lists, so the total range counting query can be done in $O(\log^{d-1} n)$ time.

5.11

An example for the plane-sweep situation is shown above.

**Event points:**
the 2n endpoints for the n horizontal line segments, and the m vertical line segments. We sort the 2n+m events from left to right in (2n+m)log(2n+m) time, or more essentially, in (n+m)log(n+m) time.

**1-Dim range search tree (BST) and update event:**
The BST is initially empty; Each time the sweep-line meet a start or endpoint of the horizontal line, the BST is updated which takes $O(\log n)$ time. So the update event takes 2nlogn = $O(n\log n)$ time all together.

**Query event:**
Each time the sweep line overlap with a vertical line l, we compute how many vertical lines in the BST intersect it. Suppose the up endpoint of l is $(x_l^{up}, y_l^{up})$, and the down endpoint of l is $(x_l^{down}, y_l^{down})$. This is actually a 1-Dim range counting query problem as shown in 5.10, because we can report how many horizontal lines lie in the range $[y_l^{up}, y_l^{down}]$, which is equivalent to how many intersections happen. From 5.10, we know the query takes $O(\log n)$ time. So the query events takes
O(m log n) time in total. We add up all the counting together and get the total number of intersections. In general, this algorithm has time complexity bounded by:

\[ O((n + m) \log (n + m)) + O(n \log n) + O(m \log n) = O((n + m) \log (n + m)) . \]

6.4

We can shot a ray from q in whatever directions. For example, let shot a vertical ray vertically upward. In addition we suppose the planar subdivision has a rectangular bound as shown below:

Since the subdivision has at most n edges, we can compute the intersection between each edge and the ray, and decide the first intersected edge in O(n) time. Then we fetch the two half-edges associated with this edge, and decide which one of them has q on the left side. The face associated with this half edge must contain q, and we are done. The checking on to half edges takes O(1) time. So the total time is bounded by O(n) time. Pay attention that even q lies outside the subdivision, since we have a “virtual” rectangular bound, the ray will still hit an edge of it, we can still make decision based upon the half-edge associated with the edge. So we are guaranteed to find the face that contains q.

6.7

Given the star-shaped polygon and a satisfied point p, we can shot a ray from p to all of the vertices of
the polygon. Hence we get n directed lines, and we sort them with respect to the direction angle, which can be done in $O(n \log n)$ time. We also compute the direction angle from p to q, and do a binary search to find the area $pq$ lies in, which can be done in $O(1+\log n)$ time.

Suppose the sector where $pq$ lies in is defined by vertex p, $v_i$ and $v_j$, then we only have to decide if q lies inside the triangle $pv_iv_j$, hence can decide if q lies inside the polygon. Apparently, This can be done in $O(1)$ time.

In general, the query time given point p can be done in $O(1+\log n) + O(1) = O(\log n)$ time. However, the pre-processing time (sorting) needs $O(n \log n)$ time.

In we are not given point q, we can use the method introduced in lecture, or that shown in Section 6.2 of the textbook to construct the trapezoidal map of the polygon, which according to Lemma 6.2, has $O(n)$ trapezoids. The problem then become a point location problem and can be done in $O(\log n)$ time, according to Corollary 6.4.

Alternatively, we can perform linear programming on the polygon’s edges, find the feasible region and thus find a satisfied point p in $O(n)$ time (just as what we have done for Exercise 4.15). Then we can do the same work above to decide whether or not q lies in the sector triangle. The query time is still bounded by $O(\log n)$.

Pay attention that, no matter which approach we choose, the pre-processing time (trapezoiding or sorting) requires more than $O(\log n)$ time. However, we’re only concerned about the query time here.
6.16

Similar to the method described in book, we can set a rectangular bounding box to these n segments, and perform a trapezoidal decomposition to this bounding box. An example is shown below:

![Trapezoidal decomposition example](image)

The trapezoidal map is the data structure we use for the vertical ray shooting problem. Complexity analysis is as follows:

**Storage:** we can see each endpoint will shoot out 2 vertical lines, one upward and the other downward. And each vertical line is shared by at most 2 trapezoids. Therefore, the total number of trapezoids is bounded by $2 \times 2 \times 2n = 8n = O(n)$. As the trapezoid map is planar subdivision, the storage we need for it is $O(n)$.

**Preprocessing time:** we can either use the algorithm described in class or that in Section 6.2 to derive the trapezoid map data structure. Both of them has time complexity bounded by $O(n \log n)$.

**Query time:** Given a ray starting at point $p$, we can perform a point location query on the map which takes $O(\log n)$ time, according to Corollary 6.4, so we decide the trapezoid that contains $p$. Then we just shot a vertical ray upward, and detect the edge the ray first intersects to the trapezoid. This edge is the first segment in $S$ intersected by the ray. Therefore, the query time is in $O(\log n)$.

If the segments intersect, we can still apply the trapezoidal decomposition. Suppose we have $m$ intersections. We just “break” the intersected pairs of lines apart, and shot vertical lines for the intersection point as well (shown below). In this way, we have $n + 2m$ equivalent points.

![Intersection points](image)

Accordingly, the storage is bounded by $O(n + m)$;
The preprocessing time is bounded by $O((n + m) \log (n + m))$;

The query time is bounded by $O(\log (n + m))$