Frequent Values

http://www.spoj.com/problems/FREQUENT/

Problem Statement

- You are given a sequence of **n integers a_1 , a_2 , ... , a_n** in non-decreasing order. In addition to that, you are given several **queries consisting of indices i and j (1 ≤ i ≤ j ≤ n)**. For each query, determine the most frequent value among the integers ai , ... , aj.

- That's what it says, at least. The example question shows that what they really want is **the number of occurrences of the most frequent value in the query range**

Input

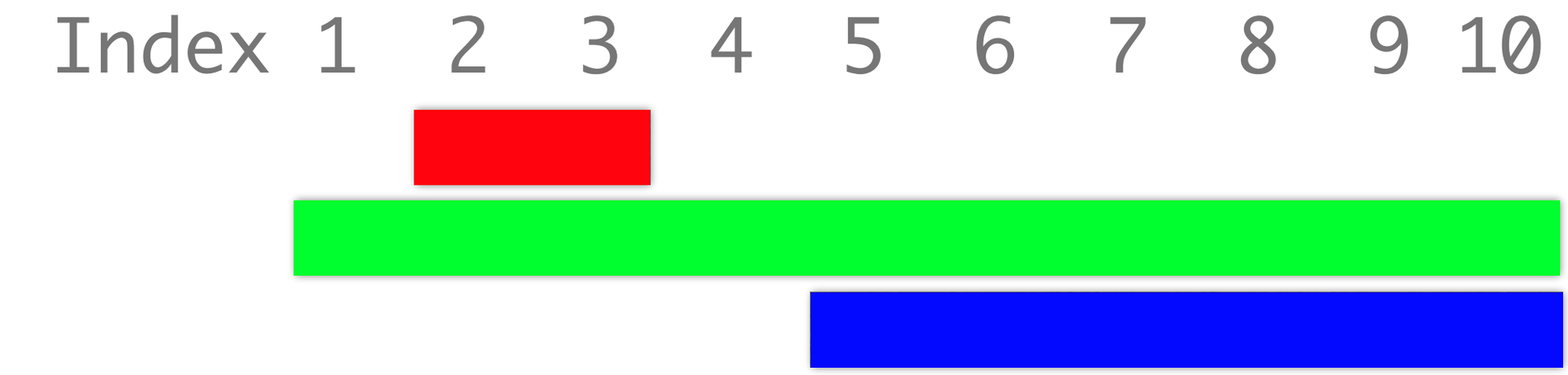
- The input consists of several test cases
- Each test case starts with a line containing two integers n and q ($1 \leq n, q \leq 100000$)
- The next line contains n integers a1 , ... , an ($-100000 \leq ai \leq 100000$, for each $i \in \{1, ..., n\}$) separated by spaces
- You can assume that for each $i \in \{1, ..., n-1\}$: $ai \leq ai+1$
- The following q lines contain one query each, consisting of two integers i and j ($1 \leq i \leq j \leq n$), which indicate the boundary indices for the query.
- The final test case is followed by a zero

- Upper limit for n, q is about 100,000. Rules out solutions of n^2 or worse

Example Problem

```
10 3
-1 -1 1 1 1 1 3 10 10 10
2 3
1 10
5 10
0
```

```
1
4
3
```



Analysis

- There is an obvious n^2 solution, but doesn't work well on large input sizes

- As seen in the previous graphic, queries can be intuitively modeled as intervals over a number line

  - So can repeated values!

  - Solutions are the maximum length of an intersection between a query interval and the repeated value intervals

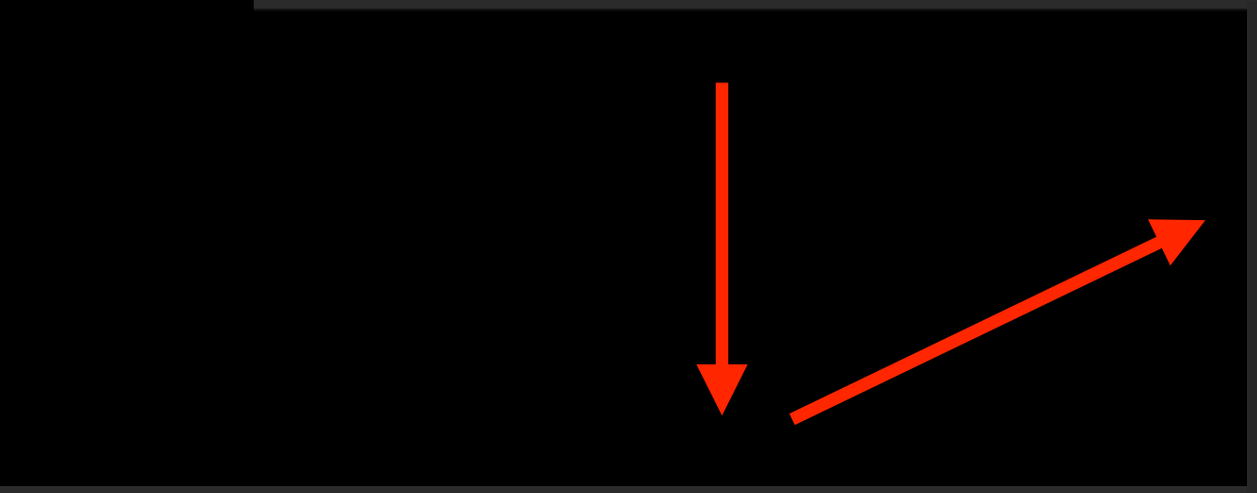
Interval Tree

- Interval trees are optimized to answer the question "Which intervals, from a set, intersect a given interval?"
- Balanced binary tree
- In addition to children, each node contains:
  1. A point on the number line
  2. A List of Intervals in the set which intersect with that point
- If a node has point 'p', its left child must have a point less than p, and its right child must have a point greater than p

Interval Tree - Construction

- Interval trees act like quick sort, partitioning space by repeatedly picking a point, and sorting all intervals into three categories:
  1. Completely left of the point
  2. Completely right of the point
  3. Intersecting that point
- Eventually, all intervals intersect one of the points in the tree
- Selecting these points effectively is important
  - one strategy: average of all endpoints

```java
Node root;
IntervalTree(LinkedList<Interval> intervals) {
    if (!intervals.isEmpty()) // avoid div by 0 error when getting average
        root = new Node(intervals); //Node constructor handles heavy lifting
}
```



```java
static class Node {

    LinkedList<Interval> ltor = new LinkedList<>()
    LinkedList<Interval> rtol = new LinkedList<>()
    Node r;
    Node l;
    int point;
```

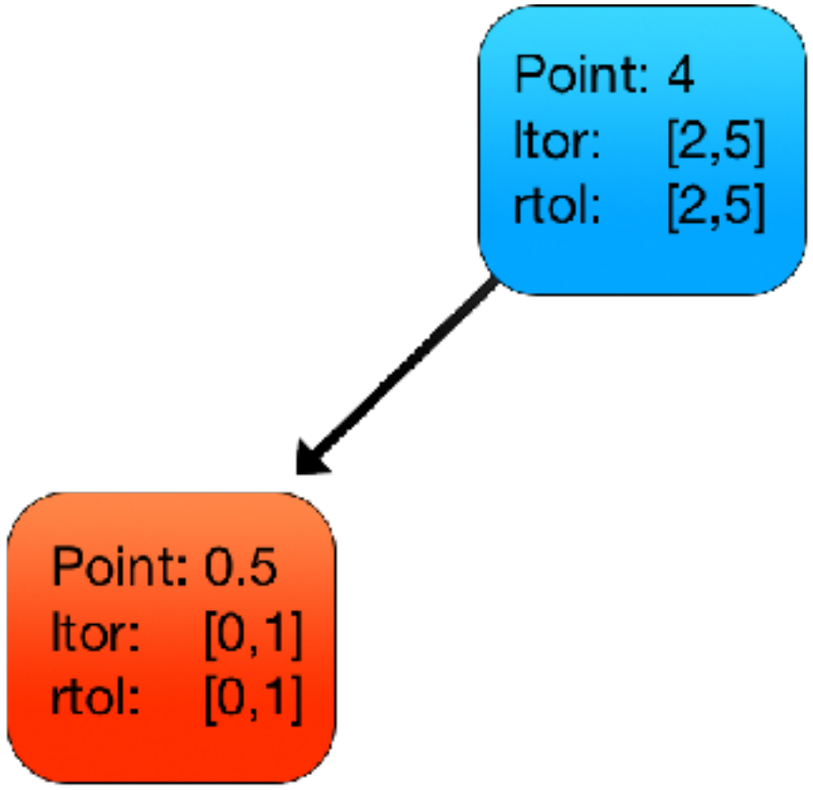
```java
public Node(LinkedList<Interval> intervals) {
    //get the average of the endpoints
    this.point = averageOfEndPoints(intervals);
    //sort intervals into three categories:
    LinkedList<Interval> left = new LinkedList<>();
    LinkedList<Interval> right = new LinkedList<>();
    while (!intervals.isEmpty()) {
        Interval i = intervals.remove();
        if (i.containsPoint(point)) {
            ltor.add(i);
            rtol.add(i);
        }
        else if (i.r < point) {
            left.add(i);
        }
        else /*if (i.l > point)*/ {
            right.add(i);
        }
    }
    //sort the intersecting list by
    //  increasing left endpoint
    //  decreasing right endpoint
    ltor.sort((Interval a, Interval b) ->
            Integer.compare(a.l, b.l));
    rtol.sort((Interval a, Interval b) ->
            -Integer.compare(a.r, b.r));

    //create children
    if (!right.isEmpty()) r = new Node(right);
    if (!left.isEmpty()) l = new Node(left);
}
```

Interval Tree - Construction

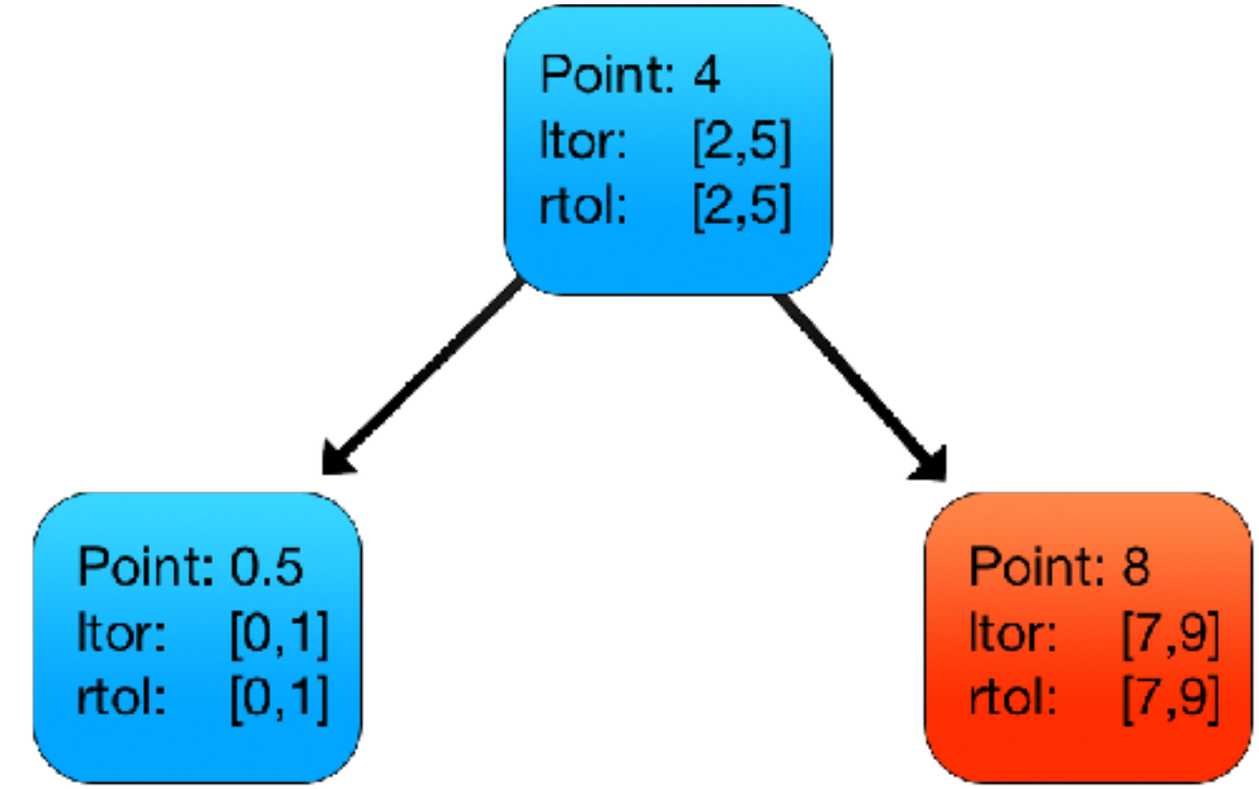
[0,1] [2,5] [7,9]

Interval Tree - Construction



[0,1]

Interval Tree - Construction

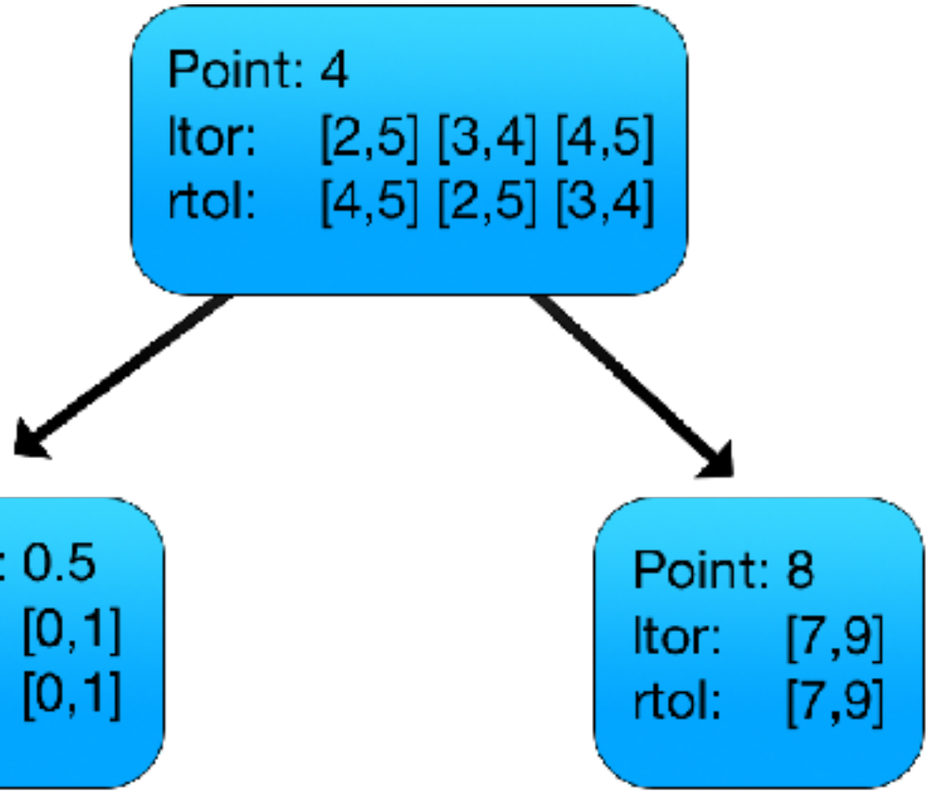


[7,9]

Interval Tree - Construction

- Why is it necessary to have **ltor** and **rtol**?
- The following example and the querying code will make it clear

Interval Tree - Construction 2



[0,1] [2,5] [3,4] [4,5] [7,9]

Interval Tree - Querying

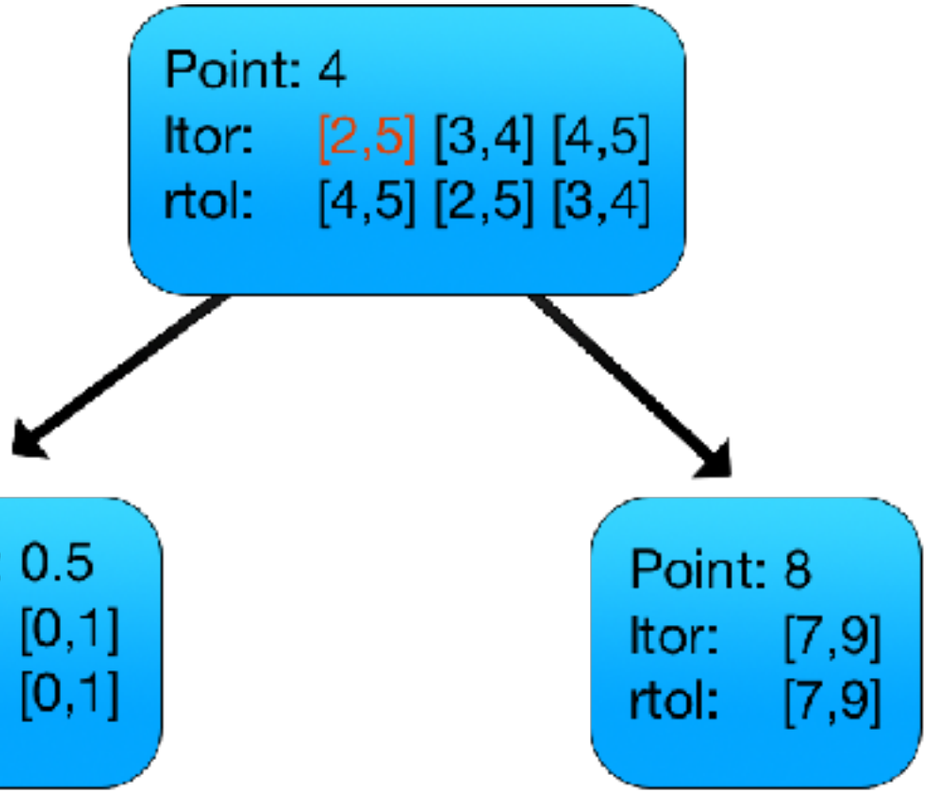
- Given an interval, find all interval in the tree which intersect it

1. Get intervals at a node
   - If you intersect a node's point, you can add all its intervals to your set
   - If not, peel away from ltor or rtol, depending on which side of the point the interval is on
2. Recurse through child..
   - R if the interval extends to the right of point
   - L if the interval extends to the left of point

- Efficiency is O(log(n) + m)
   - n: intervals in tree
   - m: intervals in tree that intersect your query

```
LinkedList<Interval> intersectingIntervals(Interval i) {
    LinkedList<Interval> set = new LinkedList<>();
    addIntersectingIntervals(i, set, root);
    return set;
}
void addIntersectingIntervals(Interval i, LinkedList<Interval> set, Node current) {
    if (current != null) {
        if (i.containsPoint(current.point)) {
            set.addAll(current.ltor); //same content as rtol, just a different ordering
        }
        else {
            ListIterator<Interval> it;
            if (current.point < i.l) //interval is to the right
                it = current.rtol.listIterator();
            else //interval is to the left
                it = current.ltor.listIterator();
            while (it.hasNext()) {
                Interval temp = it.next();
                if (i.intersects(temp)) set.add(temp);
                else break; //nothing after this one will intersect
            }
        }
        if (i.l < current.point)
            addIntersectingIntervals(i, set, current.l);
        if (i.r > current.point)
            addIntersectingIntervals(i, set, current.r);
    }
}
```

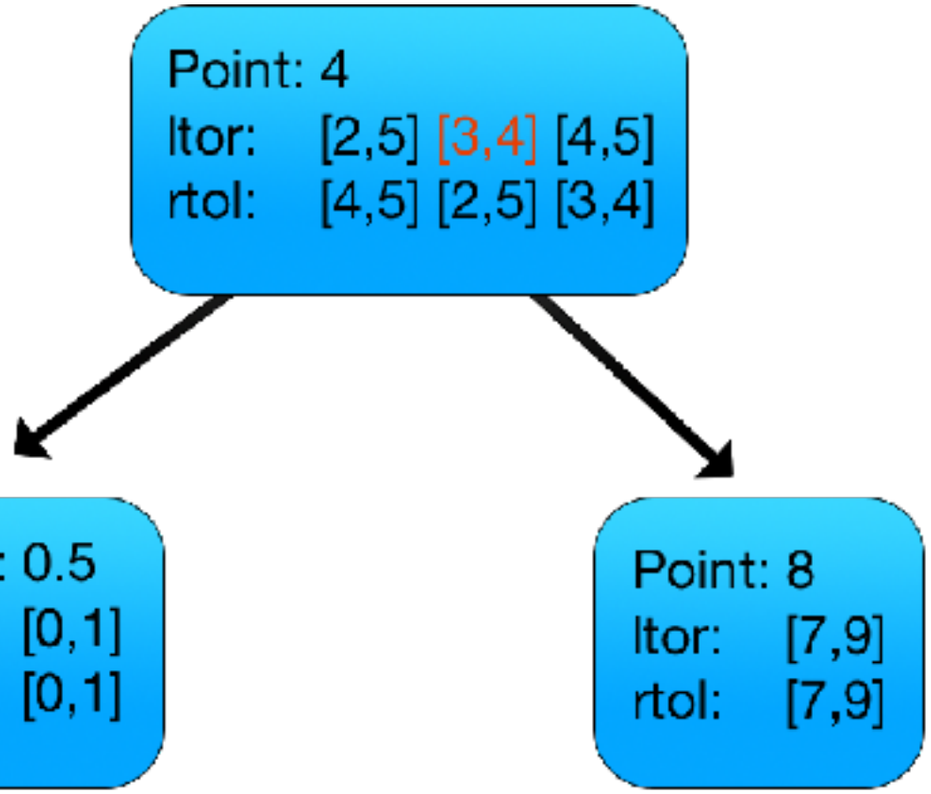
Interval Tree - Querying



Query: [1,2]
Solution Set: [2,5]

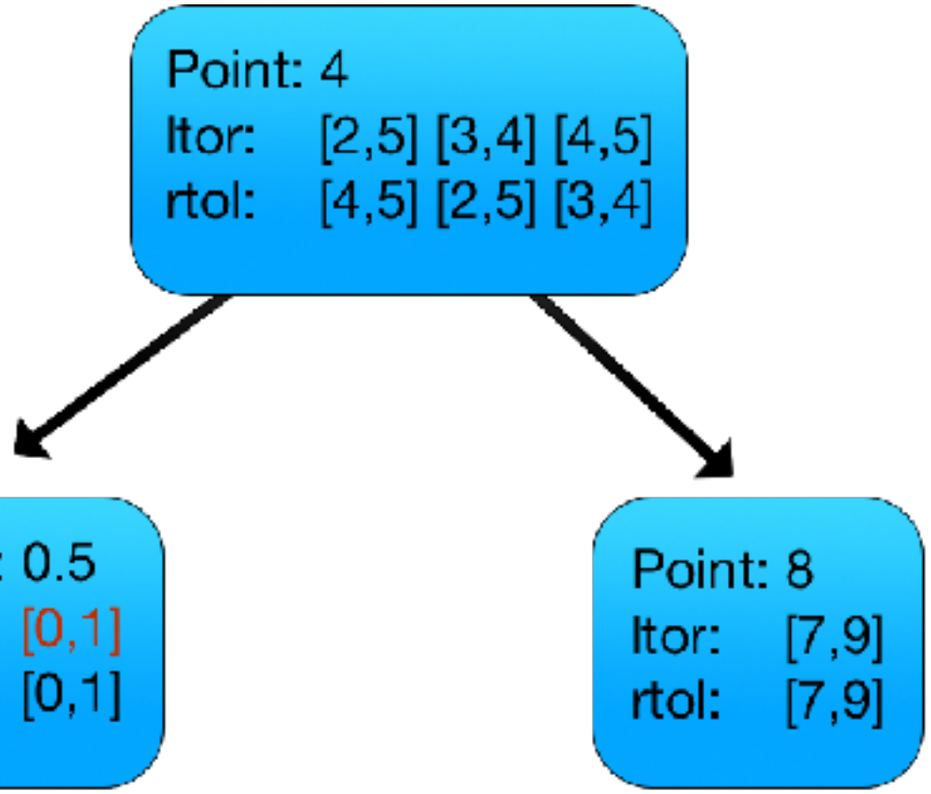
Interval Tree - Querying

Because [1,2]and [3,4] don't intersect, we don't need to check [4,5]!



Query: [1,2]
Solution Set: [2,5]

Interval Tree - Querying



Query: [1,2]

Solution Set: [2,5] [0,1]

Problem-Specific Notes

- No two intervals that go in the tree overlap
  - makes rtol and ltor unnecessary

- Once you get the set of intervals that overlap your query, you need to find the size of their intersection
  - example: ‘5’ repeats through interval [4,9], but my query only considers [3,7]. The correct answer is 4, not 6

- You know that all solutions are >=1. You can halve the time of building the tree by only constructing it after throwing out all intervals of length one

Main() Code

```
public static void main (final String[] args) throws IOException {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
    while (true) {
        //get the data
        StringTokenizer st = new StringTokenizer(in.readLine());
        int size = Integer.parseInt(st.nextToken());
        if (size == 0) break; //end of tests
        int queries = Integer.parseInt(st.nextToken());
        int[] values = new int[size];
        st = new StringTokenizer(in.readLine());
        for (int i = 0; i < size; i++) {
            values[i] = Integer.parseInt(st.nextToken());
        }
        //get all nontrivial (size > 1) intervals of consecutive values
        LinkedList<Interval> intervals = new LinkedList<>();
        int base = 0;
        while (base < size) {
            int current = base + 1;
            while (current < size) {
                if (values[current] == values[base]) current++;
                else break;
            }
            if (base-current > 1)
                intervals.add(new Interval(base,  right: current-1));
            base = current;
        }
```

More Main() Code

```
        //create the interval tree
        IntervalTree tree = new IntervalTree(intervals);
        //process the queries
        for (int q = 0; q < queries; q++) {
            st = new StringTokenizer(in.readLine());
            int ql = Integer.parseInt(st.nextToken()) - 1;
            int qr = Integer.parseInt(st.nextToken()) - 1;
            Interval query = new Interval(ql, qr);
            LinkedList<Interval> intersecting = tree.intersectingIntervals(query);
            int bestLength = -1;
            for (Interval i : intersecting) {
                Interval intersection = query.intersection(i);
                if (intersection.length() > bestLength) {
                    bestLength = intersection.length();
                }
            }
            out.write(Integer.toString(bestLength));
            out.write( str: "\n");
        }
    }
}
```