CS345: Quiz-1 Solution

Question 1 - Part 1

1. *CompleteSet* = Set of all intervals to be checked
2. *ResultSet* = {} (This will eventually consist of all the intervals to be output)
3. *UncoveredSet* = *CompleteSet* (This will eventually be empty).
4. while (*UncoveredSet* is not empty):
   - Select the interval (say I) from *UncoveredSet* with the minimum finish time.
   - Let *IntersectWithI* be the subset of intervals in *CompleteSet* which intersect with I (including I). Select the interval (say J) from *IntersectWithI* with max finish time.
   - Add J to the *ResultSet*.
   - Remove all intervals that intersect with J (including I and J) from *UncoveredSet*.
5. Return *ResultSet*.

Question 1 - Part 2

1. *CompleteSet* = Set of all intervals to be checked
2. *ResultSet* = {} (This will eventually consist of all the intervals to be output)
3. *UncoveredSet* = *CompleteSet* (This will eventually be empty).
4. while (*UncoveredSet* is not empty):
   - Select the interval (say I) from *UncoveredSet* with the minimum finish time.
   - Add I to the *ResultSet*.
   - Remove all intervals that intersect with I (including I) from *UncoveredSet*.
5. Return *ResultSet*.

Question 2 - Part 1

Note: For simplicity, we assume that all intervals have different low and high values.

Intuition: Let the query interval be (X_0, X_1). We can partition all the intervals stored in the given interval tree T into 3 sets: intervals with low value X satisfying $X < X_0$, intervals with low value X satisfying $X_0 < X < X_1$ and intervals with low value X satisfying $X > X_1$. Note that, none of the intervals with low value X satisfying $X > X_1$ will intersect the given query interval. Also, each interval with low value X satisfying $X_0 < X < X_1$ intersect with the query interval, and we can easily report them by performing successor queries repeatedly spending $O(\log n)$ time per interval that is reported (In fact we can do even better but that is not important for this problem). The only case left is the intervals with low value X satisfying $X < X_0$. Now make use of the following insight:

If an interval tree T is queried with an interval I such that low(I) is bigger than low value of each interval in T, then we can efficiently report all intervals that intersect I (by spending $O(\log n)$ time per interval reported).

Here are the arguments to convince you about this insight: We just query MaxHigh value stored at current node and proceed recursively down the subtree only if MaxHigh value of the subtree is larger than low value of the query interval. This insight ensures that we spend $O(\log n)$ time per interval reported. Now there will be $O(\log n)$ subtrees in T storing intervals with $X < X_0$. We can identify them using predecessor query for X_0. In this way we report the intervals from these subtrees that intersect with the query interval. The bound $O(k \log n)$ follows immediately.

Attached is the code for more details.

```
ResultSet = {}
#Finds all intervals intersecting with I in T.
def AllIntersect(T, I):
    FindRightIntervals(T, I)
    Pred = Predecessor(T, I)
    if (Pred == null):
        Pred = I
    FindLeftIntervals(T, I, Pred)
    return ResultSet

#Finds all intervals intersecting with I in T, with T.x > I.x.
def FindRightIntervals(T, I):
    S = Successor(T,I)
    while (S != null and S.low <= I.high):
        ResultSet.add(S)
        S = Successor(T, S)

#Finds all intervals intersecting with I in T, with T.x < I.x
```

```
def FindLeftIntervals(T, I, Pred):
    if (T == null):
        return null
    if (T.low >= Pred.low):
        if (T intersects I)
            ResultSet.add(I)
        FindLeftIntervals(T.left, I, Pred)
    else:
        if (T intersects I):
            ResultSet.add(T)

        if (T.left != null && T.left.maxHigh > I.low):
            FindLeftIntervals(T.left, I, Pred)
        if (T.right != null && T.right.maxHigh > I.low):
            FindLeftIntervals(T.right, I, Pred)
```

Question 2 - Part 2 (a)

This question was directly from Lecture. But for the sake of completeness, we provide its solution as well.

Intuition: Overlap(T,I) needs to return any interval intersecting with I, and null if none exist. It is easy to see that if the root(T) doesn't intersect with I, either $T.y < I.x$ or $I.y < T.x$. If $I.y < T.x$, it cannot possibly intersect with any interval in the right child of T (as the interval tree is ordered on x). If, however, $T.y < I.x$, a left descendant of T (say R) could intersect with I(If $R.y > I.x$). Note that, if $T.left.maxY > I.x$, we are assured of an intersection in the left subtree. We can make no such guarantees on the right child of T. Thus, we check both children in this order.

```
def Overlap(T, I):
  if (T == null):
     return null
  if (T intersects I):
     return T
  elif (I.finish < T.start):
     return Overlap(T.left, I)
  else:
     if (T.left != null and T.left.maxFinish > I.start):
        return Overlap(T.left, I)
     else
        return Overlap(T.right, I)
```

Question 2 - Part 2 (b)

Intuition: Insert(T, I) is exactly similar to a node insertion in an augmented binary search tree. The only additional care which needs to be taken is to ensure that the augmented attribute update can be done efficiently. This is done, in this case, by showing that the augmented attribute (maxFinish) can be updated in O(1) time on performing a rotation.

```
def Insert(T, I):
  #Used to preserve parent info of eventual insertion position.
  Y = null
  X = T

  while (X != null):
    X.maxFinish = max(X.maxFinish, I.finish)
    Y = X
    if (X.start > I.start):
       X = X.left
    else:
       X = X.right

    if (Y == null):
       T = new Node(start = I.start, finish = I.finish, maxFinish =
I.finish, left = null, right = null, parent = null)
    else:
      if (Y.start > I.start):
          Y.left = new Node(start = I.start, finish = I.finish, maxFinish =
I.finish, left = null, right = null, parent = Y)
          Balance(T, Y.left)
      else:
          Y.right = new Node(start = I.start, finish = I.finish, maxFinish =
I.finish, left = null, right = null, parent = Y)
          Balance(T, Y.right)
```

Consider a left rotation for balancing the tree, say LeftRotate(T, X). We need to prove that maxFinish updation takes O(1) time.

```
def LeftRotate(T,X):
    Y = X.right
    X.right = Y.left
    if (Y.left != null):
        Y.left.parent = X
    Y.parent = X.parent
    if (X.parent == null):
        T = y
    elif (X.parent.left == X):
        X.parent.left = Y
    else:
```

```
    X.parent.right = Y
Y.left = X
X.parent = Y
Y.maxFinish = X.maxFinish
X.maxFinish = X.finish
if (X.left != null && X.maxFinish < X.left.maxFinish):
    X.maxFinish = X.left.maxFinish
if (X.right != null && X.maxFinish < X.right.maxFinish):
    X.maxFinish = X.right.maxFinish
```