4. Intervals

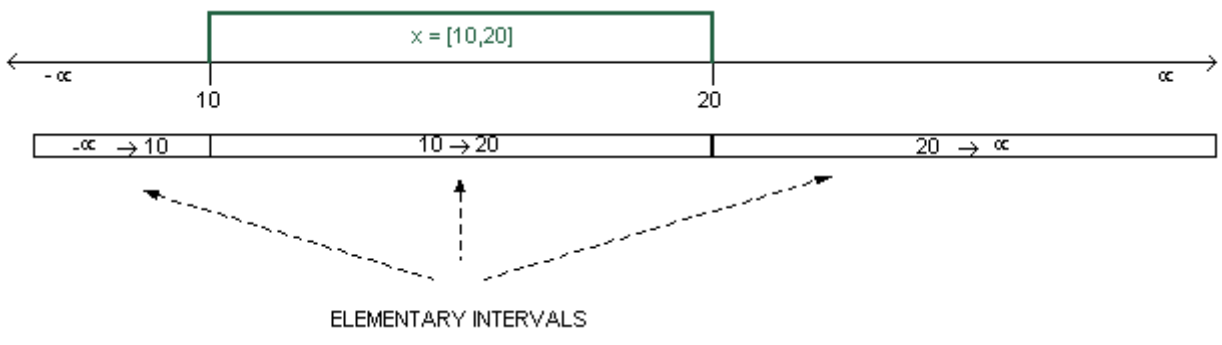
#intervals

What are intervals?

An *interval* is a pair of integers `[a,b]` such that *a* < *b*.

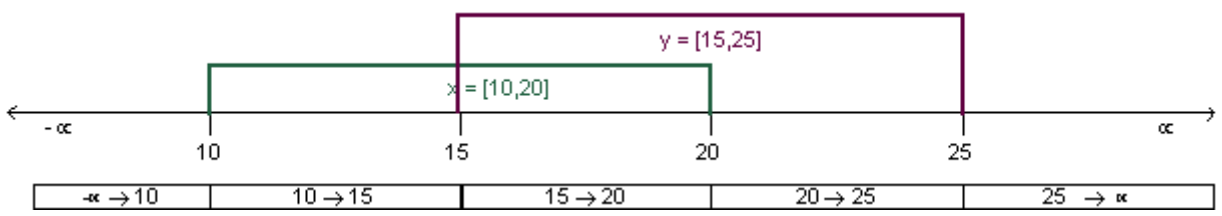
The endpoints *a* and *b* of each interval `[a,b]` divide the integer line into partitions called *elementary intervals*.

The interval x= `[10, 20]` has been added to the integer line. Notice that one interval cuts the line at two points:

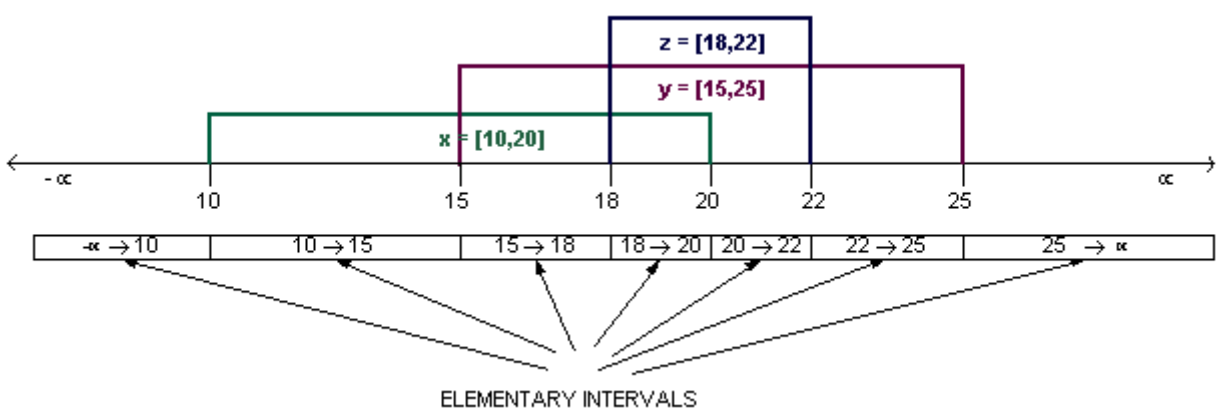


See what happens as we add new intervals. Notice how many new elementary intervals we are creating.

Add *y* = `[15, 25]`:



Add *z* = `[18, 22]`:



Given *n* intervals $[a_i, b_i]$, for `i = 1 ... n`, *exactly* how many *elementary* intervals are there, assuming that no intervals $[a_i, b_i]$ share endpoints?

We get `2n + 1` sub-intervals when there are `n` intervals on the integer line that do not share endpoints.

Every interval can be expressed as an aggregate of the sub-intervals that it spans:

	Interval	spans Sub-Intervals
x	[10,20]	[10,15], [15,18], [18-20]
y	[15,25]	[15,18], [18,20], [20-22], [22,25]
z	[18, 22]	[18,20], [20,22]

Reference Link

1. Insert Interval

Medium

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the `ith` interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` *after the insertion*.

Example 1:

Input: intervals = `[[1,3],[6,9]]`, newInterval = `[2,5]`
Output: `[[1,5],[6,9]]`

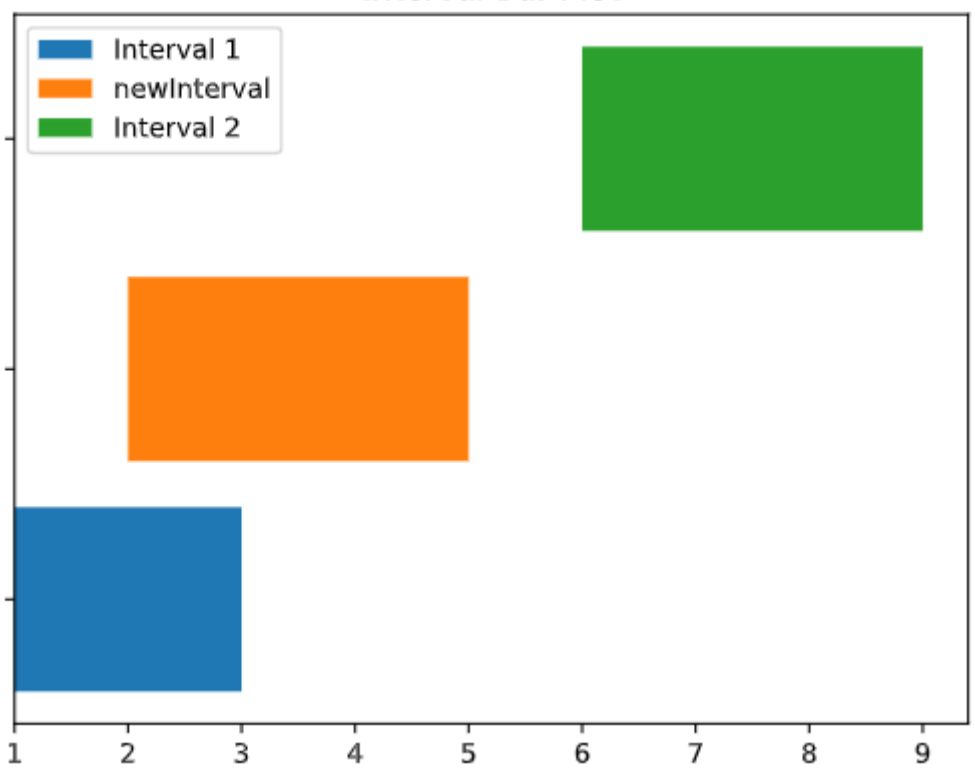
Example 2:

Input: intervals = `[[1,2],[3,5],[6,7],[8,10],[12,16]]`, newInterval = `[4,8]`
Output: `[[1,2],[3,10],[12,16]]`
Explanation: Because the new interval `[4,8]` overlaps with `[3,5],[6,7],[8,10].`

Constraints:

- `0 <= intervals.length <= 104`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 105`
- `intervals` is sorted by `starti` in **ascending** order.
- `newInterval.length == 2`
- `0 <= start <= end <= 105`

Before



Code

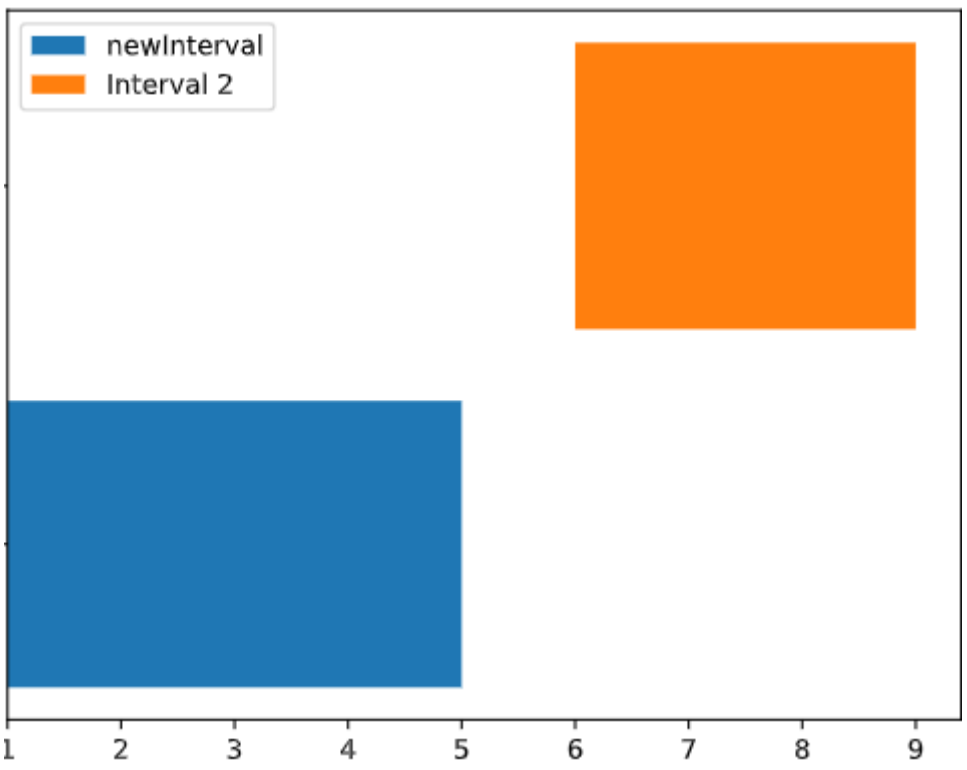
```
class Solution:
        def insert(self, intervals, newInterval):
                res = []
                for i in range(len(intervals)):
                        if newInterval[1] < intervals[i][0]:
                                res.append(newInterval)
                                return res + intervals[i:]
                        elif newInterval[0] > intervals[i][1]:
                                res.append(intervals[i])
                        else:
                                newInterval = [min(newInterval[0], intervals[i][0]), max(newInterval[1], intervals[i][1])]
                res.append(newInterval)
                return res
s = Solution()
result = s.insert([[1,3],[6,9]], [2,5])
print(result)
```

Breakdown

```
def insert(self, [[1,3], [6,9]], [2,5]):
        res = []
        for i in range(2) :
                #iteration 1
```

```
                i = 0
                intervals[0] = [1,3]
                if 5 < 1 --> False
                elif 2 > 3 --> False
                else:
                        newInterval = [min(2, 1), max(5, 3)] = [1, 5]
                #iteration 2
                i = 1
                intervals[1] = [6, 9]
                if 5 < 6 --> True:
                        res.append([1,5]) >> res = [[1,5]]
                        return [[1,5]] + intervals[1:] >> [[1,5]] + [6,9] >> [[1,5], [6,9]]
```

After



2. Merge Intervals

Medium

Given an array of `intervals` where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Example 1:

Input: intervals = `[[1,3],[2,6],[8,10],[15,18]]`
Output: `[[1,6],[8,10],[15,18]]`
Explanation: Since intervals `[1,3]` and `[2,6]` overlap, merge them into `[1,6]`.

Example 2:

Input: intervals = `[[1,4],[4,5]]`
Output: `[[1,5]]`
Explanation: Intervals `[1,4]` and `[4,5]` are considered overlapping.

Constraints:

- `1 <= intervals.length <= 104`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 104`

Code

```
def merge(intervals):
    intervals.sort()
    output = [intervals[0]]
    for start, end in intervals[1:]:
        lastEnd = output[-1][1]
        if start <= lastEnd:
            output[-1][1] = max(lastEnd, end)
```

```
        else:
            output.append([start,end])
    return output

if __name__ == "__main__":
        intervals = [[1, 3], [8, 10], [15, 18], [2, 6]]
        print(merge(intervals))
```

Breakdown

```
def merge([[1, 3], [8, 10], [15, 18], [2, 6]]):
        intervals.sort() >> [[1, 3], [2, 6], [8, 10], [15, 18]]
        output = [[1,3]]
        for start, end in [[2, 6], [8, 10], [15, 18]]:
                #iteration 1 >> [2,6]
                lastEnd = 3
                start = 2
                end = 6
                if 2 <= 3 --> True:
                        output[-1][1] = max(3, 6) >> output = [[1,6]]

                #iteration 2 >> [8,10]
                lastEnd = 6
                start = 8
                end = 10
                if 8<=6 --> False
                else --> True:
                        output.append([8,10]) >> output = [[1,6],[8,10]]

                #iteration 3 >> [15,18]
                lastEnd = 10
                start = 15
                end = 18
                if  15<=10 --> False
                else --> True
                        ouput.append([15,18]) >> output = [[1,6],[8,10],[15,18]]

        return [[1,6],[8,10],[15,18]]
```

3. Non Overlapping Intervals

Medium

Given an array of intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Example 1:

Input: intervals = `[[1,2],[2,3],[3,4],[1,3]]`
Output: 1
Explanation: `[1,3]` can be removed and the rest of the intervals are non-overlapping.

Example 2:

Input: intervals = `[[1,2],[1,2],[1,2]]`
Output: 2
Explanation: You need to remove two `[1,2]` to make the rest of the intervals non-overlapping.

Example 3:

Input: intervals = `[[1,2],[2,3]]`
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

Constraints:

- `1 <= intervals.length <= 105`
- `intervals[i].length == 2`
- `-5 * 104 <= starti < endi <= 5 * 104`

Code

```
def eraseOverlapIntervals(intervals):
    intervals.sort()
    res = 0
    prevEnd = intervals[0][1]
    for start, end in intervals[1:]:
        if start >= prevEnd:
            prevEnd = end
        else:
            res += 1
            prevEnd = min(prevEnd, end)
    return res
if __name__ == "__main__":
    intervals = [[1,2], [2,3], [3,4], [1,3]]
    result = eraseOverlapIntervals(intervals)
    print(result)
```

Breakdown

```
def eraseOverlapIntervals([[1,2], [2,3], [3,4], [1,3]]):
        intervals.sort() >> [[1, 2], [1, 3], [2, 3], [3, 4]]
        res = 0
        prevEnd = intervals[0][1] >> 2
        for start, end in [[1, 3], [2, 3], [3, 4]]:
                #iter 1 >> [1,3]
                prevEnd = 2
                start = 1
                end = 3
                if 1 >= 2 --> False
                else --> True:
                        res += 1 >> res = 1 # we have to erase this interval or we have erased it
                        prevEnd = min(2, 3) >> prevEnd = 2

                #iter 2 >> [2,3]
                prevEnd = 2
                start = 2
                end = 3
                if 2>=2 --> True:
                        prevEnd = 3

                #iter 3 >> [3,4]
                prevEnd = 3
                start = 3
                end = 4
                if 3>=3 --> True:
                        prevEnd = 4

        return 1
```

4. Meeting Rooms

Medium

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...] (si < ei)`, determine if a person could attend all meetings.
Example:

```
Input: intervals = [(0,30),(5,10),(15,20)]
Output: false Explanation: (0,30), (5,10) and (0,30),(15,20) will conflict
```

```
class MeetingRooms:
    def canAttendMeeting(self, intervals):
        intervals.sort(key = lambda i:i[0])
        for i in range(1,len(intervals)):
            i1 = intervals[i-1]
```

```
            i2 = intervals[i]
            if i1[1] > i2[0]:
                return False
        return True

if __name__ == "__main__":
    meetings = MeetingRooms()
    intervals = [(0,30),(5,10),(15,20)]
    attend_all = meetings.canAttendMeeting(intervals)
    print(attend_all)
```

Breakdown

```
def canAttendMeeting([(0,30),(5,10),(15,20)]):
        intervals.sort(key = lambda i: i[0]) >> [(0,30),(5,10),(15,20)]
        for i in range(1,3):
                i1 = intervals[1-1] >> intervals[0] >> (0, 30)
                i2 = intervals[1] >> (5,10)
                if 30 > 5 --> True:
                        return False
```

5.Meeting Rooms II

Medium

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...] (si < ei)`, find the minimum number of conference rooms required.)

Example

```
Input: intervals = [(0,30),(5,10),(15,20)]
Output: 2 Explanation: We need two meeting rooms
room1: (0,30)
room2: (5,10),(15,20)
```

```
class MeetingRooms:
    def minMeetingRooms(self, intervals):
        start = sorted([i[0] for i in intervals])
        end = sorted([i[1] for i in intervals])
        res, count = 0, 0
        s, e = 0, 0
        while s<len(intervals):
            if start[s] < end[e]:
                s+=1
                count+=1
            else:
                e += 1
                count-=1
            res = max(res, count)
        return res

if __name__ == "__main__":
    meetings = MeetingRooms()
    intervals = [(0,30),(5,10),(15,20)]
    rooms_needed = meetings.minMeetingRooms(intervals)
    print(rooms_needed)
```

Breakdown

```
def minMeetingRooms([(0,30),(5,10),(15,20)]):
        start = [0, 5, 15]
        end = [10, 20, 30]
        res, count = 0, 0
```

```
        s, e = 0, 0

        while s < 3:
        # s = 0 , e = 0, count=0, res = 0
                if start[0] < end[0] --> 0 < 10 --> True:
                        s+=1 >> s = 1
                        count+=1 >> count=1
                res = max(0, 1) = 1
        # s = 1, e = 0, count = 1, res = 1
                if start[1] < end[0] --> 5<10 --> True:
                        s+=1 >> s = 2
                        count+=1 >> count = 2
                res = max(1, 2) = 2
        # s = 2, e = 0, count = 2, res = 2
                if start[2] < end[0] --> 15<10 --> False
                else --> True:
                        e+=1 >> e=1
                        count-=1 >> count = 1
                res =  max(2, 1) = 2
        return 2
```

6. Meeting Rooms III

Hard

You are given an integer `n`. There are `n` rooms numbered from `0` to `n - 1`.

You are given a 2D integer array `meetings` where `meetings[i] = [starti, endi]` means that a meeting will be held during the **half-closed** time interval `[starti, endi)`. All the values of `starti` are **unique**.

Meetings are allocated to rooms in the following manner:

1. Each meeting will take place in the unused room with the **lowest** number.
2. If there are no available rooms, the meeting will be delayed until a room becomes free. The delayed meeting should have the **same** duration as the original meeting.
3. When a room becomes unused, meetings that have an earlier original **start** time should be given the room.

Return *the **number** of the room that held the most meetings.* If there are multiple rooms, return *the room with the **lowest** number.*

A **half-closed interval** `[a, b)` is the interval between `a` and `b` **including** `a` and **not including** `b`.

Example 1:

Input: n = 2, meetings = `[[0,10],[1,5],[2,7],[3,4]]`
Output: 0

Explanation:

```
- At time 0, both rooms are not being used. The first meeting starts in room 0.
- At time 1, only room 1 is not being used. The second meeting starts in room 1.
- At time 2, both rooms are being used. The third meeting is delayed.
- At time 3, both rooms are being used. The fourth meeting is delayed.
- At time 5, the meeting in room 1 finishes. The third meeting starts in room 1 for the time period [5,10).
- At time 10, the meetings in both rooms finish. The fourth meeting starts in room 0 for the time period [10,11).
Both rooms 0 and 1 held 2 meetings, so we return 0.
```

Example 2:

Input: n = 3, meetings = `[[1,20],[2,10],[3,5],[4,9],[6,8]]`
Output: 1

Explanation:

```
- At time 1, all three rooms are not being used. The first meeting starts in room 0.
- At time 2, rooms 1 and 2 are not being used. The second meeting starts in room 1.
- At time 3, only room 2 is not being used. The third meeting starts in room 2.
- At time 4, all three rooms are being used. The fourth meeting is delayed.
- At time 5, the meeting in room 2 finishes. The fourth meeting starts in room 2 for the time period [5,10).
- At time 6, all three rooms are being used. The fifth meeting is delayed.
- At time 10, the meetings in rooms 1 and 2 finish. The fifth meeting starts in room 1 for the time period [10,12).
Room 0 held 1 meeting while rooms 1 and 2 each held 2 meetings, so we return 1.
```

Constraints:

- `1 <= n <= 100`
- `1 <= meetings.length <= 105`
- `meetings[i].length == 2`
- `0 <= starti < endi <= 5 * 105`
- All the values of `starti` are **unique**.

```python
import heapq
class Solution(object):
    def mostBooked(self, n, meetings):
        busy = []
        available = [i for i in range(n)]

        count = [0]*n
        meetings.sort()

        for start, end in meetings:
            while busy and busy[0][0]<=start:
                _end, room = heapq.heappop(busy)
                heapq.heappush(available, room)
                if available:
                    room = heapq.heappop(available)
                    heapq.heappush(busy, (end, room))
                else:
                    time, room = heapq.heappop(busy)
                    heapq.heappush(busy, (time+ end-start, room))
                count[room] += 1
        return count.index(max(count))

if __name__ == '__main__':
    n = 2
    meetings = [[0,10],[1,5],[2,7],[3,4]]
    s = Solution()
    result = s.mostBooked(n, meetings)
    print(result)
```

7. Minimum Interval to Include Each Query

Hard

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 <= 105`
- `1 <= queries.length <= 105`
- `intervals[i].length == 2`
- `1 <= lefti <= righti <= 107`
- `1 <= queries[j] <= 107`

```
import heapq
class Intervals:
    def minInterval(self, intervals, queries):
        intervals.sort()
        minHeap = []
        res, i = {}, 0
        for q in sorted(queries):
            while i < len(intervals) and intervals[i][0] <= q:
                l, r = intervals[i]
                heapq.heappush(minHeap, (r-l+1, r))
                i += 1
            while minHeap and minHeap[0][1] < q:
                heapq.heappop(minHeap)
            res[q] = minHeap[0][0] if minHeap else -1
        return [res[q] for q in queries]

if __name__ == "__main__":
    i = Intervals()
    intervals = [[1,4],[2,4],[3,6],[4,4]]
    queries = [2,3,4,5]
    min_interval = i.minInterval(intervals, queries)
    print(min_interval)
```

Breakdown

```
def minInterval([[1,4],[2,4],[3,6],[4,4]], [2,3,4,5]):
        intervals.sort() >> [[1,4],[2,4],[3,6],[4,4]]
        minHeap = []
        res, i = {}, 0
        for q in sorted([2,3,4,5]):
                # q = 2, i =0
                while i<4 and intervals[i][0] <= 2:
                        i = 0
                        l, r = [1,4]
                        heapq.heappush([], (4-1+1, 4)) >> minHeap = [(4,4)]
                        i+=1 >> i = 1
                        # 1<4 and 2<=2 --> True
                        i = 1
                        l, r = [2,4]
                        heapq.heappush([],(4-2+1, 4)) >> minHeap = [(3,4),(4,4)]
                        i+=1 >> i = 2
                        # 2<4 and 2<=3 --> False
                # Ends while loop
                while [(3,4),(4,4)] and 4<2 --> False
                res[2] = minHeap[0][0] if minHeap else -1 >> res = {2:3}

                #similarly the following happens
                # q = 3, i = 2
                res = {2:3, 3: 3}
                # q = 4
                res = {2:3, 3:3, 4:1}
                # q = 5
                res = {2:3, 3:3, 4:1, 5:4}
        return [3, 3, 1, 4]
```

End

___