<https://leetcode.com/problems/find-right-interval/description/>

You are given an array of intervals, where intervals[i] = [starti, endi] and each starti is **unique**.

The **right interval** for an interval i is an interval j such that startj >= endi and startj is **minimized**. Note that i may equal j.

Return **an array of right interval indices for each interval i**. If no **right interval** exists for interval i, then put -1 at index i.

**Example 1:**

**Input:** intervals = [[1,2]]

**Output:** [-1]

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

**Example 2:**

**Input:** intervals = [[3,4],[2,3],[1,2]]

**Output:** [-1,0,1]

**Explanation:** There is no right interval for [3,4].

The right interval for [2,3] is [3,4] since start0 = 3 is the smallest start that is >= end1 = 3.

The right interval for [1,2] is [2,3] since start1 = 2 is the smallest start that is >= end2 = 2.

**Example 3:**

**Input:** intervals = [[1,4],[2,3],[3,4]]

**Output:** [-1,2,-1]

**Explanation:** There is no right interval for [1,4] and [3,4].

The right interval for [2,3] is [3,4] since start2 = 3 is the smallest start that is >= end1 = 3.

**Constraints:**

1 <= intervals.length <= 2 \* 10^4

intervals[i].length == 2

-10^6 <= starti <= endi <= 10^6

The start point of each interval is unique.

**Attempt 1: 2024-01-07**

**Solution 1: Binary Search (10min)**

class Solution {

    public int[] findRightInterval(int[][] intervals) {

        int n = intervals.length;

        List<int[]> list = new ArrayList<>();

        for(int i = 0; i < n; i++) {

            list.add(new int[] {intervals[i][0], intervals[i][1], i});

        }

        // Since each starti is unique, no need consider same value starti situation

        Collections.sort(list, (a, b) -> a[0] - b[0]);

        int[] result = new int[n];

        for(int i = 0; i < n; i++) {

            result[list.get(i)[2]] = binarySearch(list, list.get(i)[1]);

        }

        return result;

    }

    // Find lower boundary

    private int binarySearch(List<int[]> list, int val) {

        int lo = 0;

        int hi = list.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(list.get(mid)[0] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // No need condition 'list.get(lo)[0] != val'

        // Since we will only encounter two scenarios:

        // 1.When lo == list.size(), it means IndexOutOfBound, no more

        // element to scan and no list.get(lo)[0] >= val happen

        // 2.Even while loop end with lo > hi but 'list.get(lo)[0] != val'

        // is not a problem, since 'list.get(lo)[0] > val' also what we

        // are looking forward, that's guaranteed because at that moment

        // 'lo < list.size()' and 'lo > hi' and 'list.get(hi)[0] >= val'

        // all satisfied, with a sorted list, 'list.get(lo)[0] > val' also

        // satisfied for sure

        // Test out by:

        // Input: [[4,5],[2,3],[1,2]]

        // Output: [-1,-1,1] , Expect: [-1,0,1]

        //if(lo == list.size() || list.get(lo)[0] != val) {

        if(lo == list.size()) {

            return -1;

        }

        return list.get(lo)[2];

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Step by Step**

arr = [[3,4],[2,3],[1,2]]

add original index as arr[i][2]

-> list = [[3,4,0],[2,3,1],[1,2,2]]

sort based on arr[i][0], since each starti is unique, no need consider same value starti situation

-> list = [[1,2,2],[2,3,1],[3,4,0]]

Round 1:

Binary search startj of end2 as list[0][1] = 2 on list[0] = [1,2,2]

expect return 1 since list[1][0] = 2 is the smallest start that is >= end2 = 2

result[list[0][2]] = list[1][2] -> result[2] = 1

Round 2:

Binary search startj of end1 as list[1][1] = 3 on list[1] = [2,3,1]

expect return 0 since list[0][0] = 3 is the smallest start that is >= end1 = 3

result[list[1][2]] = list[0][2] -> result[1] = 0

Round 3:

Binary search startj of end0 as list[2][1] = 4 on list[2] = [3,4,0]

expect return -1 since no interval has the smallest start that is >= end0 = 4

result[list[2][2]] = -1 -> result[0] = -1

**Refer to**

<https://grandyang.com/leetcode/436/>

这道题给了我们一堆区间，让我们找每个区间的最近右区间，要保证右区间的 start 要大于等于当前区间的 end，由于区间的顺序不能变，所以我们不能给区间排序，我们需要建立区间的 start 和该区间位置之间的映射，由于题目中限定了每个区间的 start 都不同，所以不用担心一对多的情况出现。然后我们把所有的区间的 start 都放到一个数组中，并对这个数组进行降序排序，那么 start 值大的就在数组前面。然后我们遍历区间集合，对于每个区间，我们在数组中找第一个小于当前区间的 end 值的位置，如果数组中第一个数就小于当前区间的 end，那么说明该区间不存在右区间，结果 res 中加入-1；如果找到了第一个小于当前区间 end 的位置，那么往前推一个就是第一个大于等于当前区间 end 的 start，我们在 HashMap 中找到该区间的坐标加入结果 res 中即可，参见代码如下：

class Solution {

public:

vector<int> findRightInterval(vector<vector<int>>& intervals) {

vector<int> res, starts;

unordered\_map<int, int> m;

for (int i = 0; i < intervals.size(); ++i) {

m[intervals[i][0]] = i;

starts.push\_back(intervals[i][0]);

}

sort(starts.rbegin(), starts.rend());

for (auto interval : intervals) {

int i = 0;

for (; i < starts.size(); ++i) {

if (starts[i] < interval[1]) break;

}

res.push\_back((i > 0) ? m[starts[i - 1]] : -1);

}

return res;

}

};

上面的解法可以进一步化简，我们可以利用 STL 的 lower\_bound 函数来找第一个不小于目标值的位置，这样也可以达到我们的目标，参见代码如下：

class Solution {

public:

vector<int> findRightInterval(vector<vector<int>>& intervals) {

vector<int> res;

map<int, int> m;

for (int i = 0; i < intervals.size(); ++i) {

m[intervals[i][0]] = i;

}

for (auto interval : intervals) {

auto it = m.lower\_bound(interval[1]);

if (it == m.end()) res.push\_back(-1);

else res.push\_back(it->second);

}

return res;

}

};

**Refer to**

<https://algo.monster/liteproblems/436>

**Problem Description**

The goal of this problem is to find the "right interval" for a given set of intervals. Given an array intervals, where each element intervals[i] represents an interval with a start\_i and an end\_i, we need to identify for each interval i another interval j where the interval j starts at or after the end of interval i, and among all such possible j, it starts the earliest. This means that the start of interval j (start\_j) must be greater than or equal to the end of interval i (end\_i), and we want the smallest such start\_j. If there is no such interval j that meets these criteria, the corresponding index should be -1.

The challenge is to write a function that returns an array of indices of the right intervals for each interval. If an interval has no right interval, as mentioned -1 will be the placeholder for that interval.

**Intuition**

To approach this problem, we utilize a binary search strategy to efficiently find the right interval for each entry. Here's how we arrive at the solution:

To make it possible to return indices, we augment each interval with its original index in the intervals array.

We then sort the augmented intervals according to the starting points. The sort operation allows us to apply binary search later since binary search requires a sorted sequence.

Prepare an answer array filled with -1 to assume initially that there is no right interval for each interval.

For each interval, use binary search (bisect\_left) to efficiently find the least starting interval j that is greater than or equal to the end of the current interval i.

If the binary search finds such an interval, update the answer array at index i with the index of the found interval j.

After completing the search for all intervals, the answer array is returned.

Implementing binary search reduces the complexity of finding the right interval from a brute-force search which would be O(n^2) to O(n log n) since each binary search operation takes O(log n) and we perform it for each of the n intervals.

**Solution Approach**

The solution to this problem involves the following steps, which implement a binary search algorithm:

Augment Intervals with Indices: First, we update each interval to include its original index. This is achieved by iterating through the intervals array and appending the index to each interval.

for i, v in enumerate(intervals):

v.append(i)

Sort Intervals by Start Times: We then sort the augmented intervals based on their start times. This allows us to leverage binary search later on since it requires the list to be sorted.

intervals.sort()

Initialize Answer Array: We prepare an answer array, ans, with the same length as the intervals array and initialize all its values to -1, which indicates that initially, we assume there is no right interval for any interval.

ans = [-1] \* n

Binary Search for Right Intervals: For each interval in intervals, we perform a binary search to find the minimum start\_j value that is greater than or equal to end\_i. To do this, we use the bisect\_left method from the bisect module. It returns the index at which the end\_i value could be inserted to maintain the sorted order.

for \_, e, i in intervals:

j = bisect\_left(intervals, [e])

Updating the Answer: If the binary search returns an index less than the number of intervals (n), it means we have found a right interval. We then update the ans[i] with the original index of the identified right interval, which is stored at intervals[j][2].

if j < n:

ans[i] = intervals[j][2]

Return the Final Array: After the loop, the ans array is populated with the indices of right intervals for each interval in the given array. This array is then returned as the final answer.

return ans

This approach efficiently uses binary search to minimize the time complexity. The key to binary search is the sorted nature of the intervals after they are augmented with their original indices. By maintaining a sorted list of start times and employing binary search, we're able to significantly reduce the number of comparisons needed to find the right interval from linear (checking each possibility one by one) to logarithmic, thus enhancing the performance of the solution.

**Example Walkthrough**

Let's walk through this approach with a small example. Consider the list of intervals intervals = [[1,2], [3,4], [2,3], [4,5]].

First, we augment each interval with its index.

augmented\_intervals = [[1, 2, 0], [3, 4, 1], [2, 3, 2], [4, 5, 3]]

Next, we sort the augmented intervals by their start times.

sorted\_intervals = [[1, 2, 0], [2, 3, 2], [3, 4, 1], [4, 5, 3]]

We initialize the answer array with all elements set to -1.

ans = [-1, -1, -1, -1]

Now, using a binary search, we look for the right interval for each interval in sorted\_intervals.

For interval [1, 2, 0]:

The end time is 2.

The binary search tries to find the minimum index where 2 could be inserted to maintain the sorted order, which is index 1 (interval [2, 3, 2]).

Thus, the right interval index is 2.

For interval [2, 3, 2]:

The end time is 3.

The binary search finds index 2 (interval [3, 4, 1]).

The right interval index is 1.

For interval [3, 4, 1]:

The end time is 4.

The binary search finds index 3 (interval [4, 5, 3]).

The right interval index is 3.

For interval [4, 5, 3]:

The end time is 5, and there's no interval starting after 5.

The binary search returns an index of 4, which is outside the array bounds.

There's no right interval, so the value remains -1.

After performing the binary search for all intervals, we update the answer array with the right intervals' indices.

ans = [2, 1, 3, -1]

The ans array, which keeps track of the right interval for each interval, is returned as the final answer. In our example, this is [2, 1, 3, -1], indicating that the interval [1,2] is followed by [2,3], [3,4] follows [2,3], and [4,5] follows [3,4]. The last interval, [4,5], has no following interval that satisfies the conditions.

This approach efficiently finds the right intervals for each given interval with reduced time complexity.

**Java Solution**

class Solution {

public int[] findRightInterval(int[][] intervals) {

int numIntervals = intervals.length;

// This list will hold the start points and their corresponding indices

List<int[]> startIndexPairs = new ArrayList<>();

// Populate the list with the start points and their indices

for (int i = 0; i < numIntervals; ++i) {

startIndexPairs.add(new int[] {intervals[i][0], i});

}

// Sort the startIndexPairs based on the start points in ascending order

startIndexPairs.sort(Comparator.comparingInt(a -> a[0]));

// Prepare an array to store the result

int[] result = new int[numIntervals];

// Initialize an index for placing interval results

int resultIndex = 0;

// Loop through each interval to find the right interval

for (int[] interval : intervals) {

int left = 0, right = numIntervals - 1;

int intervalEnd = interval[1];

// Binary search to find the minimum start point >= interval's end point

while (left < right) {

int mid = (left + right) / 2;

if (startIndexPairs.get(mid)[0] >= intervalEnd) {

right = mid;

} else {

left = mid + 1;

}

}

// Check if the found start point is valid and set the result accordingly

result[resultIndex++] = startIndexPairs.get(left)[0] < intervalEnd ? -1 : startIndexPairs.get(left)[1];

}

// Return the populated result array

return result;

}

}

**Time and Space Complexity**

The time complexity of the provided code consists of two major operations: sorting the intervals list and performing binary search using bisect\_left for each interval.

Sorting the intervals list using the sort method has a time complexity of O(n log n), where n is the number of intervals.

Iterating over each interval and performing a binary search using bisect\_left has a time complexity of O(n log n) since the binary search operation is O(log n) and it is executed n times, once for each interval.

Thus, the combined time complexity of these operations would be O(n log n + n log n). However, since both terms have the same order of growth, the time complexity simplifies to O(n log n).

The space complexity of the code is O(n) for the following reasons:

The intervals list is expanded to include the original index position of each interval, but the overall space required is still linearly proportional to the number of intervals n.

The ans list is created to store the result for each interval, which requires O(n) space.

Apart from the two lists mentioned above, no additional space that scales with the size of the input is used.

Therefore, the total space complexity is O(n).