# Sweep Line

This series of the problems have the common pattern and common solution. The input is a series of intervals with starting point and end point. We process the overlap of the interval. The idea is is to mark at the start point and end point, sometimes 1 unit after the end point then sort all the points as an ordered series, then process the end points from the smallest to the largest. The code should be very simple.

## 253. Meeting Rooms II

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 1:**

**Input:** [[0, 30],[5, 10],[15, 20]]

**Output:** 2

**Example 2:**

**Input:** [[7,10],[2,4]]

**Output:** 1

/// <summary>

/// Leet code #253. Meeting Rooms II

///

/// 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.

/// For example,

/// Given [[0, 30],[5, 10],[15, 20]],

/// return 2.

/// </summary>

int LeetCode::minMeetingRoomsII(vector<Interval>& intervals)

{

map<int, int> time\_line;

for (size\_t i = 0; i < intervals.size(); i++)

{

time\_line[intervals[i].start]++;

time\_line[intervals[i].end]--;

}

int max\_rooms = 0, rooms = 0;

for (auto time : time\_line)

{

rooms += time.second;

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

}

return max\_rooms;

}

## 554. Brick Wall

Medium

59931FavoriteShare

There is a brick wall in front of you. The wall is rectangular and has several rows of bricks. The bricks have the same height but different width. You want to draw a vertical line from the **top** to the **bottom** and cross the **least** bricks.

The brick wall is represented by a list of rows. Each row is a list of integers representing the width of each brick in this row from left to right.

If your line go through the edge of a brick, then the brick is not considered as crossed. You need to find out how to draw the line to cross the least bricks and return the number of crossed bricks.

**You cannot draw a line just along one of the two vertical edges of the wall, in which case the line will obviously cross no bricks.**

**Example:**

**Input:** [[1,2,2,1],

[3,1,2],

[1,3,2],

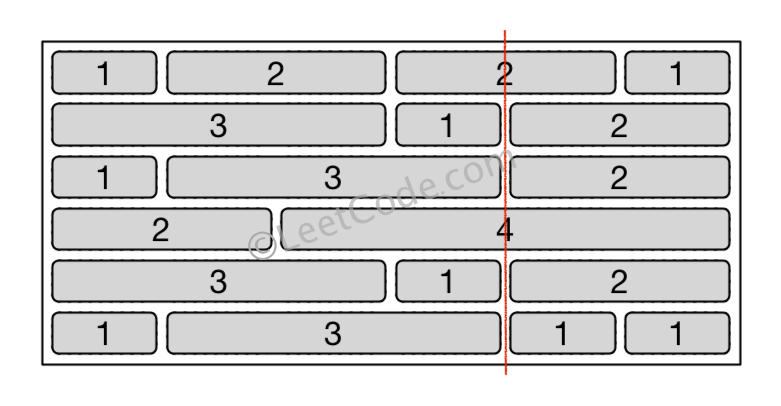
[2,4],

[3,1,2],

[1,3,1,1]]

**Output:** 2

**Explanation:**



**Note:**

1. The width sum of bricks in different rows are the same and won't exceed INT\_MAX.
2. The number of bricks in each row is in range [1,10,000]. The height of wall is in range [1,10,000]. Total number of bricks of the wall won't exceed 20,000.

/// <summary>

/// Leet code #554. Brick Wall

///

/// There is a brick wall in front of you. The wall is rectangular and has

/// several rows of bricks. The bricks have the same height but different

/// width. You want to draw a vertical line from the top to the bottom and

/// cross the least bricks.

/// The brick wall is represented by a list of rows. Each row is a list of

/// integers representing the width of each brick in this row from left to

/// right.

/// If your line go through the edge of a brick, then the brick is not

/// considered as crossed. You need to find out how to draw the line to

/// cross the least bricks and return the number of crossed bricks.

/// You cannot draw a line just along one of the two vertical edges of the

/// wall, in which case the line will obviously cross no bricks.

/// Example:

/// Input:

/// [[1,2,2,1],

/// [3,1,2],

/// [1,3,2],

/// [2,4],

/// [3,1,2],

/// [1,3,1,1]]

/// Output: 2

/// Explanation:

/// Note:

/// The width sum of bricks in different rows are the same and won't exceed

/// INT\_MAX.

/// The number of bricks in each row is in range [1,10,000]. The height of

/// wall is in range [1,10,000]. Total number of bricks of the wall won't

/// exceed 20,000.

/// </summary>

int LeetCode::leastBricks(vector<vector<int>>& wall)

{

unordered\_map<int, int> align\_map;

for (size\_t i = 0; i < wall.size(); i++)

{

int distance = 0;

for (size\_t j = 0; j < wall[i].size() - 1; j++)

{

distance += wall[i][j];

align\_map[distance]++;

}

}

int min\_bricks = wall.size();

for (auto itr : align\_map)

{

min\_bricks = min(min\_bricks, (int)wall.size() - itr.second);

}

return min\_bricks;

}

## 1094. Car Pooling

You are driving a vehicle that has capacity empty seats initially available for passengers.  The vehicle **only** drives east (ie. it **cannot** turn around and drive west.)

Given a list of trips, trip[i] = [num\_passengers, start\_location, end\_location] contains information about the i-th trip: the number of passengers that must be picked up, and the locations to pick them up and drop them off.  The locations are given as the number of kilometers due east from your vehicle's initial location.

Return true if and only if it is possible to pick up and drop off all passengers for all the given trips.

**Example 1:**

**Input:** trips = [[2,1,5],[3,3,7]], capacity = 4

**Output:** false

**Example 2:**

**Input:** trips = [[2,1,5],[3,3,7]], capacity = 5

**Output:** true

**Example 3:**

**Input:** trips = [[2,1,5],[3,5,7]], capacity = 3

**Output:** true

**Example 4:**

**Input:** trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11

**Output:** true

**Constraints:**

1. trips.length <= 1000
2. trips[i].length == 3
3. 1 <= trips[i][0] <= 100
4. 0 <= trips[i][1] < trips[i][2] <= 1000
5. 1 <= capacity <= 100000

/// <summary>

/// Leet code #1094. Car Pooling

///

/// You are driving a vehicle that has capacity empty seats initially

/// available for passengers. The vehicle only drives east (ie. it

/// cannot turn around and drive west.)

///

/// Given a list of trips, trip[i] = [num\_passengers, start\_location,

/// end\_location] contains information about the i-th trip: the number

/// of passengers that must be picked up, and the locations to pick them

/// up and drop them off. The locations are given as the number of

/// kilometers due east from your vehicle's initial location.

///

/// Return true if and only if it is possible to pick up and drop off all

/// passengers for all the given trips.

///

/// Example 1:

/// Input: trips = [[2,1,5],[3,3,7]], capacity = 4

/// Output: false

///

/// Example 2:

/// Input: trips = [[2,1,5],[3,3,7]], capacity = 5

/// Output: true

///

/// Example 3:

/// Input: trips = [[2,1,5],[3,5,7]], capacity = 3

/// Output: true

///

/// Example 4:

/// Input: trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11

/// Output: true

///

/// Constraints:

/// 1. trips.length <= 1000

/// 2. trips[i].length == 3

/// 3. 1 <= trips[i][0] <= 100

/// 4. 0 <= trips[i][1] < trips[i][2] <= 1000

/// 5. 1 <= capacity <= 100000

/// </summary>

bool LeetCode::carPooling(vector<vector<int>>& trips, int capacity)

{

map<int, int> stops;

for (size\_t i = 0; i < trips.size(); i++)

{

stops[trips[i][1]] += trips[i][0];

stops[trips[i][2]] -= trips[i][0];

}

int result = 0;

for (auto itr : stops)

{

result += itr.second;

if (result > capacity) return false;

}

return true;

}

## 1109. Corporate Flight Bookings

Medium

There are n flights, and they are labeled from 1 to n.

We have a list of flight bookings.  The i-th booking bookings[i] = [i, j, k] means that we booked k seats from flights labeled i to jinclusive.

Return an array answer of length n, representing the number of seats booked on each flight in order of their label.

**Example 1:**

**Input:** bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5

**Output:** [10,55,45,25,25]

**Constraints:**

* 1 <= bookings.length <= 20000
* 1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
* 1 <= bookings[i][2] <= 10000

/// <summary>

/// Leet code #1109. Corporate Flight Bookings

///

/// There are n flights, and they are labeled from 1 to n.

/// We have a list of flight bookings. The i-th booking

/// bookings[i] = [i, j, k] means that we booked k seats from flights

/// labeled i to j inclusive.

///

/// Return an array answer of length n, representing the number of seats

/// booked on each flight in order of their label.

///

/// Example 1:

///

/// Input: bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5

/// Output: [10,55,45,25,25]

///

/// Constraints:

/// 1. 1 <= bookings.length <= 20000

/// 2. 1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000

/// 3. 1 <= bookings[i][2] <= 10000

/// </summary>

vector<int> LeetCode::corpFlightBookings(vector<vector<int>>& bookings, int n)

{

vector<int> result(n);

for (size\_t i = 0; i < bookings.size(); i++)

{

result[bookings[i][0] - 1] += bookings[i][2];

if (bookings[i][1] < n)

{

result[bookings[i][1]] -= bookings[i][2];

}

}

for (size\_t i = 0; i < result.size(); i++)

{

if (i > 0) result[i] += result[i - 1];

}

return result;

}

## 218. The Skyline Problem

Hard

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).

[[](https://leetcode.com/static/images/problemset/skyline1.jpg)](https://leetcode.com/static/images/problemset/skyline1.jpg)[](https://leetcode.com/static/images/problemset/skyline2.jpg)

The geometric information of each building is represented by a triplet of integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed that 0 ≤ Li, Ri ≤ INT\_MAX, 0 < Hi ≤ INT\_MAX, and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: [ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ] .

The output is a list of "**key points**" (red dots in Figure B) in the format of [ [x1,y1], [x2, y2], [x3, y3], ... ] that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ].

**Notes:**

* The number of buildings in any input list is guaranteed to be in the range [0, 10000].
* The input list is already sorted in ascending order by the left x position Li.
* The output list must be sorted by the x position.
* There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...[2 3], [4 5], [12 7], ...]

/// <summary>

/// Leet code #218. The Skyline Problem

///

/// A city's skyline is the outer contour of the silhouette formed

/// by all the buildings in that city when viewed from a distance.

/// Now suppose you are given the locations and height of all the buildings

/// as shown on a cityscape photo (Figure A), write a program to output

/// the skyline formed by these buildings collectively (Figure B).

///

/// Buildings Skyline Contour

/// The geometric information of each building is represented by a triplet of

/// integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left

/// and right edge of the ith building, respectively, and Hi is its height.

/// It is guaranteed that 0 ≤ Li, Ri ≤ INT\_MAX, 0 < Hi ≤ INT\_MAX,

/// and Ri - Li > 0.

/// You may assume all buildings are perfect rectangles grounded on an

/// absolutely flat surface at height 0.

///

/// For instance, the dimensions of all buildings in Figure A are recorded as:

/// [ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ] .

/// The output is a list of "key points" (red dots in Figure B) in the format of

/// [ [x1,y1], [x2, y2], [x3, y3], ... ] that uniquely defines a skyline. A key

/// point is the left endpoint of a horizontal line segment. Note that the last

/// key point, where the rightmost building ends, is merely used to mark the

/// termination of the skyline, and always has zero height. Also, the ground in

/// between any two adjacent buildings should be considered part of the skyline

/// contour.

/// For instance, the skyline in Figure B should be represented as:

/// [ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ].

/// Notes:

/// 1. The number of buildings in any input list is guaranteed to be in the

/// range [0, 10000].

/// 2. The input list is already sorted in ascending order by the left x

/// position Li.

/// 3. The output list must be sorted by the x position.

/// 4. There must be no consecutive horizontal lines of equal height in the

/// output skyline.

/// 5. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not

/// acceptable;

/// 6. the three lines of height 5 should be merged into one in the final

/// output as such: [...[2 3], [4 5], [12 7], ...]

/// </summary>

vector<vector<int>> LeetCode::getSkyline(vector<vector<int>>& buildings)

{

vector<vector<int>> result;

map<int, vector<int>> edge\_map;

for (size\_t i = 0; i < buildings.size(); i++)

{

// left side

edge\_map[buildings[i][0]].push\_back(buildings[i][2]);

// right side

edge\_map[buildings[i][1]].push\_back(-buildings[i][2]);

}

// we may have multiple building with same height

map<int, int> skylines;

for (auto edge : edge\_map)

{

for (auto height : edge.second)

{

if (height > 0)

{

skylines[height]++;

}

else

{

skylines[-height]--;

if (skylines[-height] == 0) skylines.erase(-height);

}

}

int skyline = 0;

// skyline is heightest building

if (!skylines.empty())

{

skyline = skylines.rbegin()->first;

}

// push to result if not same height

if (result.empty() || result.back()[1] != skyline)

{

result.push\_back({ edge.first, skyline });

}

}

return result;

}

**There is another case that we can not always scan from beginning, so we need to track the maximum overlap on each endpoint. We can track by interval, however if we want to simplify it we can simply track the layer before the point to reduce interval to points. Remember to set a zero at the far end.**

## 699. Falling Squares

Hard

On an infinite number line (x-axis), we drop given squares in the order they are given.

The i-th square dropped (positions[i] = (left, side\_length)) is a square with the left-most point being positions[i][0] and sidelength positions[i][1].

The square is dropped with the bottom edge parallel to the number line, and from a higher height than all currently landed squares. We wait for each square to stick before dropping the next.

The squares are infinitely sticky on their bottom edge, and will remain fixed to any positive length surface they touch (either the number line or another square). Squares dropped adjacent to each other will not stick together prematurely.

Return a list ans of heights. Each height ans[i] represents the current highest height of any square we have dropped, after dropping squares represented by positions[0], positions[1], ..., positions[i].

**Example 1:**

**Input:** [[1, 2], [2, 3], [6, 1]]

**Output:** [2, 5, 5]

**Explanation:**

After the first drop of positions[0] = [1, 2]: \_aa \_aa ------- The maximum height of any square is 2.

After the second drop of positions[1] = [2, 3]: \_\_aaa \_\_aaa \_\_aaa \_aa\_\_ \_aa\_\_ -------------- The maximum height of any square is 5. The larger square stays on top of the smaller square despite where its center of gravity is, because squares are infinitely sticky on their bottom edge.

After the third drop of positions[1] = [6, 1]: \_\_aaa \_\_aaa \_\_aaa \_aa \_aa\_\_\_a -------------- The maximum height of any square is still 5. Thus, we return an answer of [2, 5, 5].

**Example 2:**

**Input:** [[100, 100], [200, 100]]

**Output:** [100, 100]

**Explanation:** Adjacent squares don't get stuck prematurely - only their bottom edge can stick to surfaces.

**Note:**

* 1 <= positions.length <= 1000.
* 1 <= positions[i][0] <= 10^8.
* 1 <= positions[i][1] <= 10^6.

/// <summary>

/// Leet code #699. Falling Squares

///

/// On an infinite number line (x-axis), we drop given squares in the order

/// they are given.

///

/// The i-th square dropped (positions[i] = (left, side\_length)) is a

/// square with the left-most point being positions[i][0] and sidelength

/// positions[i][1].

///

/// The square is dropped with the bottom edge parallel to the number line,

/// and from a higher height than all currently landed squares. We wait for

/// each square to stick before dropping the next.

///

/// The squares are infinitely sticky on their bottom edge, and will remain

/// fixed to any positive length surface they touch (either the number line

/// or another square). Squares dropped adjacent to each other will not

/// stick together prematurely.

///

/// Return a list ans of heights. Each height ans[i] represents the current

/// highest height of any square we have dropped, after dropping squares

/// represented by positions[0], positions[1], ..., positions[i].

///

/// Example 1:

/// Input: [[1, 2], [2, 3], [6, 1]]

/// Output: [2, 5, 5]

/// Explanation:

///

/// After the first drop of

/// positions[0] = [1, 2]:

/// \_aa

/// \_aa

/// -------

/// The maximum height of any square is 2.

///

/// After the second drop of

/// positions[1] = [2, 3]:

/// \_\_aaa

/// \_\_aaa

/// \_\_aaa

/// \_aa\_\_

/// \_aa\_\_

/// --------------

/// The maximum height of any square is 5.

/// The larger square stays on top of the smaller square despite where its

/// center of gravity is, because squares are infinitely sticky on their

/// bottom edge.

///

/// After the third drop of

/// positions[1] = [6, 1]:

/// \_\_aaa

/// \_\_aaa

/// \_\_aaa

/// \_aa

/// \_aa\_\_\_a

/// --------------

/// The maximum height of any square is still 5.

///

/// Thus, we return an answer of

/// [2, 5, 5]

///

/// Example 2:

/// Input: [[100, 100], [200, 100]]

/// Output: [100, 100]

/// Explanation: Adjacent squares don't get stuck prematurely - only their

/// bottom edge can stick to surfaces.

///

/// Note:

/// 1 <= positions.length <= 1000.

/// 1 <= positions[i][0] <= 10^8.

/// 1 <= positions[i][1] <= 10^6.

/// </summary>

vector<int> LeetCodeGreedy::fallingSquares(vector<vector<int>>& positions)

{

vector<int> result;

// remember the start position and height

map<int, int> pos\_map;

pos\_map[INT\_MAX] = 0;

int max\_height = 0;

for (size\_t i = 0; i < positions.size(); i++)

{

auto itr = pos\_map.lower\_bound(positions[i][0]);

int prev = itr->second;

int height = 0;

while (itr->first < positions[i][0] + positions[i][1])

{

auto temp = itr;

if (itr->first > positions[i][0])

{

height = max(height, itr->second);

}

itr++;

pos\_map.erase(temp);

}

height = max(height, itr->second);

height += positions[i][1];

// set start as prev height

pos\_map[positions[i][0]] = prev;

// set end as maximum height in range

pos\_map[positions[i][0] + positions[i][1]] = height;

// calculate max height

max\_height = max(max\_height, height);

result.push\_back(max\_height);

}

return result;

}

## 715. Range Module

Hard

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

 addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

 queryRange(int left, int right) Returns true if and only if every real number in the interval [left, right) is currently being tracked.

 removeRange(int left, int right) Stops tracking every real number currently being tracked in the interval [left, right).

**Example 1:**

**addRange(10, 20)**: null

**removeRange(14, 16)**: null

**queryRange(10, 14)**: true (Every number in [10, 14) is being tracked)

**queryRange(13, 15)**: false (Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

**queryRange(16, 17)**: true (The number 16 in [16, 17) is still being tracked, despite the remove operation)

**Note:**

 A half open interval [left, right) denotes all real numbers left <= x < right.

 0 < left < right < 10^9 in all calls to addRange, queryRange, removeRange.

 The total number of calls to addRange in a single test case is at most 1000.

 The total number of calls to queryRange in a single test case is at most 5000.

 The total number of calls to removeRange in a single test case is at most 1000.

/// <summary>

/// Leet code #715. Range Module

///

/// A Range Module is a module that tracks ranges of numbers. Your task is

/// to design and implement the following interfaces in an efficient

/// manner.

///

/// addRange(int left, int right) Adds the half-open interval

/// [left, right), tracking every real number in that interval. Adding

/// an interval that partially overlaps with currently tracked numbers

/// should add any numbers in the interval [left, right) that are not

/// already tracked.

/// queryRange(int left, int right) Returns true if and only if every real

/// number in the interval [left, right) is currently being tracked.

/// removeRange(int left, int right) Stops tracking every real number

/// currently being tracked in the interval [left, right).

///

/// Example 1:

/// addRange(10, 20): null

/// removeRange(14, 16): null

/// queryRange(10, 14): true (Every number in [10, 14) is being tracked)

/// queryRange(13, 15): false (Numbers like 14, 14.03, 14.17 in [13, 15)

/// are not being tracked)

/// queryRange(16, 17): true (The number 16 in [16, 17) is still being

/// tracked, despite the remove operation)

/// Note:

///

/// A half open interval [left, right) denotes all real numbers

/// left <= x < right.

/// 0 < left < right < 10^9 in all calls to addRange, queryRange,

/// removeRange.

/// The total number of calls to addRange in a single test case is at most

/// 1000.

/// The total number of calls to queryRange in a single test case is at

/// most 5000.

/// The total number of calls to removeRange in a single test case is at

/// most 1000.

/// </summary>

class RangeModule

{

private:

map<int, int> m\_range;

public:

RangeModule()

{

m\_range[INT\_MAX] = 0;

}

void addRange(int left, int right)

{

auto itr = m\_range.lower\_bound(left);

int prev = itr->second;

while (itr->first < right)

{

int val = itr->first;

itr++;

m\_range.erase(val);

}

m\_range[left] = prev;

m\_range[right] = 1;

}

void removeRange(int left, int right)

{

auto itr = m\_range.lower\_bound(left);

int prev = itr->second;

while (itr->first < right)

{

int val = itr->first;

itr++;

m\_range.erase(val);

}

m\_range[left] = prev;

m\_range[right] = 0;

}

bool queryRange(int left, int right)

{

auto itr = m\_range.upper\_bound(left);

while (itr->first < right)

{

if (itr->second == 0) return false;

itr++;

}

if (itr->second == 0) return false;

else return true;

}

};

## 732. My Calendar III

Hard

405103Add to ListShare

Implement a MyCalendarThree class to store your events. A new event can **always** be added.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A *K-booking* happens when **K** events have some non-empty intersection (ie., there is some time that is common to all K events.)

For each call to the method MyCalendar.book, return an integer K representing the largest integer such that there exists a K-booking in the calendar.

Your class will be called like this: MyCalendarThree cal = new MyCalendarThree(); MyCalendarThree.book(start, end)

**Example 1:**

MyCalendarThree();

MyCalendarThree.book(10, 20); // returns 1

MyCalendarThree.book(50, 60); // returns 1

MyCalendarThree.book(10, 40); // returns 2

MyCalendarThree.book(5, 15); // returns 3

MyCalendarThree.book(5, 10); // returns 3

MyCalendarThree.book(25, 55); // returns 3

**Explanation:**

The first two events can be booked and are disjoint, so the maximum K-booking is a 1-booking.

The third event [10, 40) intersects the first event, and the maximum K-booking is a 2-booking.

The remaining events cause the maximum K-booking to be only a 3-booking.

Note that the last event locally causes a 2-booking, but the answer is still 3 because

eg. [10, 20), [10, 40), and [5, 15) are still triple booked.

**Note:**

* The number of calls to MyCalendarThree.book per test case will be at most 400.
* In calls to MyCalendarThree.book(start, end), start and end are integers in the range [0, 10^9].

/// <summary>

/// Leet code #732. My Calendar III

///

/// Implement a MyCalendarThree class to store your events. A new event can

/// always be added.

///

/// Your class will have one method, book(int start, int end). Formally,

/// this represents a booking on the half open interval [start, end), the

/// range of real numbers x such that start <= x < end.

///

/// A K-booking happens when K events have some non-empty intersection

/// (ie., there is some time that is common to all K events.)

///

/// For each call to the method MyCalendar.book, return an integer K

/// representing the largest integer such that there exists a K-booking

/// in the calendar.

///

/// Your class will be called like this: MyCalendarThree cal = new

/// MyCalendarThree(); MyCalendarThree.book(start, end)

///

/// Example 1:

/// MyCalendarThree();

/// MyCalendarThree.book(10, 20); // returns 1

/// MyCalendarThree.book(50, 60); // returns 1

/// MyCalendarThree.book(10, 40); // returns 2

/// MyCalendarThree.book(5, 15); // returns 3

/// MyCalendarThree.book(5, 10); // returns 3

/// MyCalendarThree.book(25, 55); // returns 3

/// Explanation:

/// The first two events can be booked and are disjoint, so the maximum

/// K-booking is a 1-booking.

/// The third event [10, 40) intersects the first event, and the maximum

/// K-booking is a 2-booking.

/// The remaining events cause the maximum K-booking to be only a

/// 3-booking.

/// Note that the last event locally causes a 2-booking, but the answer

/// is still 3 because eg. [10, 20), [10, 40), and [5, 15) are still

/// triple booked.

/// Note:

///

/// The number of calls to MyCalendarThree.book per test case will be at

/// most 400.

/// In calls to MyCalendarThree.book(start, end), start and end are

/// integers in the range [0, 10^9].

/// </summary>

class MyCalendarThree

{

private:

int m\_book;

map<int, int> m\_timemap;

public:

MyCalendarThree()

{

m\_timemap[INT\_MAX] = 0;

m\_book = 0;

}

int book(int start, int end)

{

auto itr = m\_timemap.lower\_bound(start);

int prev = itr->second;

while (itr->first < end)

{

// We track the layers before the point,

/// so we can not increase start point

if (itr->first > start) itr->second++;

m\_book = max(m\_book, itr->second);

itr++;

}

m\_timemap[start] = prev;

// end point layer is determined by the layer before next point

m\_timemap[end] = itr->second + 1;

// we can not use itr->second again, since it may already be increased

m\_book = max(m\_book, m\_timemap[end]);

return m\_book;

}

};

## 731. My Calendar II

Medium

Implement a MyCalendarTwo class to store your events. A new event can be added if adding the event will not cause a **triple** booking.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A *triple booking* happens when **three** events have some non-empty intersection (ie., there is some time that is common to all 3 events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a **triple** booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

**Example 1:**

MyCalendar();

MyCalendar.book(10, 20); // returns true

MyCalendar.book(50, 60); // returns true

MyCalendar.book(10, 40); // returns true

MyCalendar.book(5, 15); // returns false

MyCalendar.book(5, 10); // returns true

MyCalendar.book(25, 55); // returns true

**Explanation:**

The first two events can be booked. The third event can be double booked.

The fourth event (5, 15) can't be booked, because it would result in a triple booking.

The fifth event (5, 10) can be booked, as it does not use time 10 which is already double booked.

The sixth event (25, 55) can be booked, as the time in [25, 40) will be double booked with the third event;

the time [40, 50) will be single booked, and the time [50, 55) will be double booked with the second event.

**Note:**

* The number of calls to MyCalendar.book per test case will be at most 1000.
* In calls to MyCalendar.book(start, end), start and end are integers in the range [0, 10^9].

/// <summary>

/// Leet code #731. My Calendar II

///

/// Implement a MyCalendarTwo class to store your events. A new event can

/// be added if adding the event will not cause a triple booking.

///

/// Your class will have one method, book(int start, int end). Formally,

/// this represents a booking on the half open interval [start, end), the

/// range of real numbers x such that start <= x < end.

///

/// A triple booking happens when three events have some non-empty

/// intersection (ie., there is some time that is common to all 3 events.)

///

/// For each call to the method MyCalendar.book, return true if the event

/// can be added to the calendar successfully without causing a triple

/// booking. Otherwise, return false and do not add the event to the

/// calendar.

///

/// Your class will be called like this:

/// MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

/// Example 1:

/// MyCalendar();

/// MyCalendar.book(10, 20); // returns true

/// MyCalendar.book(50, 60); // returns true

/// MyCalendar.book(10, 40); // returns true

/// MyCalendar.book(5, 15); // returns false

/// MyCalendar.book(5, 10); // returns true

/// MyCalendar.book(25, 55); // returns true

/// Explanation:

/// The first two events can be booked. The third event can be double

/// booked.

/// The fourth event (5, 15) can't be booked, because it would result

/// in a triple booking.

/// The fifth event (5, 10) can be booked, as it does not use time 10

/// which is already double booked.

/// The sixth event (25, 55) can be booked, as the time in [25, 40) will

/// be double booked with the third event;

/// the time [40, 50) will be single booked, and the time [50, 55) will be

/// double booked with the second event.

/// Note:

///

/// 1. The number of calls to MyCalendar.book per test case will be at most

/// 1000.

/// 2. In calls to MyCalendar.book(start, end), start and end are integers in

/// the range [0, 10^9].

/// </summary>

class MyCalendarTwo {

private:

map<int, int> m\_TimeMap;

// fine the location for the point in space which is less than or equal

// to the input point, the input point may start with the existing value

map<int, int>::iterator getLocation(int time\_stamp)

{

auto itr = m\_TimeMap.lower\_bound(time\_stamp);

if (itr == m\_TimeMap.end() || time\_stamp < itr->first)

{

itr--;

}

return itr;

};

public:

MyCalendarTwo()

{

m\_TimeMap[0] = 0;

}

bool book(int start, int end)

{

auto itr = getLocation(start);

while (itr != m\_TimeMap.end() && itr->first < end)

{

if (itr->second == 2) return false;

itr++;

}

itr = getLocation(end);

m\_TimeMap[end] = itr->second;

itr = getLocation(start);

m\_TimeMap[start] = itr->second + 1;

itr = m\_TimeMap.find(start);

itr++;

while (itr != m\_TimeMap.end() && itr->first < end)

{

itr->second++;

itr++;

}

return true;

}

};