<https://www.cnblogs.com/cnoodle/p/12635738.html>

Given the availability time slots arrays slots1 and slots2 of two people and a meeting duration duration, return the earliest time slot that works for both of them and is of duration duration.

If there is no common time slot that satisfies the requirements, return an empty array.

The format of a time slot is an array of two elements [start, end] representing an inclusive time range from start to end.

It is guaranteed that no two availability slots of the same person intersect with each other. That is, for any two time slots [start1, end1] and [start2, end2] of the same person, either start1 > end2 or start2 > end1.

Example 1:

Input: slots1 = [[10,50],[60,120],[140,210]], slots2 = [[0,15],[60,70]], duration = 8

Output: [60,68]

Example 2:

Input: slots1 = [[10,50],[60,120],[140,210]], slots2 = [[0,15],[60,70]], duration = 12

Output: []

Constraints:

* 1 <= slots1.length, slots2.length <= 10^4
* slots1[i].length, slots2[i].length == 2
* slots1[i][0] < slots1[i][1]
* slots2[i][0] < slots2[i][1]
* 0 <= slots1[i][j], slots2[i][j] <= 10^9
* 1 <= duration <= 10^6

**Attempt 1: 2023-12-05**

**Solution 1: Sorting + Interval intersection check + Two Pointers (60 min)**

**Similar to L1272.Remove Interval, we have to check on each interval boundary and move on to next interval with two pointers strategy**

class Solution {

public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {

// Sort the time slots for both people based on the start times

Arrays.sort(slots1, (a, b) -> a[0] - b[0]);

Arrays.sort(slots2, (a, b) -> a[0] - b[0]);

int index1 = 0;

int index2 = 0;

while(index1 < slots1.length && index2 < slots2.length) {

// Calculate the overlap start time if any

int overlapStart = Math.max(slots1[index1][0], slots2[index2][0]);

// Calculate the overlap end time if any

int overlapEnd = Math.min(slots1[index1][1], slots2[index2][1]);

// Check if satisfy more than duration

if(overlapEnd - overlapStart >= duration) {

// If so, return the start time of the meeting and start time

// plus duration directly since we need the first matched one

return Arrays.asList(overlapStart, overlapStart + duration);

}

// Two pointers strategy

// Move to the next slot in the list that has an earlier end time

if(slots1[index1][1] > slots2[index2][1]) {

index2++;

} else {

index1++;

}

}

return new ArrayList<>();

}

}

Time Complexity: O(MlogM + NlogN)

Space Complexity: O(M + N)

**Refer to**

<https://algo.monster/liteproblems/1229>

## Problem Description

The problem is about finding a mutual meeting time slot between two people given their individual schedules and a required meeting duration. Each person's schedule is represented by a list of non-overlapping time slots where a time slot is an array [start, end] showing availability from start to end. The goal is to find the earliest starting time slot that is available in both schedules and lasts at least for the given duration. If there's no such common time slot, we return an empty array.

## Intuition

To solve this problem, we can use the two-pointer technique. Since no individual time slots overlap within a single person's schedule, we can sort both schedules by the starting times of the slots. We then compare the slots from both schedules to find overlapping slots.

We use two pointers i and j to traverse slots1 and slots2 respectively. In each iteration, we find the latest start time by taking the maximum of slots1[i][0] and slots2[j][0] and the earliest end time by taking the minimum of slots1[i][1] and slots2[j][1]. If the overlapping time between the latest start and earliest end is greater than or equal to the required duration, we have found a suitable time slot and return the start and end of this meeting slot.

If the overlap is not sufficient, we move the pointer forward in the list which has the earlier ending time slot, hoping to find a longer slot that might overlap with the other person's next slot. This process continues until we either find a suitable slot or exhaust all available slots in either list.

## Solution Approach

The provided Python solution follows a straightforward two-pointer approach to solve the problem:

* **Sorting the time slots**: Both slots1 and slots2 are sorted based on their starting times. This ensures that we are always considering the earliest available slots first and eliminates the need for checking past time slots. Sorting is crucial as it sets up the structure for the two-pointer technique to work effectively.
* **Two-pointer Technique**: Two pointers, i and j, are used to iterate through slots1 and slots2 respectively. At each step, i refers to the current slot in the first person's schedule, and j refers to the current slot in the second person's schedule.
* **Finding Overlaps**: For the current pair of time slots pointed by i and j, we calculate the overlap by determining the maximum of the two start times and the minimum of the two end times. The variables start and end are used to record these values:

start = max(slots1[i][0], slots2[j][0])

end = min(slots1[i][1], slots2[j][1])

* **Checking Duration**: We then check if the overlapping duration is greater than or equal to the required duration by subtracting start from end:

if end - start >= duration:

If the condition is met, we have found a valid time slot and can return [start, start + duration] as the solution.

* **Advancing the Pointers**: If the overlapping time slot is not long enough, we need to discard the time slot that ends earlier and move forward. This decision is made by comparing the end times of the current time slots pointed by i and j. The pointer corresponding to the slot with the earlier end time is incremented:

if slots1[i][1] < slots2[j][1]:

i += 1

else:

j += 1

This step ensures that we're always trying to find overlap with the nearest possible future slot.

* **Continuation and Termination**: The above steps are continued in a loop until one of the lists is exhausted (i.e., i reaches the end of slots1 or j reaches the end of slots2). If no common time slots with sufficient duration are found by the end of either list, we return an empty array [] as specified.

This algorithm makes efficient use of the sorted structure of the time slots and the two-pointer technique to minimize the number of comparisons and quickly find the first suitable time slot, achieving a time complexity that is linear in the size of the input time slots after the initial sort.

### Example Walkthrough

Let's consider an example to illustrate the solution approach:

* Person A's schedule (slots1): [[10, 50], [60, 120], [140, 210]]
* Person B's schedule (slots2): [[0, 15], [25, 50], [60, 70], [80, 100]]
* Required duration for the meeting: 8 minutes
* **Sorting the time slots**: Both schedules are already sorted based on their starting times, eliminating the need for a sort operation in this example.
* **Two-pointer Technique**: Initialize two pointers, i for slots1 and j for slots2. Initially, i = 0 and j = 0.
* **Finding Overlaps**:

First Comparison:

slots1[i] = [10, 50] and slots2[j] = [0, 15]

The overlap is from 10 (max of 10 and 0) to 15 (min of 50 and 15), which is 5 minutes long. This is not enough for the 8-minute duration.

Move the pointer j forward because slots2[j][1] is less than slots1[i][1].

Second Comparison:

slots1[i] = [10, 50] and slots2[j] = [25, 50]

The overlap is from 25 to 50, which is 25 minutes long and suffices for the 8-minute duration.

We have found a suitable slot, so we can return the result [25, 25 + 8], which is [25, 33].

This result indicates that Person A and Person B can successfully schedule a meeting starting at 25 minutes past the hour and lasting until 33 minutes past the hour.

The method described above is efficient because it continuously seeks the earliest possible meeting time by looking for overlaps in the schedules and moves forward based on the end times of the current slots. As soon as a fitting time slot is found, it returns the result without unnecessary comparisons of later time slots, saving time and computation.

class Solution {

public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {

// Sort the time slots for both people based on the start times

Arrays.sort(slots1, (a, b) -> a[0] - b[0]);

Arrays.sort(slots2, (a, b) -> a[0] - b[0]);

int index1 = 0; // Index for navigating person 1's time slots

int index2 = 0; // Index for navigating person 2's time slots

int len1 = slots1.length; // Total number of slots for person 1

int len2 = slots2.length; // Total number of slots for person 2

// Iterate through both sets of slots

while (index1 < len1 && index2 < len2) {

// Calculate the overlap start time

int overlapStart = Math.max(slots1[index1][0], slots2[index2][0]);

// Calculate the overlap end time

int overlapEnd = Math.min(slots1[index1][1], slots2[index2][1]);

// Check if the overlapping duration is at least the required duration

if (overlapEnd - overlapStart >= duration) {

// If so, return the start time of the meeting and start time plus duration

return Arrays.asList(overlapStart, overlapStart + duration);

}

// Move to the next slot in the list that has an earlier end time

if (slots1[index1][1] < slots2[index2][1]) {

index1++;

} else {

index2++;

}

}

// If no common slot is found that fits the duration, return an empty list

return Collections.emptyList();

}

}

## Time and Space Complexity

### Time Complexity

The given algorithm primarily consists of two parts: sorting the time slots and then iterating through the sorted lists to find a common available duration.

First, we sort both slots1 and slots2. Assuming that slots1 has m intervals and slots2 has n intervals, the time taken to sort these lists using a comparison-based sorting algorithm, like Timsort (Python's default sorting algorithm), is O(mlogm) for slots1 and O(nlogn) for slots2.

After sorting, we have a while loop that runs until we reach the end of one of the slot lists. In the worst case, we'll compare each pair of slots once, which leads to a complexity of O(m + n) since each list is traversed at most once.

The overall time complexity is the sum of the complexities of sorting and iterating through the slot lists, which is O(mlogm + nlogn + m + n). However, the log factor dominates the linear factor in computational complexity analysis for large values. Hence, the time complexity simplifies to O(mlogm + nlogn).

### Space Complexity

The space complexity of the algorithm is determined by the space we use to sort slots1 and slots2. Since the Timsort algorithm can require a certain amount of space for its operation, the space complexity is O(m+n) due to the auxiliary space used in sorting. All other operations use a constant amount of space, and no additional space is used that depends on the input size, so the space complexity remains O(m+n).

**Solution 2: Priority Queue (60 min)**

class Solution {

public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {

// 去掉两人各自无效的interval（interval本身小于会议时间duration的）之后，

// 把两人所有有效的interval加入pq，pq是按interval的开始时间排序的

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

for(int[] slot : slots1) {

if(slot[1] - slot[0] >= duration) {

minPQ.offer(slot);

}

}

for(int[] slot : slots2) {

if(slot[1] - slot[0] >= duration) {

minPQ.offer(slot);

}

}

// 先弹出一个interval，比较这个interval的结束时间[1] 是否大于等于堆顶interval[0] + duration。

// 这个思路利用到了题目中给的最后一句话，同一个人的空闲时间不会出现交叠的情况。那么在从pq弹出的时候，

// 如果有两个intervals能有交集且满足duration，那么说明这两个intervals一定来自不同的人

while(minPQ.size() > 1) {

if(minPQ.poll()[1] - minPQ.peek()[0] >= duration) {

return Arrays.asList(minPQ.peek()[0], minPQ.peek()[0] + duration);

}

}

return new ArrayList<>();

}

}

Time Complexity: O(MlogM + NlogN)

Space Complexity: O(M + N)

**Refer to**

<https://www.cnblogs.com/cnoodle/p/12635738.html>

这个题不需要考虑一些invalid的case诸如interval的开始时间大于interval的结束时间，所以会好处理一些。思路依然是经典的**扫描线**。discussion里面[目前最高票的答案](https://leetcode.com/problems/meeting-scheduler/discuss/408506/JavaPython-3-simple-code-using-PriorityQueueheapq-w-brief-explanation-and-analysis) [4.4.2020] 给的是priority queue的做法。他的思路是去掉两人各自无效的interval（interval本身小于会议时间duration的）之后，把两人所有有效的interval加入pq，pq是按interval的开始时间排序的。先弹出一个interval，比较这个interval的结束时间[1] 是否大于等于堆顶interval[0] + duration。这个思路利用到了题目中给的最后一句话，**同一个人的空闲时间不会出现交叠的情况**。那么在从pq弹出的时候，如果有两个intervals能有交集且满足duration，那么说明这两个intervals一定来自不同的人。

时间O(nlogn) - sort

空间O(n)

Java实现 - pq思路

class Solution {

public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparing(a -> a[0]));

for (int[] s : slots1) {

if (s[1] - s[0] >= duration) {

pq.offer(s);

}

}

for (int[] s : slots2) {

if (s[1] - s[0] >= duration) {

pq.offer(s);

}

}

while (pq.size() > 1) {

if (pq.poll()[1] >= pq.peek()[0] + duration) {

return Arrays.asList(pq.peek()[0], pq.peek()[0] + duration);

}

}

return Arrays.asList();

}

}

**Solution 3: Sweep Line + TreeMap (60 min)**

class Solution {

public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {

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

for(int[] slot : slots1) {

timeline.put(slot[0], timeline.getOrDefault(slot[0], 0) + 1);

timeline.put(slot[1] + 1, timeline.getOrDefault(slot[1] + 1, 0) - 1);

}

for(int[] slot : slots2) {

timeline.put(slot[0], timeline.getOrDefault(slot[0], 0) + 1);

timeline.put(slot[1] + 1, timeline.getOrDefault(slot[1] + 1, 0) - 1);

}

// 'count' is the running presum, timeline.values() are all deltas

int count = 0;

int candidate\_start = -1;

for(Map.Entry<Integer, Integer> e : timeline.entrySet()) {

int prev\_count = count;

count += e.getValue();

// A candidate start timestamp found, record for further loop usage

if(count == 2) {

candidate\_start = e.getKey();

}

// If we already have a candidate start timestamp and in current loop

// (loop based on timestamp, current loop means a timestamp happen after

// previous recorded candidate start timestamp) we find a decline on

// running presum (count), it means two people intersection end timestamp

// happen, so if previous recorded candidate start timestamp plus required

// duration less than current timestamp (decline) means we found a satisfied

// time range {candidate start timestamp, candidate start timestamp + duration}

if(candidate\_start != -1 && count < prev\_count) {

if(e.getKey() - candidate\_start >= duration) {

return Arrays.asList(candidate\_start, candidate\_start + duration);

} else {

candidate\_start = -1;

}

}

}

return new ArrayList<>();

}

}

Time Complexity: O(MlogM + NlogN), TreeMap put() method time complexity is O(logN) in worst case

In a TreeMap, the key/value entries are stored in a Red-Black tree, and in order to find if a key is contained in the tree, you have to traverse it from the root, down some path, until reaching the required key or reaching a leaf.

A tree containing n elements has an O(log n) height, and therefore that's the time it would take to search for a key.

Space Complexity: O(M + N)

**Step by step explain:**

e.g

int[][] slots1 = {{10,50},{60,120},{140,210}};

int[][] slots2 = {{0,15},{25,50},{60,70},{80,100}};

duration = 8

Since {start, end} inclusive, for 'end' need 'end + 1' to mark '-1'

delta (treemap.values())

1 1 -1 1 -2 2 -1 1 -1 -1 1 -1

\*

\* \* \* \* \* \*

0 10 16 25 51 60 71 80 101 121 140 211

\* \* \* \* \* \*

\*

presum (count)

1 2 1 2 0 2 1 2 1 0 1 0

\* \* \* \*

\* \* \* \* \* \* \* \* \*

0 10 16 25 51 60 71 80 101 121 140 211

^ ^

1st 2nd

1st candidate: 10 + 8 > next decline timestamp 16, failed

2nd candidate: 25 + 8 < next decline timestamp 51, satisfy

Only when presum (count) == 2 means slots1 & slots2 (two people)

have intersection the first candidate is 10, we try 10 + 8 = 18

the minimum timestamp can satisfy the duration requirement, but

we find at 16 the presum (count) is 1, which already start decline

means slots1 & slots2 (two people) will not have intersection start

from 16, so first candidate 10 failed, the second candidate is 25,

we try 25 + 8 = 33 the minimum timestamp can satisfy the duration

requirement, and we find next timestamp 51 have presum (count) is

0, which start decline, but since 33 < 51, which means the decline

at 51 won't impact slots1 & slots2 (two people)'s intersection, so

25 is the minimum start timestamp we are looking for, return

{25, 25 + 8 = 33}

**Refer to**

<https://leetcode.com/discuss/study-guide/2166045/line-sweep-algorithms>

Some pointers, we have 2 candidate, so if both have started only then only take there latest time, for example suppose Person is available from [10:00 , 12:00] but person 2 is available from [11:00 , 12:00]after we sweep the line past [11:00] count would be 2 and then we can take [11:00] as candidate start time as this time is common to both person.

Next whenever we encounter an end of time AND we have already noted down the common\_time , check if we met the duration ? if yes return , because this earlies, otherwise reset the common\_time, as now we have to find some common start all over again.

Interval earliestAppropriateDuration(vector<Interval> &slots1, vector<Interval> &slots2, int duration) {

// --- write your code here ---

map<int, int> m;

for(auto& i : slots1){

m[i.start]++;

m[i.end]--;

}

for(auto& i : slots2){

m[i.start]++;

m[i.end]--;

}

int count = 0;

Interval candidate(-1, -1);

for(auto& p : m){

int old = count;

count += p.second;

if(count ==2){

candidate.start = p.first;

}

// If count is decreasing that mean some slot is ending

if(count < old and candidate.start!=-1){

if(p.first - candidate.start >= duration) // we found

{

candidate.end = candidate.start+duration;

return candidate;

}

else{

candidate.start = -1;

}

}

}

return {-1, -1};

}