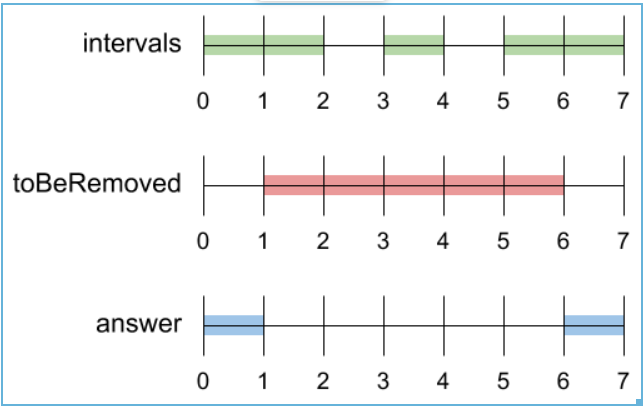
<https://www.cnblogs.com/cnoodle/p/13437387.html>

A set of real numbers can be represented as the union of several disjoint intervals, where each interval is in the form [a, b). A real number x is in the set if one of its intervals [a, b) contains x (i.e. a <= x < b).

You are given a sorted list of disjoint intervals intervals representing a set of real numbers as described above, where intervals[i] = [ai, bi] represents the interval [ai, bi). You are also given another interval toBeRemoved.

Return *the set of real numbers with the interval*toBeRemoved*removed from*intervals*. In other words, return the set of real numbers such that every*x*in the set is in*intervals*but not in*toBeRemoved*. Your answer should be a sorted list of disjoint intervals as described above.*

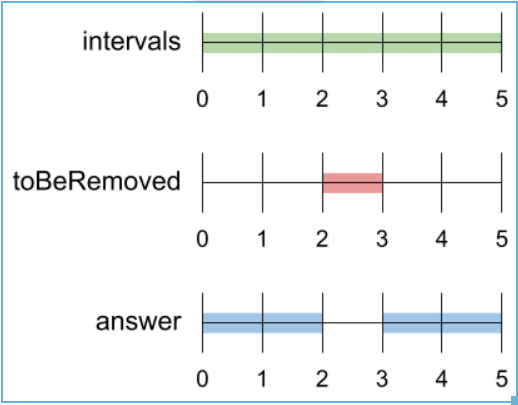
Example 1:



Input: intervals = [[0,2],[3,4],[5,7]], toBeRemoved = [1,6]

Output: [[0,1],[6,7]]

Example 2:



Input: intervals = [[0,5]], toBeRemoved = [2,3]

Output: [[0,2],[3,5]]

Example 3:

Input: intervals = [[-5,-4],[-3,-2],[1,2],[3,5],[8,9]], toBeRemoved = [-1,4]

Output: [[-5,-4],[-3,-2],[4,5],[8,9]]

Constraints:

* 1 <= intervals.length <= 10^4
* -10^9 <= intervals[i][0] < intervals[i][1] <= 10^9

**Attempt 1: 2023-12-04**

**Solution 1: Interval intersection check (10 min)**

class Solution {

public List<List<Integer>> removeInterval(int[][] intervals, int[] toBeRemoved) {

int removeStart = toBeRemoved[0];

int removeEnd = toBeRemoved[1];

List<List<Integer>> result = new ArrayList<>();

for(int[] interval : intervals) {

int start = interval[0];

int end = interval[1];

if(start >= removeEnd || end <= removeStart) {

result.add(Arrays.asList(start, end));

} else {

if(start < removeStart) {

result.add(Arrays.asList(start, removeStart));

}

if(end > removeEnd) {

result.add(Arrays.asList(removeEnd, end));

}

}

}

return result;

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://www.cnblogs.com/cnoodle/p/13437387.html>

这也是属于扫描线一类的题目。思路也比较直接，遍历 input 数组，跟需要去掉的区间 **toBeRemoved**没有重叠的子区间就直接加入结果集。如果跟 toBeRemoved 有重叠的部分，则比较两者的左边界和右边界来得出需要去掉的区间是什么。

class Solution {

public List<List<Integer>> removeInterval(int[][] intervals, int[] toBeRemoved) {

List<List<Integer>> res = new ArrayList<>();

for (int[] i : intervals) {

// no overlap

if (i[1] <= toBeRemoved[0] || i[0] >= toBeRemoved[1]) {

res.add(Arrays.asList(i[0], i[1]));

}

// i[1] > toBeRemoved[0] && i[0] < toBeRemoved[1]

else {

// left end no overlap

if (i[0] < toBeRemoved[0]) {

res.add(Arrays.asList(i[0], toBeRemoved[0]));

}

// right end no overlap

if (i[1] > toBeRemoved[1]) {

res.add(Arrays.asList(toBeRemoved[1], i[1]));

}

}

}

return res;

}

}

**Refer to**

<https://algo.monster/liteproblems/1272>

## Problem Description

In this problem, we're dealing with a mathematical representation of sets using intervals of real numbers. Each interval is represented as [a, b), which means it includes all real numbers x such that a <= x < b.

We are provided with two things:

* A **sorted** list of disjoint intervals, intervals, which together make up a set. The intervals are disjoint, meaning they do not overlap, and they are sorted in ascending order based on their starting points.
* Another interval, toBeRemoved, which we need to remove from the set represented by intervals.

Our objective is to return a new set of real numbers obtained by removing toBeRemoved from intervals. This set also needs to be represented as a sorted list of disjoint intervals. We need to consider that part of an interval might be removed, all of it might be removed, or it might not be affected at all, depending on whether it overlaps with toBeRemoved.

## Intuition

The key to solving this problem is to examine each interval in intervals and figure out its relation with toBeRemoved. There are three possibilities:

* The interval is completely outside the range of toBeRemoved and therefore remains unaffected.
* The interval is partially or completely inside the range of toBeRemoved and needs to be trimmed or removed.
* The interval straddles the edges of toBeRemoved and might need to be split into two intervals.

Given that intervals is sorted, we can iterate over each interval and handle the cases as follows:

* If the current interval ends before toBeRemoved starts or starts after toBeRemoved ends, it's disjoint and can be added to the result as is.
* If there is overlap, we may need to trim the current interval. If the start of the current interval is before toBeRemoved, we can take the portion from the interval's start up to the start of toBeRemoved. Similarly, if the interval ends after toBeRemoved, we can take the portion from the end of toBeRemoved to the interval's end.
* We need to handle the edge cases where toBeRemoved completely covers an interval, in which case we add nothing to the result for that interval.

By iterating through each interval once, and considering these cases, we can construct our output set of intervals with toBeRemoved taken out.

## Solution Approach

The provided solution employs a straightforward approach to tackle the problem by iterating through each interval in the given sorted list intervals and comparing it with the toBeRemoved interval. Here's a step by step process used in the implementation:

* The solution starts by initializing an empty list ans, which will eventually contain the resulting set of intervals after the removal process.
* It then enters a loop over each interval [a, b] in the intervals list.
* For each interval, it checks whether there is an intersection with the toBeRemoved interval, [x, y]. It does this by verifying two conditions:

1. If a >= y, then the interval [a, b] is completely after toBeRemoved and thus is unaffected.
2. If b <= x, then the interval [a, b] is completely before toBeRemoved and also remains unaffected.

* When either of the above conditions is true, the current interval can be added directly to the ans list without modification since it doesn't intersect with toBeRemoved.
* If the interval does intersect with toBeRemoved, the solution needs to handle slicing the interval into potentially two parts:

1. If the start of the interval a is before x (the start of toBeRemoved), then the segment [a, x) of the original interval is unaffected by the removal and is added to ans.
2. Similarly, if the end of the interval b is after y (the end of toBeRemoved), then the segment [y, b) remains after the removal and is also added to ans.

* The loops continue for all intervals in intervals, applying the above logic.
* After processing all intervals, the solution returns the ans list, which now contains the modified set of intervals, representing the original set with the toBeRemoved interval excluded.

The algorithm makes use of simple conditional checks and relies on the sorted nature of the input intervals for its correctness and efficiency. The overall time complexity is O(n), where n is the number of intervals in intervals, since it processes each interval exactly once.

### Example Walkthrough

Let's consider the following small example to illustrate the solution approach. Assume we have the following intervals list and toBeRemoved interval:

* intervals = [[1, 4), [6, 8), [10, 13)]
* toBeRemoved = [7, 12)

Using the steps outlined in the solution approach:

#### Step 1: Initialize Result List

ans = [] (empty to begin with)

#### Step 2: Loop Over Each Interval in intervals

Current interval [1, 4).

#### Step 3: Check for Intersection with toBeRemoved

The interval [1, 4) does not intersect with [7, 12), as 4 < 7.

Since the interval is completely before toBeRemoved, add it to ans: ans = [[1, 4)].

Next, we take the interval [6, 8).

#### Step 4: Check for Intersection with toBeRemoved

The interval [6, 8) does intersect with [7, 12) since the interval starts before and ends in the range of toBeRemoved.

#### Step 5: Handle Slicing the Interval

The start of the interval 6 is before the start of toBeRemoved 7.

Add the segment [6, 7) to ans: ans = [[1, 4), [6, 7)].

Next, we take the interval [10, 13).

#### Step 6: Check for Intersection with toBeRemoved

The interval [10, 13) does intersect with [7, 12), because the interval starts inside and ends after the range of toBeRemoved.

#### Step 7: Handle Slicing the Interval

Since the end of the interval 13 is after the end of toBeRemoved 12, we add the segment [12, 13) to ans: ans = [[1, 4), [6, 7), [12, 13)].

#### Step 8: Continue the Loop

No more intervals to process.

#### Step 9: Return the ans List

The final result is ans = [[1, 4), [6, 7), [12, 13)].

The solution approach has efficiently handled the example intervals list by considering the toBeRemoved interval and has produced a result that correctly represents the set after removal.

class Solution {

// Function to remove a specific interval from a list of intervals

public List<List<Integer>> removeInterval(int[][] intervals, int[] toBeRemoved) {

// x and y represents the start and end of the interval to be removed

int removeStart = toBeRemoved[0];

int removeEnd = toBeRemoved[1];

// Preparing a list to store the resulting intervals after removal

List<List<Integer>> updatedIntervals = new ArrayList<>();

// Iterate through each interval in the input intervals array

for (int[] interval : intervals) {

// a and b represents the start and end of the current interval

int start = interval[0];

int end = interval[1];

// Check if the current interval is completely before or after the interval to be removed

if (start >= removeEnd || end <= removeStart) {

// Add to the result as there is no overlap

updatedIntervals.add(Arrays.asList(start, end));

} else {

// If there's an overlap, we may need to add the non-overlapping parts of the interval

if (start < removeStart) {

// Add the part of the interval before the interval to be removed

updatedIntervals.add(Arrays.asList(start, removeStart));

}

if (end > removeEnd) {

// Add the part of the interval after the interval to be removed

updatedIntervals.add(Arrays.asList(removeEnd, end));

}

}

}

// Return the list of updated intervals

return updatedIntervals;

}

}

## Time and Space Complexity

The code snippet provided is for a function that removes an interval from a list of existing intervals and returns the resulting list of disjoint intervals after the removal. The computational complexity analysis for time and space complexity is as follows:

**Time complexity:**

The primary operation in this function occurs within a single loop that iterates over all the original intervals in the list intervals. Within each iteration of the loop, the function performs constant-time checks and operations to possibly add up to two intervals to the ans list. Since there are no nested loops and the operations inside the loop are of constant time complexity, the overall time complexity of the function is directly proportional to the number of intervals n in the input list. Therefore, the time complexity is O(n).

**Space complexity:**

For space complexity, the function creates a new list ans to store the resulting intervals after the potential removal and modification of the existing intervals. In the worst-case scenario, where no interval is completely removed and every interval needs to be split into two parts (one occurring before x and one after y of the toBeRemoved interval), the resulting list could potentially hold up to 2n intervals - doubling the input size. However, notice that this is a linear relationship with respect to the number of input intervals n. Therefore, the space complexity of the function is O(n) as well.

In summary, both the time complexity and space complexity of the given code are O(n), where n is the number of intervals in the input list intervals.