<https://leetcode.com/problems/minimum-interval-to-include-each-query/description/>

You are given a 2D integer array intervals, where intervals[i] = [lefti, righti] describes the ith interval starting at lefti and ending at righti **(inclusive)**. The **size** of an interval is defined as the number of integers it contains, or more formally righti - lefti + 1.

You are also given an integer array queries. The answer to the jth query is the **size of the smallest interval** i such that lefti <= queries[j] <= righti. If no such interval exists, the answer is -1.

Return **an array containing the answers to the queries**.

**Example 1:**

**Input:** intervals = [[1,4],[2,4],[3,6],[4,4]], queries = [2,3,4,5]

**Output:** [3,3,1,4]

**Explanation:** The queries are processed as follows:

- Query = 2: The interval [2,4] is the smallest interval containing 2. The answer is 4 - 2 + 1 = 3.

- Query = 3: The interval [2,4] is the smallest interval containing 3. The answer is 4 - 2 + 1 = 3.

- Query = 4: The interval [4,4] is the smallest interval containing 4. The answer is 4 - 4 + 1 = 1.

- Query = 5: The interval [3,6] is the smallest interval containing 5. The answer is 6 - 3 + 1 = 4.

**Example 2:**

**Input:** intervals = [[2,3],[2,5],[1,8],[20,25]], queries = [2,19,5,22]

**Output:** [2,-1,4,6]

**Explanation:** The queries are processed as follows:

- Query = 2: The interval [2,3] is the smallest interval containing 2. The answer is 3 - 2 + 1 = 2.

- Query = 19: None of the intervals contain 19. The answer is -1.

- Query = 5: The interval [2,5] is the smallest interval containing 5. The answer is 5 - 2 + 1 = 4.

- Query = 22: The interval [20,25] is the smallest interval containing 22. The answer is 25 - 20 + 1 = 6.

**Constraints:**

* 1 <= intervals.length <= 10^5
* 1 <= queries.length <= 10^5
* intervals[i].length == 2
* 1 <= lefti <= righti <= 10^7
* 1 <= queries[j] <= 10^7

**Attempt 1: 2023-12-15**

**Solution 1: Binary Search (10 min, TLE 34/42)**

class Solution {

    public int[] minInterval(int[][] intervals, int[] queries) {

        int n = queries.length;

        int[] result = new int[n];

        Arrays.fill(result, 10000001);

        for(int i = 0; i < n; i++) {

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

                if(binarySearch(intervals[j], queries[i])) {

                    result[i] = Math.min(result[i], intervals[j][1] - intervals[j][0] + 1);

                }

            }

            if(result[i] == 10000001) {

                result[i] = -1;

            }

        }

        return result;

    }

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

        int lo = arr[0];

        int hi = arr[1];

        while(lo <= hi) {

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

            if(mid == val) {

                return true;

            } else if(mid > val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return false;

    }

}

Time Compleity: O(N^2\*logN)

Space Complexity: O(1)

**注意：本题无法直接移植L2251的第一种解法，即如下的代码：**

**因为L1851需要获得的是最短的包含query值的区间，而不是区间的个数，L2251可以通过构建treemap delta => presum => presum数组存储了对应具体坐标的区间个数，但是L1851中就算求出treemap delta和presum，然后获得具体坐标对应的区间个数，这对于获得对应具体坐标的最短区间没有直接帮助，我们甚至需要额外建立一个基于presum存储了presum上所有坐标和对应一个坐标上所有包含该坐标的区间的区间长度的map，这个map的构造并不容易，所以我们需要一种比较另类的融合了Priority Queue的Line Sweep模型，见Solution 2**

// L2251. Solution 1:  Sweep Line + Sorting + TreeMap + Binary Search (60 min)

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;

    }

}

**Solution 2: Priority Queue + Sorting (180 min)**

class Solution {

    // 思路和L2251.Number of Flowers in Full Bloom的Priority Queue

    // + Sorting做法非常相似, L2251是给了一批人看花的时间坐标，然后查找

    // 每个时间坐标被多少朵花的盛开周期包含，L1851是给了一批数字，然后

    // 查找每个数字被多少给定区间包含，并且更进一步的是查找包含该数字的

    // 所有区间中最短的区间，如果映射到L2251就是查找所有包含时间坐标的花

    // 的盛开周期中最短的花期

    public int[] minInterval(int[][] intervals, int[] queries) {

        // Create a sorted version of queries called 'sorted\_queries'

        int[] sorted\_queries = Arrays.copyOf(queries, queries.length);

        Arrays.sort(sorted\_queries);

        // Sort intervals based on 'start'

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

        // Initialize a hash map that maps a query to the minimum

        // interval length it belongs to

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

        // A minPQ based on 'length' of intervals

        // element on minPQ => {interval length, interval end}

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // To find the intervals with start less than a given 'query',

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

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

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

        // successive 'query'.

        int i = 0;

        for(int query : sorted\_queries) {

            // While intervals[i][0] < 'query' (the interval at i already started,

            // means interval start before 'query', push intervals[i][1] (when the

            // interval finishes) to heap and increment i, alone with intervals[i][1]

            // we also need another information as length of interval as

            // 'intervals[i][1] - intervals[i][0] + 1' binding with for final result

            while(i < intervals.length && intervals[i][0] <= query) {

                minPQ.offer(new int[]{intervals[i][1] - intervals[i][0] + 1, intervals[i][1]});

                i++;

            }

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

            // 'query', means the interval finishes before 'query' coming,

            // pop from heap

            while(!minPQ.isEmpty() && minPQ.peek()[1] < query) {

                minPQ.poll();

            }

            // The answer for 'query' is simply the interval length stored

            // in peek element of the heap, in case none of the intervals

            // contain 'query', we default interval length as -1

            if(!minPQ.isEmpty()) {

                map.put(query, minPQ.peek()[0]);

            } else {

                map.put(query, -1);

            }

        }

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

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

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

        }

        return result;

    }

}

Time Complexity: O(n log n + m log m)

The time complexity of the code can be broken down into the following parts:

Sorting the intervals: n is the number of intervals, and this step takes O(n log n) time using the Timsort algorithm (Python's built-in sorting algorithm).

Sorting the queries along with their original indices: m is the number of queries, and this step also takes O(m log m) time.

Traversing the sorted queries and intervals: Each interval and query is checked at most once, which is O(m + n).

Heap operations: In the worst case, every interval can be pushed and popped once. Given that the priority queue (min-heap) can contain at most n elements, each push and pop operation takes O(log n). Therefore, if all n intervals are pushed and popped, it would take O(n log n).

Summing up the complexities, we have O(n log n) + O(m log m) for the sorting operations and O(m + n) for the linear traversal, and O(n log n) for heap operations. The most significant terms are the ones that involve sorting and heap operations, thus overall time complexity is O(n log n + m log m).

Space Complexity: O(n + m)

For space complexity:

The intervals and the extra space for storing the sorted queries and their original indices require O(n) and O(m) space respectively.

The priority queue (min-heap) can contain up to n elements at any time, and therefore, requires O(n) space.

Therefore, the space complexity is O(n + m) when considering the intervals, queries, and the min-heap storage.

**思路和L2251.Number of Flowers in Full Bloom的Priority Queue + Sorting做法非常相似, L2251是给了一批人看花的时间坐标，然后查找每个时间坐标被多少朵花的盛开周期包含，L1851是给了一批数字，然后查找每个数字被多少给定区间包含，并且更进一步的是查找包含该数字的所有区间中最短的区间，如果映射到L2251就是查找所有包含时间坐标的花的盛开周期中最短的花期**

**Refer L2251 Priority Queue + Sorting Solution**

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://www.cnblogs.com/cnoodle/p/17561751.html>

思路是扫描线。感觉思路不难但是实现起来有很多细节需要注意。

首先我们要将 intervals 按 start 排序，同时将 queries 和他们各自的 index 包装成一个二维数组且按照 queryVal 排序。再来我们需要一个最小堆存储所有的 intervals，这个堆按 interval 的宽度（end - start）排序。

接下来我们开始遍历 queries，对于每个 queryVal，我们将所有 start <= queryVal 的 interval 都放入最小堆。然后我们从最小堆中弹出所有 end < queryVal 的元素，因为这些 interval 虽然 start 小于 queryVal 但是 end 也小于，等于是这些 intervals 在 queryVal 的左侧，所以要舍弃。此时如果最小堆里还有 intervals，当前的 queryVal 则是能落在这些 intervals 中间的。此时我们再从最小堆中弹出堆顶元素，这个元素就是既能 cover 住 queryVal，且又是宽度最小的那个。

时间O(n \* nlogn) - 要处理 n 个 queryVal，对于每个 queryVal，我们都需要将所有 intervals 放入最小堆过滤一遍

空间O(n)

Java实现

class Solution {

    public int[] minInterval(int[][] intervals, int[] queries) {

        int n = queries.length;

        int[][] queriesWithIndex = new int[n][2];

        for (int i = 0; i < n; i++) {

            queriesWithIndex[i] = new int[] { queries[i], i };

        }

        // 区间按start排序

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

        // query按val排序

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

        // 最小堆，按interval的宽度排序

        PriorityQueue<int[]> minHeap = new PriorityQueue<>((a, b) -> ((a[1] - a[0]) - (b[1] - b[0])));

        int[] res = new int[n];

        int j = 0;

        for (int i = 0; i < n; i++) {

            int queryVal = queriesWithIndex[i][0];

            int queryIndex = queriesWithIndex[i][1];

            while (j < intervals.length && intervals[j][0] <= queryVal) {

                minHeap.offer(intervals[j]);

                j++;

            }

            while (!minHeap.isEmpty() && minHeap.peek()[1] < queryVal) {

                minHeap.poll();

            }

            res[queryIndex] = minHeap.isEmpty() ? -1 : (minHeap.peek()[1] - minHeap.peek()[0] + 1);

        }

        return res;

    }

}

**Refer to**

<https://algo.monster/liteproblems/1851>

**Problem Description**

In this problem, we are given two primary inputs:

* A 2D integer array named intervals. Each subarray contains two integer values [left\_i, right\_i] which represent an interval that starts at left\_i and ends at right\_i (inclusive). The size of an interval is the number of integers within that range, which is calculated using the formula right\_i - left\_i + 1.
* An integer array called queries. Each value queries[j] within this array denotes a question asking for the size of the smallest interval i such that left\_i <= queries[j] <= right\_i. If no interval satisfies this condition for a given query, the answer is -1.

The task is to go through each query and determine the answer by finding the appropriate interval that contains the query value and has the smallest size. The result should be returned as an array of answers corresponding to the original order of the queries.

**Intuition**

The intuition behind the solution approach is to efficiently match queries to their smallest enclosing intervals (if they exist). We need a way to handle these possible overlaps and find the smallest intervals quickly as the number of queries can be quite large. The approach can be broken down into a few key ideas:

1. Sort the intervals by their starting point to iterate over them in order.
2. Sort the queries by their value but keep track of their original indices so we can store the answer in the right place later.
3. Use a priority queue (pq) to maintain a set of active intervals at any given query value. This priority queue will store pairs of (interval size, right endpoint) sorted by size.
4. For each query value x, add all the intervals that start at or before x to the priority queue. This ensures that at any iteration, the priority queue contains all intervals that could possibly contain x.
5. Remove from the priority queue any intervals that end before x, as these intervals can no longer contain the query value.
6. Look at the smallest interval remaining in the priority queue (which is at the front of the queue due to the sorting by size) to answer the current query. If the priority queue is empty, this means there are no valid intervals for the query, and we should record -1.
7. Repeat the process for all queries. Since the queries and intervals are sorted, we can do this in a single pass using two pointers, where one iterates over the sorted intervals and the other iterates over the sorted queries.

By using a sorted structure and a priority queue, we can efficiently manage the set of potential intervals for each query, enabling us to quickly find the smallest interval when it exists.

**Solution Approach**

The solution to the problem leverages both sorting and a heap (priority queue) to efficiently determine the smallest interval containing each query. The implementation proceeds as follows:

1. Sort Intervals and Queries: The first step is to sort the intervals based on their starting points, so they can be processed in ascending order. Simultaneously, the queries are sorted based on their value, but each query is transformed into a tuple containing its value and original index. This transformation allows us to return the results according to the original order of queries.
2. Initialize Data Structures: A priority queue in Python is implemented using a heap, specifically through the built-in heapq module. The pq list will serve as our priority queue for the intervals. A list ans of size m (number of queries) is also initialized to -1 and will be used to store the final answer to each query.
3. Iterate Over Queries: We iterate over each sorted query value x along with the corresponding original index j. The goal during this iteration is to add any new intervals that could potentially contain x and to remove intervals that no longer apply.
4. Add Matching Intervals to Priority Queue:

* Using a pointer i  intervals x
* If it does, we calculate the size of the interval by b - a + 1(where a and b are the starting and ending points of the interval) and push a tuple (size, b) onto the priority queue (heap).
* This ensures that the smallest intervals (by size) will be at the top of the heap.
* Increment to move to the next interval for future queries.

1. Remove Invalid Intervals from Priority Queue:

* Before checking the smallest interval, we first clear out any intervals from the priority queue whose end point is before the query value x
* This is done by popping elements off the heap as long as the smallest interval's end point is less than x

1. Retrieve the Smallest Valid Interval:

* After potentially removing intervals, if the priority queue is not empty, the interval at the top of the heap is the smallest interval that contains x
* We set ans[j] (where j is the original index of the query) to the size of this interval.

1. Return Results: Finally, after all queries have been processed, we return the array ans, which contains the size of the smallest interval corresponding to each query in the order they were originally presented.

By sorting the intervals and queries and using a min-heap to keep track of active intervals, this approach efficiently manages the various size intervals that could satisfy the range condition for each query, allowing us to determine the minimum interval size required in an optimized manner.

**Example Walkthrough**

Let's illustrate the solution approach using a small example.

Suppose our input intervals array is [[1,4], [2,5], [6,8]], and the queries array is [3, 5, 7].

1. Sort Intervals and Queries:

* Firstly, we sort the intervals by their start point, which doesn’t change the order in this case: [[1,4], [2,5], [6,8]]
* Then, sort the queries [3, 5, 7] They are already sorted, so we transform them into pairs of values and original indices: [(3, 0), (5, 1), (7, 2)]

1. Initialize Data Structures:

* We create a priority queue pq which starts as an empty list []
* An ans list is initialized to store the results: [-1, -1, -1] (since there are 3 queries).

1. Iterate Over Queries:

* Begin by processing the query value 3 and its corresponding index 0

1. Add Matching Intervals to Priority Queue:

* We check intervals starting with [1,4], Since 1 <= 3, we calculate its size 4 - 1 + 1 = 4 and add (4, 4) to the priority queue
* Next, the interval [2,5], Since 2 <= 3, its size is 5 - 2 + 1 = 4, and we add (4, 5) to the priority queue.
* The queue now has [(4, 4), (4, 5)]

1. Remove Invalid Intervals from Priority Queue:

* No intervals end before 3, so nothing is removed.

1. Retrieve the Smallest Valid Interval:

* The top of the heap is (4, 4), so ans at the original index 0 is updated to 4.

Next, we process the remaining queries. For 5, both [1,4] and [2,5] are considered, but [1,4] is removed since it doesn't contain 5. The heap has [(4, 5)], and the answer to query 5 is 4.

For 7, [6,8] is considered and (3, 8) is added to the heap. After removing intervals that end before 7, we get the smallest interval size 3 for query 7.

1. Return Results

* The final results array is [4, 4, 3]: the smallest intervals containing each query value respectively.

By following these steps, we effectively matched each query to its smallest enclosing interval, if such one existed, and obtained the solution to our problem.