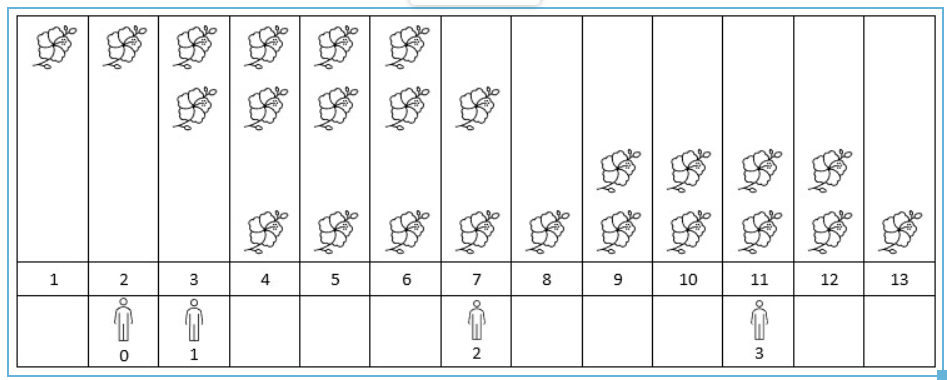
<https://leetcode.com/problems/number-of-flowers-in-full-bloom/description/>

You are given a **0-indexed** 2D integer array flowers, where flowers[i] = [starti, endi] means the ith flower will be in **full bloom** from starti to endi (**inclusive**). You are also given a **0-indexed** integer array people of size n, where people[i] is the time that the ith person will arrive to see the flowers.

Return *an integer array* answer *of size* n*, where* answer[i] *is the* ***number*** *of flowers that are in full bloom when the* ith *person arrives.*

**Example 1:**



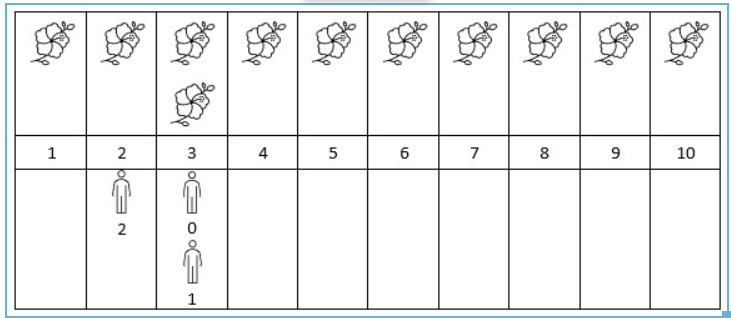
Input: flowers = [[1,6],[3,7],[9,12],[4,13]], people = [2,3,7,11]

Output: [1,2,2,2]

Explanation: The figure above shows the times when the flowers are in full bloom and when the people arrive.

For each person, we return the number of flowers in full bloom during their arrival.

**Example 2:**



Input: flowers = [[1,10],[3,3]], people = [3,3,2]

Output: [2,2,1]

Explanation: The figure above shows the times when the flowers are in full bloom and when the people arrive.

For each person, we return the number of flowers in full bloom during their arrival.

**Constraints:**

* 1 <= flowers.length <= 5 \* 10^4
* flowers[i].length == 2
* 1 <= starti <= endi <= 10^9
* 1 <= people.length <= 5 \* 10^4
* 1 <= people[i] <= 10^9

**Attempt 1: 2023-12-10**

**Solution 1:  Sweep Line + Sorting + TreeMap + Binary Search (60 min)**

**以前没遇到必须在TreeMap delta中添加(0, 0)的问题是因为不涉及binary search，但是本题中由于涉及到binary search，所以左边界lo是0还是1很重要，如果delta中不添加(0, 0)，那么左边界必须从lo = 1开始计算而不是0**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

TreeMap<Integer, Integer> delta = new TreeMap<>();

// Why we have to add (0, 0) onto delta array ?

// What happens if there is a person that arrives before

// any flower blooms?

// This may confuse our binary search since the minimum

// value in positions will be greater than person. We will

// initialize difference with 0: 0 to represent at time 0,

// we don't see any new flowers.

// In another word: Since this problem relate to Binary Search,

// the lower boudary is important, if not adding 0 as lower

// boundary, and only based on given 'flowers' array which start

// index from 1 only, the lower boundary in Binary Search will

// also be 1 not 0

// ========================================================

// If no 'delta.put(0, 0)', error out:

// IndexOutOfBoundsException: Index -1 out of bounds for length 4

// Test out by:

// flowers = [[19,37],[19,38],[19,35]]

// people = [6,7,21,1,13,37,5,37,46,43]

// Failed when binary search on 6 which exceed flowers min

// range before 19, 'lo' will be 0 - 1 = -1, when attempt

// on 'list.get(lo)' will error out "IndexOutOfBoundsException"

delta.put(0, 0);

for(int[] flower : flowers) {

delta.put(flower[0], delta.getOrDefault(flower[0], 0) + 1);

delta.put(flower[1] + 1, delta.getOrDefault(flower[1] + 1, 0) - 1);

}

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

int count = 0;

for(int val : delta.values()) {

count += val;

presum.add(count);

}

List<Integer> keys = new ArrayList<>(delta.keySet());

int[] result = new int[people.length];

for(int i = 0; i < result.length; i++) {

int index = binarySearch(keys, people[i]);

result[i] = presum.get(index);

}

return result;

}

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

int lo = 0;

int hi = list.size() - 1;

while(lo <= hi) {

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

if(list.get(mid) == val) {

return mid;

} else if(list.get(mid) > val) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

// Refer L704.Binary Search

// (1) Why need 'list.get(lo) != val' check and what should return ?

// If not able to find target 'val' index till go through the list,

// return the index able to insert 'val' into list as 'lo - 1'.

// And why always 'lo - 1' as insert index ? Because when it happen

// the loop ending condition is 'lo > hi', which means when loop

// terminated, the 'lo' index is one position right to 'hi' index,

// and still no matched 'val', the 'val' should insert at 'lo - 1'

// index, which is left to 'lo' index but right to 'hi' index

// e.g

// loop terminated:

// hi lo

// |----| -> no 'val' found

// adding 'insert = lo - 1' index left to 'lo' but right to 'hi'

// hi insert lo

// |----|----|

// ================================================================

// (2) Why need 'lo >= list.size()' condition ?

// If no 'lo >= list.size()' condition, error out:

// IndexOutOfBoundsException: Index 5 out of bounds for length 5

// Test out by:

// flowers = [[19,37],[19,38],[19,35]]

// people = [6,7,21,1,13,37,5,37,46,43]

// Failed when binary search on 46, 43 which exceed flowers max

// range till 38, 'lo' will be 5 >= list.size() = 5, when attempt

// on 'list.get(lo)' will error out "IndexOutOfBoundsException"

if(lo >= list.size() || list.get(lo) != val) {

return lo - 1;

}

return lo;

}

}

Time complexity: O((n+m)⋅log⁡n)

Our first loop sets difference, which costs O(n⋅log⁡n)

Next, we calculate the prefix sum, which will cost either O(n) or O(n⋅log⁡n) depending on your language's implementation. This is because difference will have a size between n and 2n.

Finally, we have a loop over people. We perform a binary search that costs O(log⁡n) at each iteration. Thus, we spend m⋅log⁡n here. This gives us a final time complexity of O((n+m)⋅log⁡n)

Space complexity: O(n), difference as a size of O(n). prefix and positions have the same size as difference.

**Refer to**

<https://leetcode.com/problems/number-of-flowers-in-full-bloom/editorial/>

#### **Approach 2: Difference Array + Binary Search**

**Intuition**

There is a technique called difference array that can be used to solve many "range" based problems. The technique involves creating an array difference and iterating over all ranges [start, end]. We perform difference[start]++ and difference[end + 1]-- for each range.

The idea is that each index of difference represents the **change** in the number of flowers we can see when we cross this index (not the actual number of flowers on this index), with each index representing a unit of time. Thus, we could take a prefix sum of this difference array to find how many flowers can be seen at any given time with prefix[time]

Unfortunately, if we look at the constraints, we find that values of start, end, people can be up to 10^9. It would not be feasible to create an array with such a large size. Thus, we need to use a map structure instead. Like in the previous approach, we still want to process everything chronologically. We will use the following data structures:

* In Java, we will use TreeMap.
* In C++, we will use std::map.
* In Python, we will use sortedcontainers.SortedDict.

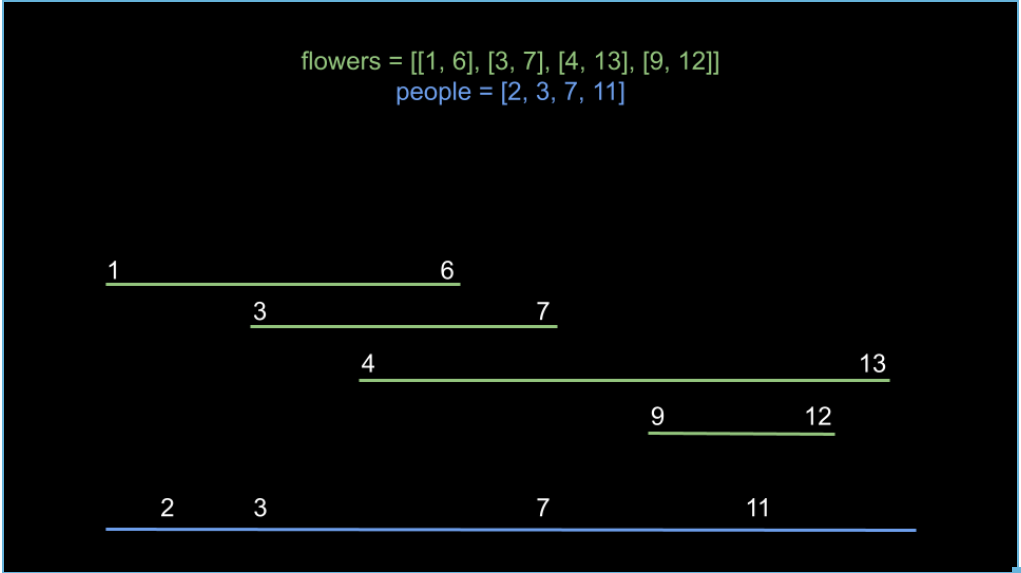
Note that if you were not allowed to use these structures in an interview, you could still implement this approach using a normal hash map. You would just need to sort the elements in the hash map by key values after you populated it.

Once we have this data structure difference, we will follow the process described above. We iterate over each flower = [start, end] and increment difference[start] while decrementing difference[end + 1]. The idea is that when we reach start, the number of flowers we see increases by one. When we reach end + 1, the number of flowers we see decreases by one.

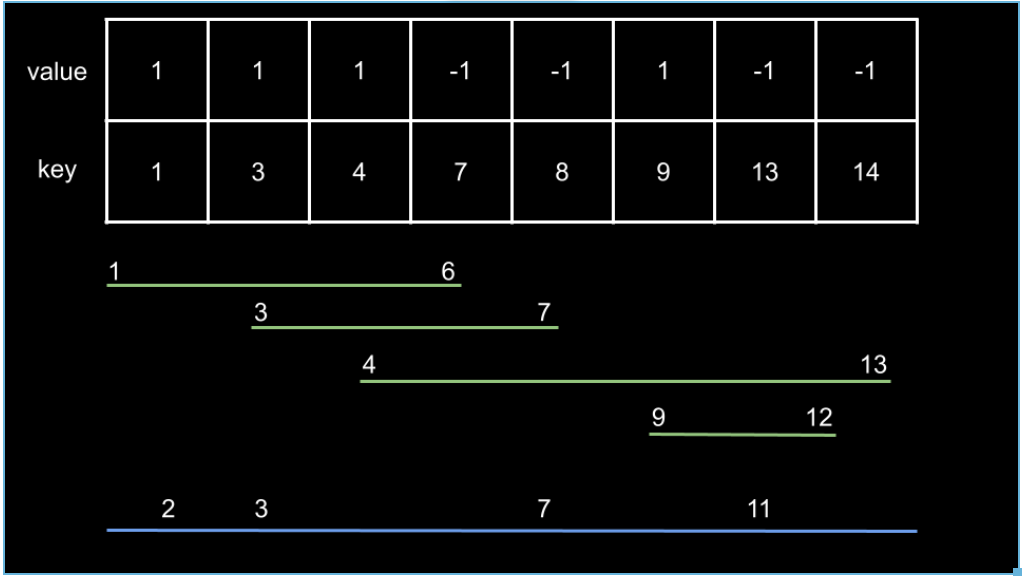
We then create a prefix sum of the values in difference. We also need to know what time each value is associated with, so we will create an array positions to go along with our prefix array. Here, prefix[i] is the number of flowers available at time positions[i].

Finally, we can iterate over people and find the answer for each person. How do we do this? We can perform a binary search over positions to find the index i where person fits. prefix[i] is the answer for this person.

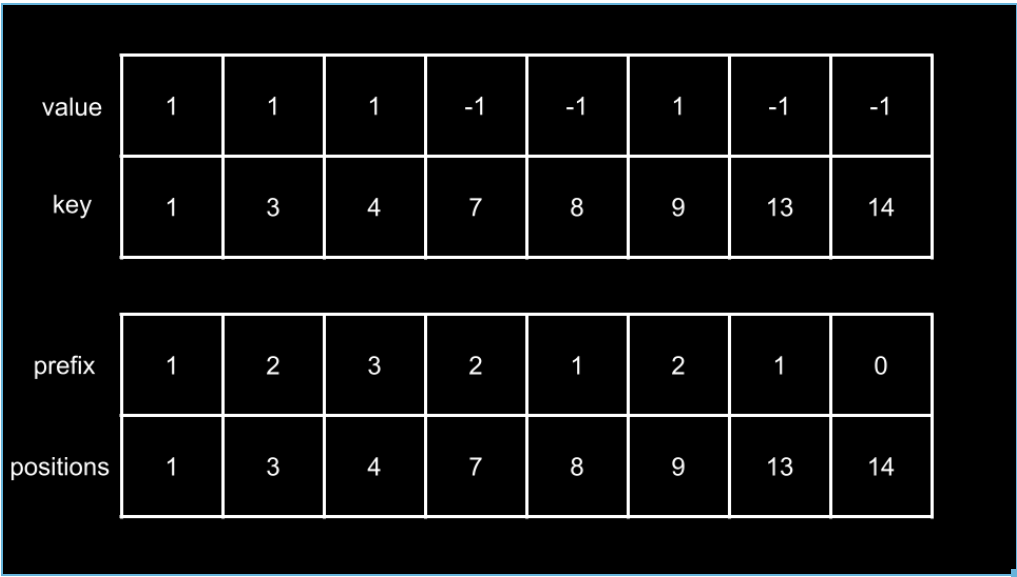
Let's summarize the algorithm with an example:



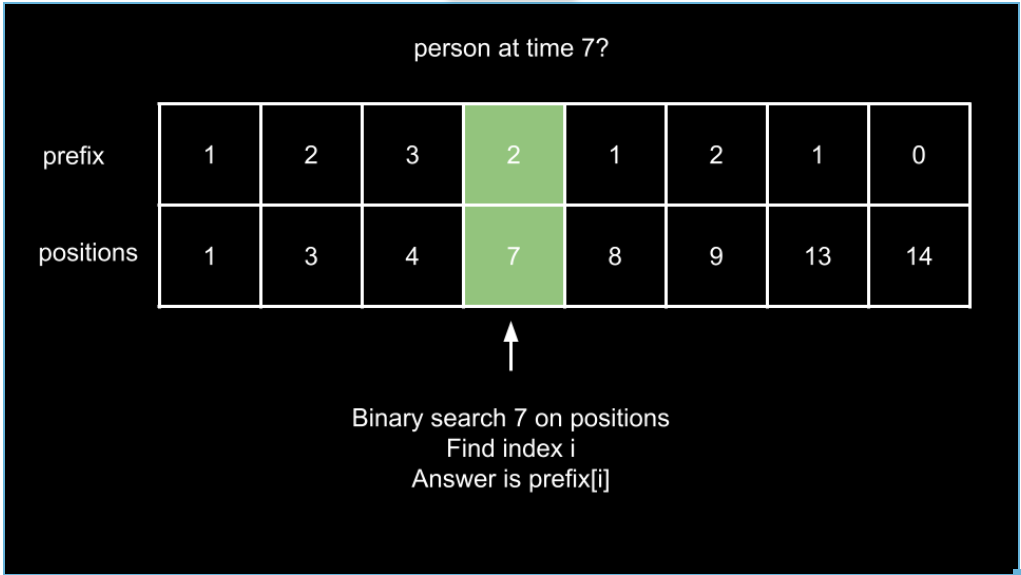
Our first step is to populate difference. Each key, value pair in difference represents "at time key, we see a change in value new flowers". For example, the key value pair of 6: -2 means that at time 6, we see two less flowers.



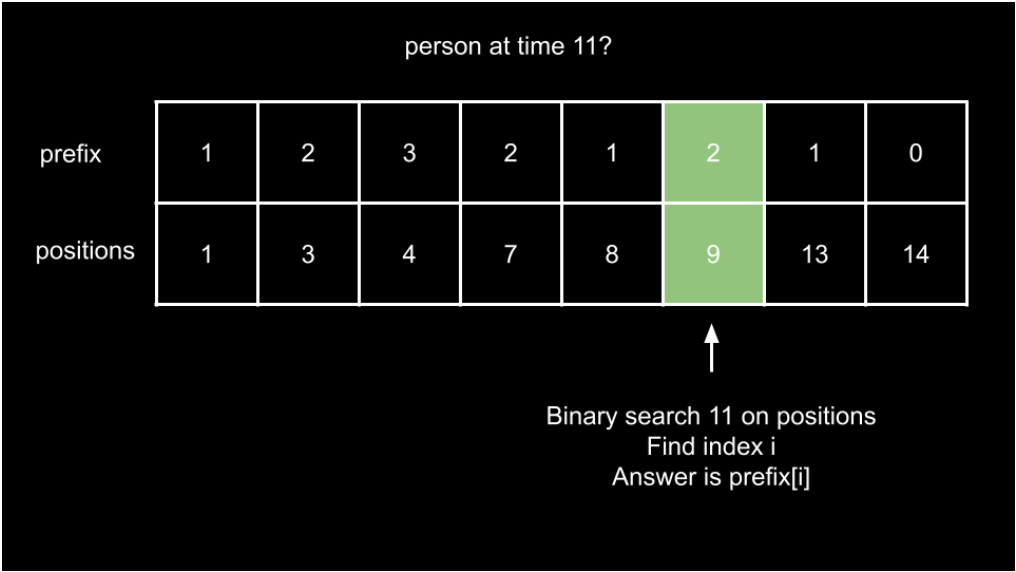
Next, we create a prefix sum on the values of difference, as well as an array positions to associate each prefix value with a position in time. Notice that positions is just the keys of difference.



With these arrays, we can now use binary search to identify how many flowers a given person will see. For example, consider person at time 7:



What about person at time 11?



There are a few more things to consider before we start implementation.

* What happens if there is a person that arrives before any flower blooms? This may confuse our binary search since the minimum value in positions will be greater than person. We will initialize difference with 0: 0 to represent at time 0, we don't see any new flowers.
* Regarding the binary search; how should it be configured? Referencing the above example images, inserting 11 into the given positions array will put it at index 6. However, we need index 5. Thus, we need the insertion index minus one. What if the value exists in positions, as is the case with person = 7? To offset the minus one, we will binary search for the rightmost insertion index (bisect\_right in Python, upper\_bound in C++).

**Algorithm**

1. Initialize a sorted-map data structure difference with 0: 0.
2. Iterate over each flower = [start, end] in flowers:

* Increment difference[start].
* Decrement difference[end + 1].

1. Initialize two arrays, positions and prefix. Iterate over the keys of difference:

* positions contains all the keys in the order they are traversed.
* prefix contains the prefix sum of the corresponding values.

1. Initialize the answer array ans. Iterate over each person in people:

* Perform a right-insertion index binary search on positions with person.
* Calculate i as the result of this binary search minus one.
* Add prefix[i] to ans.

1. Return ans.

**Implementation**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

TreeMap<Integer, Integer> difference = new TreeMap<>();

difference.put(0, 0);

for (int[] flower : flowers) {

int start = flower[0];

int end = flower[1] + 1;

difference.put(start, difference.getOrDefault(start, 0) + 1);

difference.put(end, difference.getOrDefault(end, 0) - 1);

}

List<Integer> positions = new ArrayList();

List<Integer> prefix = new ArrayList();

int curr = 0;

for (int key : difference.keySet()) {

positions.add(key);

curr += difference.get(key);

prefix.add(curr);

}

int[] ans = new int[people.length];

for (int j = 0; j < people.length; j++) {

int i = binarySearch(positions, people[j]) - 1;

ans[j] = prefix.get(i);

}

return ans;

}

public int binarySearch(List<Integer> arr, int target) {

int left = 0;

int right = arr.size();

while (left < right) {

int mid = (left + right) / 2;

if (target < arr.get(mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

}

**Complexity Analysis**

Given n as the length of flowers and m as the length of people,

* Time complexity: O((n+m)⋅log⁡n)
* Our first loop sets difference, which costs O(n⋅log⁡n)
* Next, we calculate the prefix sum, which will cost either O(n) or O(n⋅log⁡n) depending on your language's implementation. This is because difference will have a size between n and 2n.
* Finally, we have a loop over people. We perform a binary search that costs O(log⁡n) at each iteration. Thus, we spend m⋅log⁡n here.
* This gives us a final time complexity of O((n+m)⋅log⁡n)
* Space complexity: O(n)
* difference as a size of O(n). prefix and positions have the same size as difference.

**Solution 2: Promote Binary Search (60 min)**

**Wrong Solution (31/52)**

**Test out by**

Input

flowers = [[36,39],[29,49],[32,35],[14,43],[42,49],[48,48],[32,46],[6,41],[14,19]]

people = [14,4]

Output = [2,0]

Expected = [3,0]

**Regarding the binary searches: when binary searching on starts, we want to search for the rightmost insertion index. This is because if a person arrives at the same time as a flower starts blooming, we want to include this flower. 这就是使用Find Upper Boundary的原因，比如在arr = [6,14,14,29,32,32, ...]中寻找val = 14，要是直接在arr[mid] == val (arr[1] == 14) 的时候返回，那么mid = 1就满足了(此时是[14, 43]或[14, 19]其中一朵花)，然而实际上mid = 2的时候也满足(此时是[14, 43]或[14, 19]中另一朵花)，而我们为了包含所有盛开的花，直接在mid = 1的时候返回就会失去在mid = 2的时候人到了另一朵花开的情况**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

int n = flowers.length;

int[] start = new int[n];

int[] end = new int[n];

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

start[i] = flowers[i][0];

end[i] = flowers[i][1] + 1;

}

Arrays.sort(start);

Arrays.sort(end);

int[] result = new int[people.length];

for(int i = 0; i < people.length; i++) {

int a = binarySearch(start, people[i]);

int b = binarySearch(end, people[i]);

result[i] = a - b;

}

return result;

}

// Find upper boundary

private int binarySearch(int[] arr, int val) {

int lo = 0;

int hi = arr.length - 1;

while(lo <= hi) {

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

if(arr[mid] == val) {

return mid;

} else if(arr[mid] > val) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

if(lo >= arr.length || arr[lo] != val) {

return lo - 1;

}

return lo - 1;

}

}

**Correct Solution**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

int n = flowers.length;

int[] start = new int[n];

int[] end = new int[n];

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

start[i] = flowers[i][0];

end[i] = flowers[i][1] + 1;

}

Arrays.sort(start);

Arrays.sort(end);

int[] result = new int[people.length];

for(int i = 0; i < people.length; i++) {

int a = binarySearch(start, people[i]);

int b = binarySearch(end, people[i]);

result[i] = a - b;

}

return result;

}

// Find upper boundary

private int binarySearch(int[] arr, int val) {

int lo = 0;

int hi = arr.length - 1;

while(lo <= hi) {

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

//if(arr[mid] == val) {

// return mid;

//} else if(arr[mid] > val) {

// Change to standard find upper boundary style

// 要找 x ≤ target 的上界，首先套用上文的模板代码，

// 实现找 x > target 的下界的函数(注意不是x >= target)

if(arr[mid] > val) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

// Since same output as 'lo - 1' for exception

// handling and find upper boundary style, so

// remove duplicate statements

//if(lo >= arr.length || arr[lo] != val) {

// return lo - 1;

//}

return lo - 1;

}

}

Given n as the length of flowers and m as the length of people,

Time complexity: O((n+m)⋅log⁡n)

We first create two arrays of length n, startsand ends, then sort them. This costs O(n⋅log⁡n).

Next, we iterate over people and perform two binary searches at each iteration. This costs O(m⋅log⁡n).

Thus, our time complexity is O((n+m)⋅log⁡n).

Space complexity: O(n)

starts and ends both have a size of n.

**Refer to**

<https://leetcode.com/problems/number-of-flowers-in-full-bloom/editorial/>

#### **Approach 3: Simpler Binary Search**

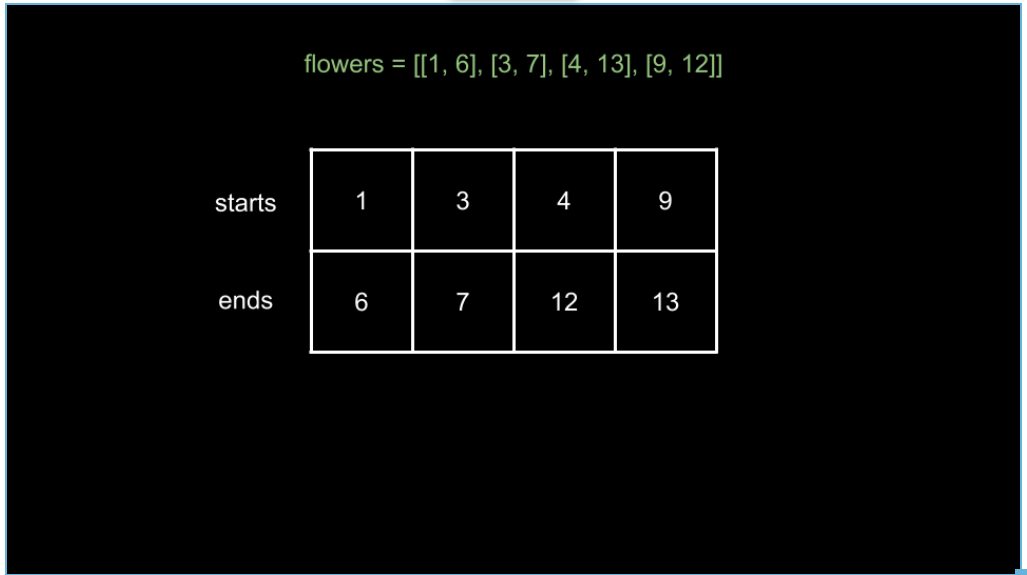
**Intuition**

In the previous approach, we used the concept of a difference array/line sweep to calculate how many flowers are seen at a given time. For each flower = [start, end], we indicated that at time start, we see one more flower, and at time end + 1, we see one less flower. We identified when a flower started blooming and when it finished blooming.

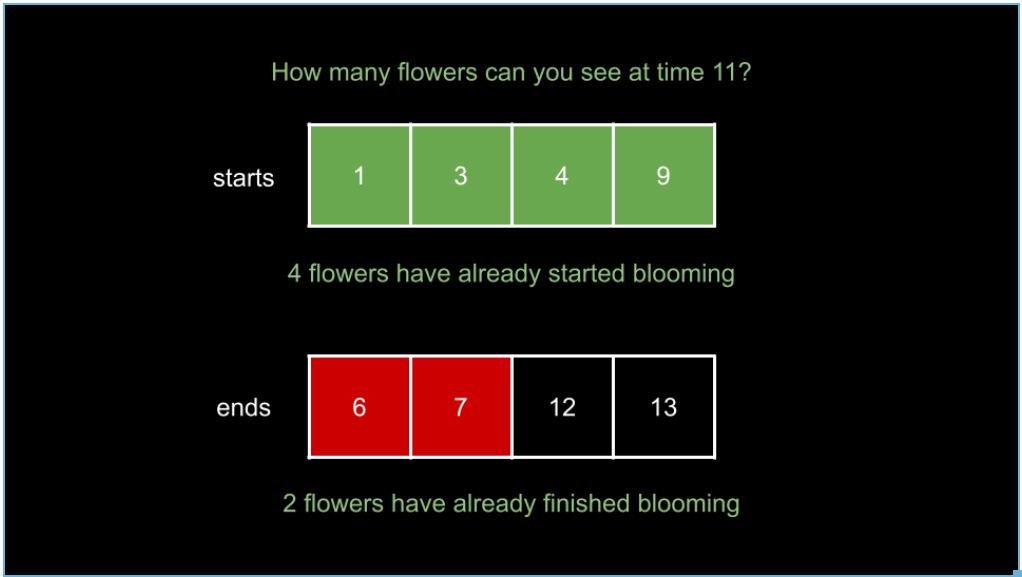
The idea behind this strategy is that at any given time, **the number of flowers we see is the number of flowers that have already started blooming minus the amount of flowers have finished blooming.**

Is there a simpler way to identify at a given time, how many flowers have started blooming, and how many flowers have finished blooming? In the first two approaches, we always associate the start and end of the same flower together for processing, which is more intuitive but can be more complex to handle. What if we separately consider these two sets of times?

We can simply collect all start points in one array starts, sort it, and then perform a binary search. We can do the exact same thing with another array ends for all end points. Take a look at the following example:



Here, we have collected all start and end times and then sorted them. How many flowers can somebody at time 11 see?



As you can see, 4 flowers have started blooming and 2 flowers have finished blooming. Thus, 4 - 2 = 2 flowers can be seen at time 11. Because starts and ends is sorted, we can use binary search to quickly identify how many flowers have started and finished blooming for any given time.

Regarding the binary searches: when binary searching on starts, we want to search for the rightmost insertion index. This is because if a person arrives at the same time as a flower starts blooming, we want to include this flower.

Note that a flower = [start, end] stops blooming at end + 1, not end. There are two ways we can handle this. We can either binary search on end for the leftmost insertion index (since we want to include all flowers with end equal to the current time), or we can assemble ends using end + 1 for each flower. We will implement the algorithm using the second option in this article.

**Algorithm**

1. Create two arrays starts and ends.
2. Iterate over each flower = [start, end] in flowers:

* Add start to starts.
* Add end + 1 to ends.

1. Sort both starts and ends.
2. Initialize the answer array ans and iterate over each person in people:

* Perform a binary search on starts for the rightmost insertion index of person to find i.
* Perform a binary search on ends for the rightmost insertion index of person to find j.
* Add i - j to ans.

1. Return ans.

**Implementation**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

List<Integer> starts = new ArrayList();

List<Integer> ends = new ArrayList();

for (int[] flower: flowers) {

starts.add(flower[0]);

ends.add(flower[1] + 1);

}

Collections.sort(starts);

Collections.sort(ends);

int[] ans = new int[people.length];

for (int index = 0; index < people.length; index++) {

int person = people[index];

int i = binarySearch(starts, person);

int j = binarySearch(ends, person);

ans[index] = i - j;

}

return ans;

}

public int binarySearch(List<Integer> arr, int target) {

int left = 0;

int right = arr.size();

while (left < right) {

int mid = (left + right) / 2;

if (target < arr.get(mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

}

**Complexity Analysis**

Given n as the length of flowers and m as the length of people,

* Time complexity: O((n+m)⋅log⁡n)
* We first create two arrays of length n, startsand ends, then sort them. This costs O(n⋅log⁡n).
* Next, we iterate over people and perform two binary searches at each iteration. This costs O(m⋅log⁡n).
* Thus, our time complexity is O((n+m)⋅log⁡n).
* Space complexity: O(n)
* starts and ends both have a size of n.

**Another style but same purpose:**

**Refer to**

<https://algo.monster/liteproblems/2251>

## Problem Description

In this problem, we are dealing with a scenario related to flowers blooming. The input includes two arrays:

* A 2D array flowers, where each sub-array contains two elements indicating the start and end of the full bloom period for a particular flower. The flower blooms inclusively from start\_i to end\_i.
* An array persons, where each element represents the time a person arrives to see the flowers.

The goal is to determine how many flowers are in full bloom for each person when they arrive. The output should be an array answer, where answer[i] corresponds to the number of flowers in full bloom at the time the ith person arrives.

## Intuition

To solve this problem, we can use a two-step strategy involving sorting and binary search:

* **Sorting**: We separate the start and end times of the bloom periods into two lists and sort them. The sorted start times help us determine how many flowers have started blooming at a given point, and the sorted end times indicate how many flowers have finished blooming.
* **Binary Search**: When a person arrives, we want to count the flowers that have begun blooming but haven't finished. We use the binary search algorithm to find:

1. The index of the *first* end time that is *strictly greater* than the arrival time of the person, which indicates how many flowers have finished blooming. We get this number using bisect\_left on the sorted end times.
2. The index of the *first* start time that is *greater than or equal* to the arrival time, which tells us how many flowers have started to bloom. We use bisect\_right for this on the sorted start times.

By subtracting the number of flowers that have finished blooming from those that have started, we get the count of flowers in full bloom when a person arrives. We repeat this process for each person and compile the results into the final answer array.

## Solution Approach

The solution approach uses a combination of sorting and binary search to efficiently determine how many flowers are in full bloom for each person's arrival time. Here's the implementation explained step by step:

* **Sort Starting and Ending Times**: First, we extract all the start times and end times from the flowers array into separate lists and sort them:

start = sorted(a for a, \_ in flowers)

end = sorted(b for \_, b in flowers)

Sorting these lists allows us to use binary search later on. The start list will be used to determine how many flowers have started blooming by a certain time, and the end list will help determine how many flowers have ended their bloom.

* **Binary Search for Bloom Count**: The next step is to iterate over each person's arrival time p in the persons list and find out the count of flowers in bloom at that particular time. For each p:

bisect\_right(start, p) - bisect\_left(end, p)

bisect\_right(start, p) finds the index in the sorted start list where p would be inserted to maintain the order. This index represents the count of all flowers that have started blooming up to time p (including p).

bisect\_left(end, p) finds the index in the sorted end list where p could be inserted to maintain the order. This index signifies the count of flowers that have not finished blooming by time p.

By subtracting the numbers obtained from bisect\_left on the end list from bisect\_right on the start list, we obtain the total number of flowers in bloom at the arrival time of p.

* **Compile Results**: The above operation is repeated for each person's arrival time, and the results are compiled into the answer list. This list comprehends the count of flowers in bloom for each person, as per their arrival times:

return [bisect\_right(start, p) - bisect\_left(end, p) for p in persons]

In the end, the answer list is returned, which provides the solution, i.e., the number of flowers in full bloom at the time of each person's arrival.

The algorithms and data structures used here, like sorting and binary search (bisect module in Python), enable us to solve the problem in a time-efficient manner, taking advantage of the ordered datasets for quick lookups.

### Example Walkthrough

Imagine we have an array of flowers where the blooms are represented as flowers = [[1,3], [2,5], [4,7]] and an array of persons with arrival times as persons = [1, 3, 5]. We want to find out how many flowers are in full bloom each person sees when they arrive.

First, we need to process the flowers' bloom times. We sort the start times [1, 2, 4] and the end times [3, 5, 7] of the blooming periods.

Now let's walk through the steps to get the number of flowers in bloom for each person:

Person at time 1:

Using bisect\_right for the sorted start times: bisect\_right([1, 2, 4], 1) gives us index 1, indicating one flower has started blooming.

Using bisect\_left for the end times: bisect\_left([3, 5, 7], 1) gives us index 0, indicating no flowers have finished blooming.

The difference 1 (started) - 0 (ended) tells us that exactly one flower is in full bloom.

Person at time 3:

bisect\_right([1, 2, 4], 3) results in index 2, as two flowers have bloomed by time 3.

bisect\_left([3, 5, 7], 3) gives us index 1, as one flower has stopped blooming.

The difference 2 (started) - 1 (ended) is 1, so one flower is blooming for this person.

Person at time 5:

bisect\_right([1, 2, 4], 5) gives an index of 3 - all three flowers have started blooming by time 5.

bisect\_left([3, 5, 7], 5) yields index 2, as two flowers have finished blooming strictly before time 5.

The difference 3 (started) - 2 (ended) is 1, indicating that one flower is in bloom when this person arrives.

Thus, for the persons arriving at times 1, 3, and 5, the function will return [1, 1, 1] as the number of flowers in full bloom at each of their arrival times.

import java.util.Arrays;

public class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

int flowerCount = flowers.length; // Number of flowers

int[] bloomStart = new int[flowerCount];

int[] bloomEnd = new int[flowerCount];

// Extract the start and end bloom times for each flower into separate arrays

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

bloomStart[i] = flowers[i][0];

bloomEnd[i] = flowers[i][1];

}

// Sort the start and end bloom times arrays

Arrays.sort(bloomStart);

Arrays.sort(bloomEnd);

int peopleCount = people.length; // Number of people

int[] answer = new int[peopleCount]; // Array to store the answers

// For each person, calculate the number of flowers in full bloom

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

// Number of flowers that have started blooming minus

// the number of flowers that have already ended blooming

answer[i] = findInsertionPoint(bloomStart, people[i] + 1) - findInsertionPoint(bloomEnd, people[i]);

}

return answer; // Return the array containing the number of flowers in full bloom for each person

}

private int findInsertionPoint(int[] times, int value) {

int left = 0; // Start of the search range

int right = times.length; // End of the search range

// Binary search to find the insertion point of 'value'

while (left < right) {

int mid = (left + right) / 2; // Midpoint of the current search range

if (times[mid] >= value) {

right = mid; // Adjust the search range to the left half

} else {

left = mid + 1; // Adjust the search range to the right half

}

}

return left; // The insertion point is where we would add 'value' to keep the array sorted

}

}

**Solution 3:  Priority Queue + Sorting (180 min)**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

// Create a sorted version of people called sortedPeople.

int[] sorted\_people = Arrays.copyOf(people, people.length);

Arrays.sort(sorted\_people);

// Sort flowers based on 'start'

Arrays.sort(flowers, (a, b) -> a[0] - b[0]);

// Initialize a hash map that maps a person to the number

// of flowers they see

Map<Integer, Integer> map = new HashMap<>();

// A minPQ based on 'end' of flowers

PriorityQueue<Integer> minPQ = new PriorityQueue<>();

// Note: 'person' means the element in sorted people array,

// where people[i] is the time that the ith person will arrive

// to see the flowers

// To find the flowers with start less than a given 'person',

// we can use a pointer i that starts at 0. We will move i

// along the flowers array and never decrement or reset it.

// This allows us to pick up where we left off for each

// successive 'person'.

int i = 0;

for(int p : sorted\_people) {

// While flowers[i][0] < 'person' (the flower at i already

// started blooming), means flower start blooming before

// 'person' coming, push flowers[i][1] (when the flower

// finishes blooming) to heap and increment i.

while(i < flowers.length && flowers[i][0] <= p) {

minPQ.offer(flowers[i][1]);

i++;

}

// While the top of heap (minimum element) is less than

// 'person', means the flower finishes blooming before

// 'person' coming, pop from heap

while(!minPQ.isEmpty() && minPQ.peek() < p) {

minPQ.poll();

}

// The answer for 'person' is simply the size of the heap.

map.put(p, minPQ.size());

}

int[] result = new int[people.length];

for(int j = 0; j < people.length; j++) {

result[j] = map.get(people[j]);

}

return result;

}

}

Time complexity: O(n⋅log⁡n+m⋅(log⁡n+log⁡m))

We start by sorting both flowers and people. This costs O(n⋅log⁡n) and O(m⋅log⁡m) respectively. Next, we perform O(m) iterations. At each iteration, we perform some heap operations. The cost of these operations is dependent on the size of the heap. Our heap cannot exceed a size of n, so these operations cost O(log⁡n).

There are some other linear time operations that don't affect our time complexity. In total, our time complexity is O(n⋅log⁡n+m⋅(log⁡n+log⁡m)).

Space complexity: O(n+m)

We create an array sortedPeople of length m. dic also grows to a length of m, and heap can grow to a size of O(n).

**Refer to**

<https://leetcode.com/problems/number-of-flowers-in-full-bloom/editorial/>

#### **Approach 1: Heap/Priority Queue**

**Intuition**

For each person in people, we need to find how many flower ranges [start, end] contain person. An intuitive first step is to sort both input arrays so that we can process both flowers and people in chronological order.

For the first person (in terms of arrival time), we can find all the flowers that have start less than person - these are the flowers that have started blooming before person arrived, and thus person might have a chance of seeing them. Of those flowers, we remove the ones that have end less than person as well, as these are the flowers that have finished blooming, and person missed them. The number of remaining flowers is the answer for the first person. Note that because we sorted people, the flowers we remove here are guaranteed never to be seen again and therefore will not affect anyone else after person.

Let's move to the second person. Once again, we find all the flowers that have start less than person. But do we need to start from scratch? No! Because we are processing both the flowers and people in order, we can start from where we left off with the previous person. More specifically, because the second person's arrival time is greater than or equal to the previous person's, the flowers that bloom before the previous person must also bloom before the second person, so there's no need for us to handle this portion of flowers again. Therefore, we will add all the flowers that have start less than the second person, starting after the last flower we took.

Similarly, the flowers that the previous person missed are definitely also missed by the second person, so there's no need for us to handle this portion of removed flowers again. Once we have taken all the flowers with start less than person, we can simply remove all the flowers that have end less than person. The number of remaining flowers is the answer for the second person.

We can continue this process for each person. To find the flowers with start less than a given person, we can use a pointer i that starts at 0. We will move i along the flowers array and never decrement or reset it. This allows us to pick up where we left off for each successive person.

How can we remove the flowers that have end less than a given person? This one is trickier because we can only sort flowers by one dimension. To use the pointer technique we just described, we must sort by the start times. Thus, the end times are not necessarily in order. For example, you could have flowers like this:

[2, 9], [3, 6]

In this case, using another pointer like j for the end times would not work since 9 is greater than 6 but comes earlier in the input.

As we are concerned with the flowers that have earlier end times, we can use a heap/priority queue to keep track of which flowers finish blooming. We will maintain a min heap and push end times of flowers onto this heap. Once we have added all flowers with start less than person, we will pop from the heap as long as the top of it is less than person.

After popping from heap, it will hold the end times of all flowers that person can see. Thus, the answer for person is simply the size of the heap.

To summarize, we use a pointer i to iterate along flowers. For a given person, we find all the flowers that started blooming before person arrives. We push the end time of these flowers onto a heap. We can then remove all the flowers that finished blooming by popping from the heap, since a min heap efficiently gives us the minimum (earliest) times.

As we sort both input arrays, flowers that we pop from heap will never be seen again by future people.

A note on implementation: here, we are sorting people, but the problem description asks us for the answer according to the original order. We will use a hash map that maps a person to the number of flowers they see. We will also keep the original order of people by creating a copy of it to sort. Once we have calculated the answer for everyone in the sorted order, we can iterate through the original people and refer to the hash map to build the final answer by restoring their original order.

**Algorithm**

1. Sort flowers. Create a sorted version of people called sortedPeople.
2. Initialize a hash map dic, a min heap, and an integer i = 0.
3. Iterate over sortedPeople. For each person:

* While flowers[i][0] < person (the flower at i already started blooming), push flowers[i][1] (when the flower finishes blooming) to heap and increment i.
* While the top of heap (minimum element) is less than person, pop from heap.
* Set dic[person] to the size of heap.

1. Initialize an array ans. Iterate over people and populate ans using dic.

**Implementation**

class Solution {

public int[] fullBloomFlowers(int[][] flowers, int[] people) {

int[] sortedPeople = Arrays.copyOf(people, people.length);

Arrays.sort(sortedPeople);

Arrays.sort(flowers, (a, b) -> Arrays.compare(a, b));

Map<Integer, Integer> dic = new HashMap();

PriorityQueue<Integer> heap = new PriorityQueue();

int i = 0;

for (int person : sortedPeople) {

while (i < flowers.length && flowers[i][0] <= person) {

heap.add(flowers[i][1]);

i++;

}

while (!heap.isEmpty() && heap.peek() < person) {

heap.remove();

}

dic.put(person, heap.size());

}

int[] ans = new int[people.length];

for (int j = 0; j < people.length; j++) {

ans[j] = dic.get(people[j]);

}

return ans;

}

}

**Complexity Analysis**

Given n as the length of flowers and m as the length of people,

Time complexity: O(n⋅log⁡n+m⋅(log⁡n+log⁡m))

We start by sorting both flowers and people. This costs O(n⋅log⁡n) and O(m⋅log⁡m) respectively. Next, we perform O(m) iterations. At each iteration, we perform some heap operations. The cost of these operations is dependent on the size of the heap. Our heap cannot exceed a size of n, so these operations cost O(log⁡n).

There are some other linear time operations that don't affect our time complexity. In total, our time complexity is O(n⋅log⁡n+m⋅(log⁡n+log⁡m)).

Space complexity: O(n+m)

We create an array sortedPeople of length m. dic also grows to a length of m, and heap can grow to a size of O(n).