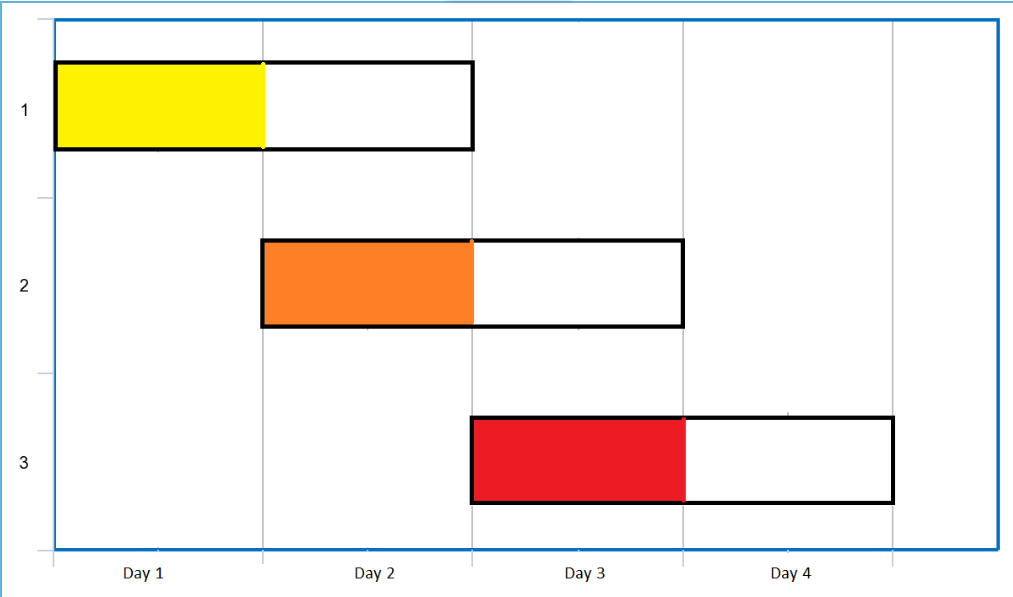
<https://leetcode.com/problems/maximum-number-of-events-that-can-be-attended/description/>

You are given an array of events where events[i] = [startDayi, endDayi]. Every event i starts at startDayiand ends at endDayi.

You can attend an event i at any day d where startTimei <= d <= endTimei. You can only attend one event at any time d.

Return *the maximum number of events you can attend*.

**Example 1:**



Input: events = [[1,2],[2,3],[3,4]]

Output: 3

Explanation: You can attend all the three events.

One way to attend them all is as shown.

Attend the first event on day 1.

Attend the second event on day 2.

Attend the third event on day 3.

**Example 2:**

Input: events= [[1,2],[2,3],[3,4],[1,2]]

Output: 4

**Example 3:**

Input: events = [[1,4],[4,4],[2,2],[3,4],[1,1]]

Output: 4

**Example 4:**

Input: events = [[1,100000]]

Output: 1

**Example 5:**

Input: events = [[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[1,7]]

Output: 7

**Constraints:**

* 1 <= events.length <= 10^5
* events[i].length == 2
* 1 <= startDayi <= endDayi <= 10^5

**Attempt 1: 2023-12-06**

**Solution 1: Priority Queue + Sorting (180 min)**

class Solution {

public int maxEvents(int[][] events) {

// 数组用开始时间排序

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

// 用最小堆按结束时间排序，含义是哪个会议最先结束就在堆顶

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

int minStart = 100001;

int maxEnd = 0;

for(int[] event : events) {

minStart = Math.min(minStart, event[0]);

maxEnd = Math.max(maxEnd, event[1]);

}

int count = 0;

int i = 0;

int n = events.length;

for(int d = minStart; d <= maxEnd; d++) {

// 会议结束时间已经早于今天的，就直接舍弃了，因为没法参加

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

minPQ.poll();

}

// 会议开始时间是今天的，把他的结束时间放到minPQ中

while(i < n && events[i][0] == d) {

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

i++;

}

// 目前minPQ中的元素都是在今天之前就开始的，并且结束时间 >= 今天

if(!minPQ.isEmpty()) {

minPQ.poll();

count++;

}

}

return count;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Step by Step**

e.g

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

1 4

----

4

-

2

-

34

--

1

-

after sort events = {{1,4},{1,1},{2,2},{3,4},{4,4}}

minStart = 1

maxEnd = 4

count = 0 -> counter for max meeting attend number

------------------------------------------------------------------------------------------

d = 1

(1) minPQ empty, no meeting abandoned

(2) i = 0, events[0][0] = 1, event[0] start by d = 1, add end time = 4 into minPQ, i++ = 1

i = 1, events[1][0] = 1, event[1] start by d = 1, add end time = 1 into minPQ, i++ = 2

minPQ = {1,4} -> sort by end time

(3) events on minPQ all start before/on today, and end after/on today

Attend the event that is ending soonest which on minPQ peek

so minPQ remove that meeting on peek

minPQ = {4} -> removed peek meeting end time = 1

count++ = 1

------------------------------------------------------------------------------------------

d = 2

(1) minPQ.peek() = 4 > d(=2), no meeting abandoned

(2) i = 2, events[2][0] = 2, event[2] start by d = 2, add end time = 2 into minPQ, i++ = 3

minPQ = {2,4} -> sort by end time

(3) events on minPQ all start before/on today, and end after/on today

Attend the event that is ending soonest which on minPQ peek

so minPQ remove that meeting on peek

minPQ = {4} -> removed peek meeting end time = 2

count++ = 2

------------------------------------------------------------------------------------------

d = 3

(1) minPQ.peek() = 4 > d(=3), no meeting abandoned

(2) i = 3, events[3][0] = 3, events[3] start by d = 3, add end time = 4 into minPQ, i++ = 4

minPQ = {4,4} -> sort by end time

(3) events on minPQ all start before/on today, and end after/on today

Attend the event that is ending soonest which on minPQ peek

so minPQ remove that meeting on peek

minPQ = {4} -> removed peek meeting end time = 4

count++ = 3

------------------------------------------------------------------------------------------

d = 4

(1) minPQ.peek() = 4 == d(=4), no meeting abandoned

(2) i = 4, events[4][0] = 4, events[4] start by d = 4, add end time = 4 into minPQ, i++ = 5

minPQ = {4,4} -> sort by end time

(3) events on minPQ all start before/on today, and end after/on today

Attend the event that is ending soonest which on minPQ peek

so minPQ remove that meeting on peek

minPQ = {4} -> removed peek meeting end time = 4

count++ = 4

------------------------------------------------------------------------------------------

d = 5 > maxEnd = 4 end for loop

final count = 4 is the maximum meeting able to attend

**Refer to**

<https://www.cnblogs.com/cnoodle/p/13744163.html>

思路是贪心。**这个题看着好像可以用扫描线的思路做，其实是没什么关系的，因为这道题不涉及区间左右边界的比较和尝试找overlap或者gap这一类的操作。**正确的思路是你需要一个最小堆存储events的结束时间endDay。首先还是对input排序，这里我们是对events的startDay排序（注意跟最小堆存储内容的不同）。因为会议的结束日期最大可以到100000所以我们从1遍历到100000，意思是从第1天遍历到第100000天，然后去看每一天是否能参加event。

在第d天的时候，

* 如果此时最小堆不为空且堆顶元素（的结束时间）小于今天的天数d，则意味着我们无法去这个event了，直接从最小堆中弹出即可
* 此时如果最小堆还有东西，剩下的东西都是一些event的结束时间endDay，且这些结束时间晚于今天的天数d，则弹出一个并且res++，意思是今天可以去其中的一个event
* 如果有event的startDay是今天，则把他的endDay加入最小堆

时间O(nlogn)

空间O(n)

Java实现

class Solution {

public int maxEvents(int[][] events) {

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

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

int res = 0;

int i = 0;

int n = events.length;

for (int d = 1; d <= 100000; d++) {

// 会议结束时间已经早于今天的，就直接舍弃了，因为没法参加

while (!pq.isEmpty() && pq.peek() < d) {

pq.poll();

}

// 会议开始时间是今天的，把他的结束时间放到pq中

while (i < n && events[i][0] == d) {

pq.offer(events[i][1]);

i++;

}

// 目前pq中的元素都是在今天之前就开始的，并且结束时间 >= 今天

if (!pq.isEmpty()) {

pq.poll();

res++;

}

}

return res;

}

}

**Refer to**

<https://algo.monster/liteproblems/1353>

## Problem Description

In this problem, we are given a list of events, where each event is represented by a start day and an end day, indicating the duration during which the event takes place. We can choose to attend an event on any day from the start day to the end day inclusive. However, we can only attend one event at any given day. Our goal is to maximize the number of events that can be attended.

## Intuition

The intuition behind the solution is to prioritize attending events based on their end dates because we want to ensure we do not miss out on events that are about to end. For this reason, a greedy algorithm works efficiently — sorting the events by their end times could help us attend as many as possible.

However, simply sorting by the end times is not adequate since we also have to consider the starting times. Therefore, we create a priority queue (min-heap) where we will keep the end days of events that are currently available to attend. We also use two variables to keep track of the minimum and maximum days we need to cover.

As we iterate through each day within the range, we do the following:

* Remove any events that have already ended.
* Add all events that start on the current day to the priority queue.
* Attend the event that is ending soonest (if any are available).

By using a priority queue (min-heap), we ensure that we are always attending the event with the nearest end day, hence maximizing the number of events we can attend.

## Solution Approach

The solution uses a greedy approach combined with a priority queue (min-heap) to facilitate the process of deciding which event to attend next. Specifically, it applies the following steps:

**Initialization:**

A dictionary d is used to map each start day to a list of its corresponding end days. This enables easy access to events starting on a particular day.

Two variables, i and j, are initialized to inf and 0, respectively, to track the minimum start day and the maximum end day across all events.

**Building the dictionary:**

The solution iterates over each event and populates the dictionary d with the start day as the key and a list of end days as the value.

It also updates i to the minimum start day and j to the maximum end day encountered.

**Setting up a min-heap:**

A priority queue (implemented as a min-heap using a list h) is created to keep track of all the end days of the currently available events.

**Iterating over each day:**

For each day s in the range from the minimum start day i to the maximum end day j inclusive:

While there are events in the min-heap that have ended before day s, they are removed from the heap since they can no longer be attended.

All events starting on day s are added to the min-heap with their end days.

If the min-heap is not empty, it means there is at least one event that can be attended. The event with the earliest end day is attended (removed from the heap), and the answer count ans is incremented by one.

**Returning the result:**

After iterating through all the days, the ans variable that has been tracking the number of events attended gives us the maximum number of events that can be attended.

In summary, by using a combination of a dictionary to map start days to events, a min-heap to efficiently find the soonest ending event that can be attended, and iteration over each day, the solution efficiently computes the maximum number of events that one can attend.

### Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we are given the following list of events:

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

**Initialization:**

We create a dictionary d, and two variables i = inf and j = 0.

**Building the dictionary:**

We iterate over the events:

For event [1,4], we update d with {1: [4]} and set i = 1 and j = 4.

For event [4,4], we update d with {1: [4], 4: [4]}. Variables i and j remain unchanged.

For event [2,2], we update d with {1: [4], 2: [2], 4: [4]}. Variables i and j remain unchanged.

For event [3,4], we update d with {1: [4], 2: [2], 3: [4], 4: [4]}. Variables i and j remain unchanged.

For event [1,1], we update d with {1: [4, 1], 2: [2], 3: [4], 4: [4]}. Variables i and j remain unchanged.

**Setting up a min-heap:**

We initialize an empty min-heap list h.

**Iterating over each day:**

We have i = 1 and j = 4, so we iterate from day 1 to day 4

On day 1:

We add all end days of events starting on day 1 to h, so h becomes [4, 1].

We pop 1 from h as it's the earliest end day, attend this event, and increment ans to 1.

On day 2:

There's no event ending before day 2, so nothing is removed from h.

We add the end day of the event starting on day 2 to h, so h becomes [4, 2].

We pop 2 from h, attend this event, and increment ans to 2.

On day 3:

There's no event ending before day 3, so nothing is removed from h.

We add the end day of the event starting on day 3 to h, so h becomes [4, 4].

We pop 4 from h (either one, as both have the same end day), attend this event, and increment ans to 3.

On day 4:

Since there is only one event with an end day of 4 left in h, we attend it and increment ans to 4.

We also check for more events starting today which is one [4, 4] and add it to the heap.

We then attend this event and increment ans to 5.

**Returning the result:**

After iterating through all days, we find that ans = 5, which means we could attend a total of 5 events.

In this example, by using the greedy approach outlined in the solution, we were methodically able to maximize the number of events that could be attended by ensuring we attend the ones ending soonest first.

class Solution {

public int maxEvents(int[][] events) {

// Create a map to associate start days with a list of their respective end days

Map<Integer, List<Integer>> dayToEventsMap = new HashMap<>();

int earliestStart = Integer.MAX\_VALUE; // Initialize earliest event start day

int latestEnd = 0; // Initialize latest event end day

// Process the events to populate the map and find the range of event days

for (int[] event : events) {

int startDay = event[0];

int endDay = event[1];

// Map the start day to the end day of the event

dayToEventsMap.computeIfAbsent(startDay, k -> new ArrayList<>()).add(endDay);

// Update earliest start and latest end

earliestStart = Math.min(earliestStart, startDay);

latestEnd = Math.max(latestEnd, endDay);

}

// Create a min-heap to manage event end days

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

int attendedEventsCount = 0; // Initialize the count of events attended

// Iterate over each day within the range of event days

for (int currentDay = earliestStart; currentDay <= latestEnd; ++currentDay) {

// Remove past events that have already ended

while (!eventsEndingQueue.isEmpty() && eventsEndingQueue.peek() < currentDay) {

eventsEndingQueue.poll();

}

// Add new events that start on the current day

List<Integer> eventsStartingToday = dayToEventsMap.getOrDefault(currentDay, Collections.emptyList());

for (int endDay : eventsStartingToday) {

eventsEndingQueue.offer(endDay);

}

// Attend the event that ends the earliest, if any are available

if (!eventsEndingQueue.isEmpty()) {

eventsEndingQueue.poll();

++attendedEventsCount; // Increment the count of events attended

}

}

return attendedEventsCount;

}

}

## Time and Space Complexity

The given Python code aims to find the maximum number of events one can attend, given a list of events where each event is represented by a start and end day. The code uses a greedy algorithm with a min-heap to facilitate the process.

### Time Complexity:

Let's analyze the time complexity step by step:

Building the dictionary d has a complexity of O(N), where N is the number of events since we iterate through all the events once.

Populating the min-heap h on each day has a variable complexity. In the worst case, we could be adding all events to the heap on a single day which will be O(NlogN) due to N heap insertions (heap push operations), each with O(logN) complexity.

The outer loop runs from the minimum start time i to the maximum end time j. Therefore, in the worst-case scenario, it would run O(j - i) times.

Inside this loop, we perform a heap pop operation for each day that an event ends before the current day. Since an event end can only be popped once, all these operations together sum up to O(NlogN), as each heap pop operation is O(logN) and there are at most N such operations throughout the loop.

We also perform a heap pop operation when we can attend an event, and this happens at most N times (once for each event).

Adding these complexities, we have:

For the worst case, a complexity of O(NlogN + (j - i)) for the loop, with O(NlogN) potentially dominating the overall time complexity when (j - i) is not significantly larger than N.

In conclusion, the time complexity of the code is O(NlogN + (j - i)). However, (j - i) may be considered negligible compared to NlogN for large values of N, yielding an effective complexity of O(NlogN).

### Space Complexity:

Let's analyze the space complexity:

The dictionary d can hold up to N entries in the form of lists, with each list containing at least one element, but potentially up to N end times in the worst case. Therefore the space required for d is O(N).

The min-heap h also requires space which in the worst-case scenario may contain all N events at once. Thus, the space complexity due to the heap is O(N).

The min-heap h and the dictionary d represent the auxiliary space used by the algorithm. Since they both have O(N) space complexity, the overall space complexity is also O(N), assuming that the space required for input and output is not taken into consideration, which is standard in space complexity analysis.

**Refer to**

<https://leetcode.com/problems/maximum-number-of-events-that-can-be-attended/solutions/510262/detailed-analysis-let-me-lead-you-to-the-solution-step-by-step/>

## **Where to start your thinking**

Always keep in mind: interviewers rarely expect you to invent new algorithms. They almost always test your understanding and skills to apply algorithms you've learned at school, so, what algorithms have you learned at schools that are usually used to solve questions involving an array? DP, search/sort, divide and conquer, greedy.... Hmm... this question reminds me of the question about scheduling meetings with limited meeting rooms, which is solved by greedy algorithm. Even if you don't know the scheduling meeting question, you can swiftly attempt with DP and divide-and-conquer, and will find it is not very straight forward to define the subproblem of DB, or to find the split point of divide-and-conquer. Hmm... so greedy algorithm looks like the right one. Let's try that.

## **Greedy algorithm intuition**

Greedy algorithms are usually very intuitive (but not necessarily correct. it requires proof). What would you do, if you have multiple equally important meetings to run, but can only make some of them? Most people probably would choose to go to the one that is going to end soon. And after that meeting, pick the next meeting from those that are still available.

## **Greedy algorithm proof**

At some day, suppose both events E1 and E2 are available to choose to attend. For contradictory purpose, suppose the event E1 that is going to end sooner is not the best choice for now. Instead, E2 that ends later is the best choice for now. By choosing E2 now, you come up with a schedule S1.

I claim that I can always construct another schedule S2 in which we choose E1 instead of E2 for now, and S2 is not worse than [S1.In](http://S1.In) S1, from now on, if E1 is picked some time after, then I can always swap E1 and E2 in S1, so I construct a S2 which is not worse than [S1.In](http://S1.In) S1, from now on, if E1 is **not** picked some time after, then I can aways replace E2 in S1 with E1, so I construct a S2 which is not worse than S1.

So it is always better (at least not worse) to always choose the event that ends sooner.

## **Greedy algorithm implementation**

As we go through each days to figure out the availability of each events, it is very intuitive to first sort the events by the starting day of the events. Then the question is, how to find out which (still available) event ends the earliest? It seems that we need to sort the **currently available** events according to the ending day of the events. How to do that? Again, the interviewers don't expect you to invent something realy new! What data structures / algorithm have you learned that can efficiently keep track of the biggest value, while you can dynamically add and remove elements? ...... Yes! Binary search/insert and min/max heap! Obviously, heap is more efficient than binary search, because adding/removing an elements after doing binary search can potentionally cause linear time complexity.

import heapq

class Solution(object):

def maxEvents(self, events):

# sort according to start time

events = sorted(events)

total\_days = max(event[1] for event in events)

min\_heap = []

day, cnt, event\_id = 1, 0, 0

while day <= total\_days:

# if no events are available to attend today, let time flies to the next available event.

if event\_id < len(events) and not min\_heap:

day = events[event\_id][0]

# all events starting from today are newly available. add them to the heap.

while event\_id < len(events) and events[event\_id][0] <= day:

heapq.heappush(min\_heap, events[event\_id][1])

event\_id += 1

# if the event at heap top already ended, then discard it.

while min\_heap and min\_heap[0] < day:

heapq.heappop(min\_heap)

# attend the event that will end the earliest

if min\_heap:

heapq.heappop(min\_heap)

cnt += 1

elif event\_id >= len(events):

break # no more events to attend. so stop early to save time.

day += 1

return cnt