Previous Next

436. Find Right Interval

March 16, 2017 | 5.3K views

Average Rating: 4.08 (13 votes)

Given a set of intervals, for each of the interval i, check if there exists an interval j whose start point is bigger than or equal to the end point of the interval i, which can be called that j is on the "right" of i.

For any interval i, you need to store the minimum interval j's index, which means that the interval j has the minimum start point to build the "right" relationship for interval i. If the interval j doesn't exist, store -1 for the interval i. Finally, you need output the stored value of each interval as an array.

Note:

1. You may assume the interval's end point is always bigger than its start point.
2. You may assume none of these intervals have the same start point.

Example 1:

```
Input: [ [1,2] ]

Output: [-1]

Explanation: There is only one interval in the collection, so it outputs -1.
```

Example 2:

```
Input: [ [3,4], [2,3], [1,2] ]

Output: [-1, 0, 1]

Explanation: There is no satisfied "right" interval for [3,4].
For [2,3], the interval [3,4] has minimum-"right" start point;
For [1,2], the interval [2,3] has minimum-"right" start point.
```

Example 3:

```
Input: [ [1,4], [2,3], [3,4] ]

Output: [-1, 2, -1]

Explanation: There is no satisfied "right" interval for [1,4] and [3,4].
For [2,3], the interval [3,4] has minimum-"right" start point.
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

Solution

Approach 1: Brute Force

The simplest solution consists of picking up every interval in the set and looking for the the interval whose start point is larger(by a minimum difference) than the chosen interval's end point by scanning the complete set for every interval chosen. While scanning, we keep a track of the interval with the minimum start point satisfying the given criteria along with its index. The result obtained for every interval chosen is stored at the corresponding index in the res array which is returned at the end.

Java

```java
class Solution {
    public int[] findRightInterval(int[][] intervals) {
        int[] res = new int[intervals.length];
        for (int i = 0; i < intervals.length; i++) {
            int min = Integer.MAX_VALUE;
            int minindex = -1;
            for (int j = 0; j < intervals.length; j++) {
                if (intervals[j][0] >= intervals[i][1] && intervals[j][0] < min) {
                    min = intervals[j][0];
                    minindex = j;
                }
            }
            res[i] = minindex;
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n^2)$. The complete set of n intervals is scanned for every(n) interval chosen.
- Space complexity : $\mathcal{O}(n)$. res array of size n is used.

Approach 2: Using Sorting + Scanning

We make use of a hashmap hash, which stores the data in the form of a $(\text{Key}, \text{Value})$ pair. Here, the Key corresponds to the interval chosen and the Value corresponds to the index of the particular interval in the given intervals array. We store every element of the intervals array in the $hash$-map.

Now, we sort the intervals array based on the starting points. We needed to store the indices of the array in the hashmap, so as to be able to obtain the indices even after the sorting.

Now, we pick up every interval of the sorted array, and find out the interval from the remaining ones whose start point comes just after the end point of the interval chosen. How do we proceed? Say, we've picked up the i^{th} interval right now. In order to find an interval satisfying the given criteria, we need not search in the intervals behind it. This is because the intervals array has been sorted based on the starting points and the end point is always greater than the starting point for a given interval. Thus, we search in the intervals only with indices j, such that $i+1 < j < n$. The first element encountered while scanning in the ascending order is the required result for the interval chosen, since all the intervals lying after this interval will have comparatively larger start points.

Then, we can obtain the index corresponding to the corresponding interval from the hashmap, which is stored in the corresponding entry of the res array. If no interval satisfies the criteria, we put a -1 in the corresponding entry.

Java

```java
class Solution {
    public int[] findRightInterval(int[][] intervals) {
        int[] res = new int[intervals.length];
        Map<int[], Integer> hash = new HashMap<>();
        for (int i = 0; i < intervals.length; i++) {
            hash.put(intervals[i], i);
        }
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        for (int i = 0; i < intervals.length; i++) {
            int min = Integer.MAX_VALUE;
            int minindex = -1;
            for (int j = i + 1; j < intervals.length; j++) {
                if (intervals[j][0] >= intervals[i][1] && intervals[j][0] < min) {
                    min = intervals[j][0];
                    minindex = hash.get(intervals[j]);
                }
            }
            res[hash.get(intervals[i])] = minindex;
        }
        return res;
    }
}

```

Complexity Analysis

- Time complexity : $O(n^2)$.
  - Sorting takes $O\big(n\log n\big)$ time.
  - For the first interval we need to search among $n-1$ elements.
  - For the second interval, the search is done among $n-2$ elements and so on leading to a total of: $(n-1)+(n-2)+...+1 = \frac{n.(n-1)}{2} = O(n^2)$ calculations.
- Space complexity : $O(n)$. res array of size n is used. A hashmap hash of size n is used.

Approach 3: Using Sorting + Binary Search

We can optimize the above approach to some extent, since we can make use of the factor of the intervals array being sorted. Instead of searching for the required interval in a linear manner, we can make use of Binary Search to find an interval whose start point is just larger than the end point of the current interval.

Again, if such an interval is found, we obtain its index from the hashmap and store the result in the appropriate res entry. If not, we put a -1 at the corresponding entry.

Java

```java
public class Solution {

  public int[] binary_search(int[][] intervals, int target, int start, int end) {
    if (start >= end) {
      if (intervals[start][0] >= target) {
        return intervals[start];
      }
      return null;
    }
    int mid = (start + end) / 2;
    if (intervals[mid][0] < target) {
      return binary_search(intervals, target, mid + 1, end);
    } else {
      return binary_search(intervals, target, start, mid);
    }
  }

  public int[] findRightInterval(int[][] intervals) {
      int[] res = new int[intervals.length];
      HashMap<int[], Integer> hash = new HashMap<>();
      for (int i = 0; i < intervals.length; i++) {
          hash.put(intervals[i], i);
      }
      Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
      for (int i = 0; i < intervals.length; i++) {
          int[] interval = binary_search(intervals, intervals[i][1], 0, intervals.length - 1);
          res[hash.get(intervals[i])] = interval == null ? -1 : hash.get(interval);
```

Complexity Analysis

- Time complexity : $O\big(n\log n\big)$. Sorting takes $O\big(n\log n\big)$ time. Binary search takes $O\big(\log n\big)$ time for each of the n intervals.
- Space complexity : $O(n)$. res array of size n is used. A hashmap hash of size $O(n)$ is used.

Approach 4: Using TreeMap

In this approach, instead of using a hashmap, we make use of a TreeMap starts, which is simply a Red-Black Tree(a kind of balanced Binary Search Tree) . This Treemap start stores data in the form of $(\text{Key}, \text{Value})$ pair and always remain sorted based on its keys. In our case, we store the data such that the start point of an interval acts as the Key and the index corresponding to the interval acts as the value, since we are concerned with data sorted based on the start points, as discussed in previous approaches. Every element of the intervals array is stored in the TreeMap.

Now, we choose each element of the intervals array and make use of a function `TreeMap.ceilingEntry(end_point)` to obtain the element in the TreeMap with its Key just larger than the end_point of the currently chosen interval. The function `ceilingEntry(Key)` returns the element just with its Key larger than the `Key` (passed as the argument) from amongst the elements of the TreeMap and returns `null` if no such element exists.

If non-null value is returned, we obtain the Value from the $(\text{Key}, \text{Value})$ pair obtained at the appropriate entry in the res array. If a null value is returned, we simply store a -1 at the corresponding res entry.

Java

```java
public class Solution {

  public int[] findRightInterval(int[][] intervals) {
    TreeMap<Integer, Integer> starts = new TreeMap<>();
    int res[] = new int[intervals.length];
    for (int i = 0; i < intervals.length; i++) {
      starts.put(intervals[i][0], i);
    }
    for (int i = 0; i < intervals.length; i++) {
      Map.Entry<Integer, Integer> pos = starts.ceilingEntry(intervals[i][1]);
      res[i] = pos == null ? -1 : pos.getValue();
    }
    return res;
  }
}
```

Complexity Analysis

- Time complexity : $O\big(N \cdot \log N\big)$. Inserting an element into TreeMap takes $O\big(\log N\big)$ time. N such insertions are done. The search in TreeMap using `ceilingEntry` also takes $O\big(\log N\big)$ time. N such searches are done.
- Space complexity : $O(N)$. res array of size n is used. TreeMap starts of size $O(N)$ is used.

Approach 5: Using Two Arrays without Binary Search

Algorithm

The intuition behind this approach is as follows: If we maintain two arrays,

1. intervals, which is sorted based on the start points.
2. endIntervals, which is sorted based on the end points.

Once we pick up the first interval(or, say the i^{th} interval) from the endIntervals array, we can determine the appropriate interval satisfying the right interval criteria by scanning the intervals in intervals array from left towards the right, since the intervals array is sorted based on the start points. Say, the index of the element chosen from the intervals array happens to be j.

Now, when we pick up the next interval(say the $(i+1)^{th}$ interval) from the endIntervals array, we need not start scanning the intervals array from the first index. Rather, we can start off directly from the j^{th} index where we left off last time in the intervals array. This is because end point corresponding to endIntervals[i+1] is larger than the one corresponding to endIntervals[i] and none of the intervals from intervals[k], such that $0 < k < j$, satisfies the right neighbor criteria with endIntervals[i], and hence not with endIntervals[i+1] as well.

If at any moment, we reach the end of the array i.e. $j = \text{intervals.length}$ and no element satisfying the right interval criteria is available in the intervals array, we put a -1 in the corresponding res entry. The same holds for all the remaining elements of the endIntervals array, whose end points are even larger than the previous interval encountered.

Also we make use of a hashmap hash initially to preserve the indices corresponding to the intervals even after sorting.

For more understanding see the below animation:

Java

```java
public class Solution {

  public int[] findRightInterval(int[][] intervals) {
    int[][] endIntervals = Arrays.copyOf(intervals, intervals.length);
    HashMap<int[], Integer> hash = new HashMap<>();
    for (int i = 0; i < intervals.length; i++) {
      hash.put(intervals[i], i);
    }
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
    Arrays.sort(endIntervals, (a, b) -> a[1] - b[1]);
    int j = 0;
    int[] res = new int[intervals.length];
    for (int i = 0; i < endIntervals.length; i++) {
      while (j < intervals.length && intervals[j][0] < endIntervals[i][1]) {
        j++;
      }
      res[hash.get(endIntervals[i])] = j == intervals.length ? -1 : hash.get(intervals[j]);
    }
    return res;
  }
}

```

Complexity Analysis

- Time complexity : $O\big(N \cdot \log N\big)$. Sorting takes $O\big(N \cdot \log N\big)$ time. A total of $O(N)$ time is spent on searching for the appropriate intervals, since the endIntervals and intervals array is scanned only once.
- Space complexity : $O(N)$. endIntervals, intervals and res array of size N are used. A hashmap hash of size $O(N)$ is used.

Rate this article: