10.1

The scheme of this problem is depicted as follows:
Build a 1-dim range tree on y-coordinate; O(n) storage, O(nlogn) depth;
For each node of the range tree, associate it with an interval tree of the canonical subset of segments.

(a) The preprocessing includes building a 2-dim range-tree, for each node of which building a canonical interval tree. The 2 algorithms from book can be used:
   (1) – Build2DRangeTree(P): Page 107
   (2) – ConstructIntervalTree(I): Page 223
The only modification we need to make is: to replace Line 1 of Alg. (1) with calling Alg. (2).
The query can be implemented with another 2 algorithm from book:
   (3) – 2DRangeQuery(): Page 108
   (4) – QueryIntervalTree(): Page 224
The only modification we need to make is: to replace Line 8 of Alg. (3) with calling Alg. (4).
(b) The query is actually a 2-level process: we first query on the main range tree, thus we are guaranteed to find all the nodes that store intervals which lies vertically in \([y: y']\); we then query the associated interval trees of these nodes, thus we further find all the intervals that contains the query point \(q_x\), from all those lies vertically in \([y:y']\). Therefore, this data structure can support a complete query.

(c) **Preprocessing time:** sort the segments w.r.t. to y-coordinate requires \(O(n\log n)\) time; For building a single canonical interval tree, we need \(O(n\log n)\) time. Thus to build canonical trees for all the \(n\) nodes of the range tree, we have \(T(n)=2T(n/2)+O(n\log n)\), which is actually bounded by \(O\left(n \log^2 n \right)\).

**Storage:** a 1-dim range tree requires \(O(n)\) space; a simple interval tree requires \(O(n)\) space. Similar to the argument of Lemma 5.6, we can find the “range tree with canonical interval tree” structure used here requires \(O(n\log n)\) space.

**Query time:** Searching the range tree for the \(O(\log n)\) canonical subsets in range \([y_{low}, y_{up}]\) requires \(O(\log n)\) time; At each of the subset, searching the interval tree for segments containing \(q_x\) requires \(O(\log n+k_s)\). So the total query time is \(O\left(\log^2 n + \sum k_s\right) = O\left(\log^2 n + k\right)\).

10.2

Please refer to Professor Goodrich’s book [1] for detailed analysis at Section 8.3.6, Page 352.

10.6

(a) We adopt the segment tree structure introduced in Section 10.3. However, instead of storing intervals in nodes, we just store the number of intervals. For example, in Fig. 10.9 of the book, for the node pointing to “S2, S5”, we just store “2”, as we have two intervals here. The other structure remains the same. As the number of nodes is bounded by \(O(n)\), and the number stored at each node is in \(O(1)\) space, thus the total space reduced from \(O(n\log n)\) to \(O(n)\).

The pre-processing time doesn’t change, thus are bounded by \(O(n\log n)\), according to Theorem 10.12. When doing query, we just add up the numbers in the query path, which has \(O(\log n)\) depth, as shown in “QuerySegmentTree()” (Page 234). So the query time is bounded by \(O(\log n)\).

(b) We can solve this problem under a Fractional Cascading framework:
While keeping the sorted edge lists at each node, we also build pointers from list of the parent node to that of the children nodes. As the intervals are stored in the interval tree with no repeat, the newly added pointers are bounded by \(O(n)\) storage. Meanwhile, we also record the position number of each
entry in its list.
When doing query, we only have to perform binary search on the root list, which takes \(O(\log n)\) time. After that, each time we visit a new node on the path of \(O(\log n)\) length. We can find the intervals in \(O(1)\) time by following the pointers. In addition, at each level, we just subtract the position number and add the result to the total numbers. So the query operation can report the number of intervals in \(O(\log n)\) time.

**Preprocessing:** To build an interval tree with sorted lists requires \(O(n \log n)\) time. To add a pointer from the parent list entry to the child list entry can be done in \(O(\log n)\) time, because the list is sorted and we can perform a binary search to decide the correct entry to receive the pointer. Thus the total time for building up the list is bounded by \(O(n \log n)\). So, the pre-processing time is still in \(O(n \log n)\).

**Storage:** The interval tree takes \(O(n)\) space. So after adding the \(O(n)\) pointers, the space is still bounded by \(O(n)\).

**Query:** As explained above, it's bounded by \(O(\log n)\).

(c) When pre-processing, we can divide the real line into \(O(n)\) elementary intervals, and record the number of intervals that covers each elementary. Then, instead of using segment tree, we can build a B.S.T. on the elementary intervals. When doing query on point \(p\), we can detect the elementary interval that contains \(p\) with binary search in \(O(\log n)\) time, and report the coverage number in \(O(1)\) time. So the total query time is \(O(\log n)\). As the number of leaves is bounded by \(O(n)\), the storage of the B.S.T. is bounded by \(O(n)\).

10.7

(a) We can adopt the segment tree structure here. Everything is identical as described in Section 10.3. Thus the storage is bounded by \(O(n \log n)\), according to Theorem 10.13.

The difference is in the query strategy: when doing query on the \(O(\log n)\) canonical subsets along the search path, we don’t do binary search to detect where \(q_y\) is for each subset. Instead, we start from the top most segment, and go down until the first segment that is below \(q_y\). In this way, the query at each subset is bounded by \(O(1+k_s)\). To go through all the \(O(\log n)\) subsets, we thus require \(O(\log n+k)\) query time.

(b) This problem is identical to Problem 6.16 in Homework 5. We can use a trapezoidal decomposition as explained there. The storage is bounded by \(O(n)\), and the query time is bounded by \(O(\log n)\). Please refer to Homework 5 Solution Set for details.
10.9

An equivalent one to this problem is: given a set of left-right endpoint pairs \( \{l_i, r_i\} \), find those lying inside \([x, x']\) or, \(x \leq l_i \leq x'\) and \(x \leq r_i \leq x'\) are both satisfied. So we can build a 2-dim data structure, in which the main tree is a 1-dim range tree sorted on the left endpoints of the \(n\) intervals, and each node in the tree has a canonical list storing the intervals sorted on its right endpoints. For the canonical lists, we adopt the fractional cascading technology. The framework is shown below:

![Diagram showing a 2-dim range tree with fractional cascading]

When doing query, we first search the main tree to find \(O(\log n)\) nodes whose associated intervals all have left endpoints between \([x, x']\); then, we check the \(O(\log n)\) canonical lists, and report all intervals that has right endpoint between \([x, x']\). Thus we find all intervals that are contained in \([x, x']\).

According to Theorem 5.11, a 2-dim range tree with fractional cascading uses \(O(n\log n)\) storage, and 2-dim range query on left and right endpoints can be done in \(O(\log n + k)\) time.

10.10

We can build a priority tree with similar trick to that explained in Figure 10.6. Major priority is the coordinate of the left endpoints of the intervals, while the branching of sub-trees is arranged according to the coordinate of the right endpoints of the intervals: we always group the intervals with larger right endpoints to the right sub-tree. An example is shown above.

Given query interval \([x: x']\):

1) we can march from root until bottom, and find a path whose right sub-tree always store intervals with larger right endpoints than \(x'\). This action is similar to that shown in Figure 10.7, and can be done in \(O(\log n)\) time. Then, for each node on the path, we check if its associated interval contains \([x: x']\). If
so, report it; otherwise, go head. This can be done in O(logn) time.

(2) For each right sub-tree of the nodes on path, we can call “ReportInSubtree(v, x)” (Page 229) recursively to find all intervals with left endpoints larger than x. For each sub-tree, this can be done is O(1+kv) time. So all together, this part can be done in O(k) time.

The two processes explained above is shown by the figure below:

Therefore in general, the query can be done in O(logn+k) time. The data structure is just a priority tree, each node of which stores an interval. Thus the storage is bounded by O(n).

Reference