# 题目

给你一个会议时间安排的数组intervals，每个会议时间都会包括开始和结束的时间intervals[i] = [starti, endi]，为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。

示例 1：

输入：intervals = [[0,30],[5,10],[15,20]]

输出：2

示例 2：

输入：intervals = [[7,10],[2,4]]

输出：1

提示：

1 <= intervals.length <= 104

0 <= starti < endi <= 106

# 分析

## 方法一：贪心/排序+遍历

解题思路

这道题其实就是操作系统课程中，计算确定运行时间的多线程占用资源至少数量的问题，创建一个序列记录每个会议的起止时间，并用1和-1分别标记这个时间是开始还是结束，这样就能确定某个时刻有几个会议室处于工作中，然后进行会议时间升序排序，注意相等时间的优先级按先结束后开始，可以理解成某个会议室的上个会议刚结束，下个会议立马开始。排序完毕进行对每个时间对应的开始（1）和结束（-1）去计算此时会议室使用数量，并取会议时间安排中会议室使用数量最大值。更具体的过程在代码注释中可以详细理解。

代码：

class Solution { // 排序+遍历

public:

struct cmp { // 排序优先级：会议时间升序 > 相等时间先结束后开始

bool operator()( const pair<int, int> & p1, const pair<int, int> & p2 ) {

if (p1.first == p2.first) return p1.second < p2.second; // 会议时间相同，先结束后开始

return p1.first < p2.first; // 按会议时间升序排序

}

};

int minMeetingRooms(vector<vector<int>>& intervals) {

int n = intervals.size();

if (n < 2) return n;

vector<pair<int, int>> times;

for (auto & inter : intervals) {

times.emplace\_back( make\_pair(inter[0], 1) ); // 会议开始，会议室+1

times.emplace\_back( make\_pair(inter[1], -1) ); // 会议结束，会议室-1

}

sort(times.begin(), times.end(), cmp()); // 排序

int works = 0, ans = 1;

for (auto & t : times) {

works += t.second; // 此时开会的会议室数量

ans = max(ans, works); // 记录最多数量

}

return ans;

}

};

时间复杂度O(NlogN)  
 空间复杂度O(N)

## 方法二：差分

思路和心得：

1、典型差分

2、最小堆

堆模拟房间，每个房间记录上一场会议的结束时间

当有新来的会议时，start和现有房间中最早腾出来的比较。能用则腾出来用

腾不出来，新开一个房间

代码：

class Solution

{

public:

int minMeetingRooms(vector<vector<int>>& intervals)

{

vector<int> diff(1000001, 0); //差分数组 差分

for (vector<int> v: intervals)

{

int start = v[0], end = v[1];

diff[start] ++;

diff[end] --;

}

for (int i = 1; i < 1000001; i ++)

diff[i] += diff[i - 1];

return \*max\_element(diff.begin(), diff.end());

}

};

## 方法三：最小堆

class Solution

{

public:

int minMeetingRooms(vector<vector<int>>& intervals)

{ //最小堆 房间一旦开了，就不会拆了。 每个房间会保留结束时间 新来的自己比对

int n = intervals.size();

if (n == 0)

return 0;

priority\_queue<int, vector<int>, greater<int>> minHeap\_room;

sort(intervals.begin(), intervals.end());

minHeap\_room.push(intervals[0][1]);

for (int i = 1; i < n; i ++)

{

if (intervals[i][0] >= minHeap\_room.top()) //最早腾出来的那个房间 可以用的话，就腾出来用

{

minHeap\_room.pop();

minHeap\_room.push(intervals[i][1]);

}

else

{

minHeap\_room.push(intervals[i][1]); //新开一个房间

}

}

return minHeap\_room.size();

}

};

## 方法四：插旗法

插旗法

这道题可以看成是一个简单的插旗问题：进入一个区间的时候将该点坐标对应的值+1，代表插上一面进入的☑，离开时将该点坐标值-1，代表插上一面离开的☑，在同一个点可以同时插进入的或离开的，因为这样并不形成区间重叠。上述思想只需要对数组进行一次遍历并利用map存储坐标和旗子数量，即可完成所有旗子的插入。由于C++里map是按照键的大小升序排列的，因此只需要遍历一次map，从头开始做累加sum，累加过程中最大的sum就是答案。

class Solution {

public:

int minMeetingRooms(vector<vector<int>>& time) {

map<int, int> mp;

for(auto& t : time){

mp[t[0]]++;

mp[t[1]]--;

}

int ans = 0, sum = 0;

for(auto& t : mp){

sum += t.second;

ans = max(ans, sum);

}

return ans;

}

};

## 方法五（推荐）：排序+双指针

要解决“计算所需会议室的最小数量”问题，核心思路是追踪会议的开始和结束时间，统计同一时间内重叠的会议数量，重叠的最大数量即为所需的最小会议室数。

解题思路

1、核心观察：

会议室的需求取决于“同一时间有多少会议正在进行”。当一个会议开始时，若没有空闲会议室（即所有会议室都被其他未结束的会议占用），则需要新增一间会议室；当一个会议结束时，对应的会议室会被释放，可用于后续会议。

2、关键步骤：

- 分离并排序时间点：将所有会议的开始时间和结束时间分别提取并排序。

- 双指针遍历统计重叠：用两个指针分别遍历已排序的开始时间和结束时间，通过比较当前最早的开始时间和最早的结束时间，判断是否需要新增会议室：

若当前开始时间早于结束时间：说明有新会议开始且无会议室释放，需新增会议室，移动开始时间指针。

若当前开始时间晚于或等于结束时间：说明有会议室释放，可复用，移动结束时间指针。

- 记录最大会议室数：遍历过程中，实时更新当前所需会议室数量，并记录最大值。

代码：

class Solution {

public:

int minMeetingRooms(vector<vector<int>>& intervals) {

int n = intervals.size();

if (n == 0) return 0; // 边界：无会议，无需会议室

// 提取所有开始时间和结束时间

vector<int> starts(n), ends(n);

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

starts[i] = intervals[i][0];

ends[i] = intervals[i][1];

}

// 分别对开始时间和结束时间排序

sort(starts.begin(), starts.end());

sort(ends.begin(), ends.end());

int rooms = 0; // 当前所需会议室数量

int max\_rooms = 0; // 最大所需会议室数量

int i = 0, j = 0; // 双指针：i遍历开始时间，j遍历结束时间

// 遍历所有开始时间

while (i < n) {

if (starts[i] < ends[j]) {

// 新会议开始时无会议室释放，需新增

rooms++;

max\_rooms = max(max\_rooms, rooms);

i++;

} else {

// 有会议室释放，复用

rooms--;

j++;

}

}

return max\_rooms;

}

};

代码解释

1、时间点提取与排序：

将每个会议的开始时间和结束时间分别存入starts和ends数组，并排序。排序后，starts[i]是第i早开始的会议时间，ends[j]是第j早结束的会议时间。

2、双指针遍历逻辑：

- 初始时i=0（第一个开始时间），j=0（第一个结束时间），rooms=0（当前会议室数）。

- 若starts[i] < ends[j]：表示最早开始的会议比最早结束的会议早，此时无会议室空闲，需新增一间（rooms++），并更新最大会议室数（max\_rooms），然后移动开始时间指针（i++）。

- 若starts[i] >= ends[j]：表示最早开始的会议在最早结束的会议之后或同时，此时有会议室释放，可复用（rooms--），移动结束时间指针（j++）。

- 重复上述过程，直到所有开始时间都被处理（i == n）。

3、结果：max\_rooms即为整个过程中所需的最小会议室数量。

复杂度分析

- 时间复杂度：O(n log n)，其中n是会议数量。排序两个数组的时间复杂度为O(n log n)，双指针遍历的时间复杂度为O(n)，整体由排序主导。

- 空间复杂度：O(n)，用于存储starts和ends数组。

该方法通过分离时间点并排序，高效统计重叠会议数量，避免了暴力枚举的冗余计算，适合题目中n ≤ 1e4的约束。