LeetCode Training Day 9 Sweep Line

Sweep Line is an algorithm to resolve the problem of interval overlap. We are given a series of interval and we want to check how many overlaps we got. Even more we want to check what is the maximum level of overlaps in the whole range. This is a problem in the category of greedy.

A normal methodology is to calculate each inerval, add up the overlapped range, but the code will be very long and complicated if we do so. There is a very easy way to handle it. For every interval when we enter, we add 1 at the start point and when we leave the interval, deduct 1 at the end point. In this case at any time the accumulated value is how many overlapped intervals are covering this point.

First, we should assume the endpoint in each interval are stored in order, due to the number can be very large, we store them in a sorted map (map in C++ or SortedDictionary in C#). We process every interval one by one by adding or deducting at endpoint, then finally we scan from the beginning point to end point to calculate the accumulated sum.

## 253. Meeting Rooms II

Medium

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return *the minimum number of conference rooms required*.

**Example 1:**

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

**Output:** 2

**Example 2:**

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

**Output:** 1

**Constraints:**

* 1 <= intervals.length <= 104
* 0 <= starti < endi <= 106

### Analysis:

On every starting point, we add 1 and on every end time, we deduct 1, the sweep from very beginning to accumulate the answer.

/// <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 LeetCodeGreedy::minMeetingRooms(vector<vector<int>>& intervals)

{

map<int, int> time\_line;

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

{

time\_line[intervals[i][0]]++;

time\_line[intervals[i][1]]--;

}

int max\_rooms = 0, rooms = 0;

for (auto &time : time\_line)

{

rooms += time.second;

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

}

return max\_rooms;

}

## 1094. Car Pooling

Medium

There is a car with capacity empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer capacity and an array trips where trip[i] = [numPassengersi, fromi, toi] indicates that the ith trip has numPassengersi passengers and the locations to pick them up and drop them off are fromi and toi respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return true*if it is possible to pick up and drop off all passengers for all the given trips, or*false*otherwise*.

**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
* trips[i].length == 3
* 1 <= numPassengersi <= 100
* 0 <= fromi < toi <= 1000
* 1 <= capacity <= 105

### Analysis:

On every starting stop, we add people and on every end stop, we deduct people, the sweep from very beginning to accumulate the answer.

/// <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 LeetCodeGreedy::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;

}

## 1854. Maximum Population Year

Easy

You are given a 2D integer array logs where each logs[i] = [birthi, deathi] indicates the birth and death years of the ith person.

The **population** of some year x is the number of people alive during that year. The ith person is counted in year x's population if x is in the **inclusive** range [birthi, deathi - 1]. Note that the person is **not** counted in the year that they die.

Return *the****earliest****year with the****maximum population***.

**Example 1:**

**Input:** logs = [[1993,1999],[2000,2010]]

**Output:** 1993

**Explanation:** The maximum population is 1, and 1993 is the earliest year with this population.

**Example 2:**

**Input:** logs = [[1950,1961],[1960,1971],[1970,1981]]

**Output:** 1960

**Explanation:**

The maximum population is 2, and it had happened in years 1960 and 1970.

The earlier year between them is 1960.

**Constraints:**

* 1 <= logs.length <= 100
* 1950 <= birthi < deathi <= 2050

### Analysis:

Add on birth, deduct on death.

/// <summary>

/// Leet code 1854. Maximum Population Year

///

/// Easy

///

/// You are given a 2D integer array logs where each logs[i] =

/// [birthi, deathi] indicates the birth and death years of the ith person.

///

/// The population of some year x is the number of people alive during

/// that year. The ith person is counted in year x's population if x is in

/// the inclusive range [birthi, deathi - 1]. Note that the person is not

/// counted in the year that they die.

///

/// Return the earliest year with the maximum population.

///

/// Example 1:

///

/// Input: logs = [[1993,1999],[2000,2010]]

/// Output: 1993

/// Explanation: The maximum population is 1, and 1993 is the earliest

/// year with this population.

///

/// Example 2:

/// Input: logs = [[1950,1961],[1960,1971],[1970,1981]]

/// Output: 1960

/// Explanation:

/// The maximum population is 2, and it had happened in years 1960 and

/// 1970.

/// The earlier year between them is 1960.

///

/// Constraints:

/// 1. 1 <= logs.length <= 100

/// 2. 1950 <= birthi < deathi <= 2050

/// </summary>

int LeetCodeGreedy::maximumPopulation(vector<vector<int>>& logs)

{

vector<int> arr(102);

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

{

arr[logs[i][0] - 1950]++;

arr[logs[i][1] - 1950]--;

}

int max\_population = 0;

int result = 0;

int population = 0;

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

{

population += arr[i];

if (population > max\_population)

{

max\_population = population;

result = i + 1950;

}

}

return result;

}

## 1893. Check if All the Integers in a Range Are Covered

Easy

You are given a 2D integer array ranges and two integers left and right. Each ranges[i] = [starti, endi] represents an **inclusive** interval between starti and endi.

Return true *if each integer in the inclusive range* [left, right] *is covered by****at least one****interval in* ranges. Return false *otherwise*.

An integer x is covered by an interval ranges[i] = [starti, endi] if starti <= x <= endi.

**Example 1:**

**Input:** ranges = [[1,2],[3,4],[5,6]], left = 2, right = 5

**Output:** true

**Explanation:** Every integer between 2 and 5 is covered:

- 2 is covered by the first range.

- 3 and 4 are covered by the second range.

- 5 is covered by the third range.

**Example 2:**

**Input:** ranges = [[1,10],[10,20]], left = 21, right = 21

**Output:** false

**Explanation:** 21 is not covered by any range.

**Constraints:**

* 1 <= ranges.length <= 50
* 1 <= starti <= endi <= 50
* 1 <= left <= right <= 50

### Analysis:

Add on left point, deduct on right point + 1, check on the way when accumulated is zero.

/// <summary>

/// Leet code 1893. Check if All the Integers in a Range Are Covered

///

/// Easy

///

/// You are given a 2D integer array ranges and two integers left and

/// right. Each ranges[i] = [starti, endi] represents an inclusive

/// interval between starti and endi.

///

/// Return true if each integer in the inclusive range [left, right]

/// is covered by at least one interval in ranges. Return false otherwise.

///

/// An integer x is covered by an interval ranges[i] = [starti, endi]

/// if starti <= x <= endi.

///

///

/// Example 1:

///

/// Input: ranges = [[1,2],[3,4],[5,6]], left = 2, right = 5

/// Output: true

/// Explanation: Every integer between 2 and 5 is covered:

/// - 2 is covered by the first range.

/// - 3 and 4 are covered by the second range.

/// - 5 is covered by the third range.

///

/// Example 2:

/// Input: ranges = [[1,10],[10,20]], left = 21, right = 21

/// Output: false

/// Explanation: 21 is not covered by any range.

/// Constraints:

/// 1. 1 <= ranges.length <= 50

/// 2. 1 <= starti <= endi <= 50

/// 3. 1 <= left <= right <= 50

/// </summary>

bool LeetCodeGreedy::isCovered(vector<vector<int>>& ranges, int left, int right)

{

map<int, int> sweep;

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

{

sweep[ranges[i][0]]++;

sweep[ranges[i][1] + 1]--;

}

if (sweep.begin()->first > left || sweep.rbegin()->first < right)

{

return false;

}

int result = 0;

for (auto itr : sweep)

{

result += itr.second;

if (result == 0 && itr.first >= left && itr.first <= right)

{

return false;

}

}

return true;

}

## 1229. Meeting Scheduler

Medium

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 <= 104
* 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] <= 109
* 1 <= duration <= 106

### Analysis:

On every starting point, we add 1, on every end point, we deduct 1, when we see two people, we start to measure time, if more than duration, we add to answers.

/// <summary>

/// Leet code #1229. Meeting Scheduler

///

/// 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. 1 <= slots1.length, slots2.length <= 10^4

/// 2. slots1[i].length, slots2[i].length == 2

/// 3. slots1[i][0] < slots1[i][1]

/// 4. slots2[i][0] < slots2[i][1]

/// 5. 0 <= slots1[i][j], slots2[i][j] <= 10^9

/// 6. 1 <= duration <= 10^6

/// </summary>

vector<int> LeetCodeGreedy::minAvailableDuration(vector<vector<int>>& slots1,

vector<vector<int>>& slots2, int duration)

{

vector<int> result;

int prev = -1;

map<int, int> time\_line;

for (size\_t i = 0; i < slots1.size(); i++)

{

time\_line[slots1[i][0]]++;

time\_line[slots1[i][1]]--;

}

for (size\_t i = 0; i < slots2.size(); i++)

{

time\_line[slots2[i][0]]++;

time\_line[slots2[i][1]]--;

}

int count = 0;

for (auto &itr : time\_line)

{

count += itr.second;

if (count == 2)

{

prev = itr.first;

}

else

{

if (prev != -1)

{

int time = itr.first - prev;

if (time >= duration)

{

result.push\_back(prev);

result.push\_back(prev + duration);

break;

}

}

prev = -1;

}

}

return result;

}

## 1943. Describe the Painting

Medium

There is a long and thin painting that can be represented by a number line. The painting was painted with multiple overlapping segments where each segment was painted with a **unique** color. You are given a 2D integer array segments, where segments[i] = [starti, endi, colori] represents the **half-closed segment** [starti, endi) with colori as the color.

The colors in the overlapping segments of the painting were **mixed** when it was painted. When two or more colors mix, they form a new color that can be represented as a **set** of mixed colors.

* For example, if colors 2, 4, and 6 are mixed, then the resulting mixed color is {2,4,6}.

For the sake of simplicity, you should only output the **sum** of the elements in the set rather than the full set.

You want to **describe** the painting with the **minimum** number of non-overlapping **half-closed segments** of these mixed colors. These segments can be represented by the 2D array painting where painting[j] = [leftj, rightj, mixj] describes a **half-closed segment** [leftj, rightj) with the mixed color **sum** of mixj.

* For example, the painting created with segments = [[1,4,5],[1,7,7]] can be described by painting = [[1,4,12],[4,7,7]] because:
  + [1,4) is colored {5,7} (with a sum of 12) from both the first and second segments.
  + [4,7) is colored {7} from only the second segment.

Return *the 2D array*painting*describing the finished painting (excluding any parts that are****not****painted). You may return the segments in****any order***.

A **half-closed segment** [a, b) is the section of the number line between points a and b **including** point a and **not including** point b.

**Example 1:**

Chart, box and whisker chart

Description automatically generated

**Input:** segments = [[1,4,5],[4,7,7],[1,7,9]]

**Output:** [[1,4,14],[4,7,16]]

**Explanation:** The painting can be described as follows:

- [1,4) is colored {5,9} (with a sum of 14) from the first and third segments.

- [4,7) is colored {7,9} (with a sum of 16) from the second and third segments.

**Example 2:**

Chart, box and whisker chart

Description automatically generated

**Input:** segments = [[1,7,9],[6,8,15],[8,10,7]]

**Output:** [[1,6,9],[6,7,24],[7,8,15],[8,10,7]]

**Explanation:** The painting can be described as follows:

- [1,6) is colored 9 from the first segment.

- [6,7) is colored {9,15} (with a sum of 24) from the first and second segments.

- [7,8) is colored 15 from the second segment.

- [8,10) is colored 7 from the third segment.

**Example 3:**

Chart, bar chart

Description automatically generated

**Input:** segments = [[1,4,5],[1,4,7],[4,7,1],[4,7,11]]

**Output:** [[1,4,12],[4,7,12]]

**Explanation:** The painting can be described as follows:

- [1,4) is colored {5,7} (with a sum of 12) from the first and second segments.

- [4,7) is colored {1,11} (with a sum of 12) from the third and fourth segments.

Note that returning a single segment [1,7) is incorrect because the mixed color sets are different.

**Constraints:**

* 1 <= segments.length <= 2 \* 104
* segments[i].length == 3
* 1 <= starti < endi <= 105
* 1 <= colori <= 109
* Each colori is distinct.

### Analysis:

Add color on left point and deduct on right point. Sweep from beginning to end, skip zero.

/// <summary>

/// Leet 1943. Describe the Painting

///

/// Medium

///

/// There is a long and thin painting that can be represented by a number

/// line. The painting was painted with multiple overlapping segments

/// where each segment was painted with a unique color. You are given a 2D

/// integer array segments, where segments[i] = [starti, endi, colori]

/// represents the half-closed segment [starti, endi) with colori as the

/// color.

///

/// The colors in the overlapping segments of the painting were mixed when

/// it was painted. When two or more colors mix, they form a new color

/// that can be represented as a set of mixed colors.

///

/// For example, if colors 2, 4, and 6 are mixed, then the resulting mixed

/// color is {2,4,6}.

/// For the sake of simplicity, you should only output the sum of the

/// elements in the set rather than the full set.

///

/// You want to describe the painting with the minimum number of

/// non-overlapping half-closed segments of these mixed colors. These

/// segments can be represented by the 2D array painting where

/// painting[j] = [leftj, rightj, mixj] describes a half-closed segment

/// [leftj, rightj) with the mixed color sum of mixj.

///

/// For example, the painting created with segments = [[1,4,5],[1,7,7]]

/// can be described by painting = [[1,4,12],[4,7,7]] because:

/// [1,4) is colored {5,7} (with a sum of 12) from both the first and

/// second segments.

/// [4,7) is colored {7} from only the second segment.

/// Return the 2D array painting describing the finished painting

/// (excluding any parts that are not painted). You may return the

/// segments in any order.

///

/// A half-closed segment [a, b) is the section of the number line between

/// points a and b including point a and not including point b.

///

/// Example 1:

/// Input: segments = [[1,4,5],[4,7,7],[1,7,9]]

/// Output: [[1,4,14],[4,7,16]]

/// Explanation: The painting can be described as follows:

/// - [1,4) is colored {5,9} (with a sum of 14) from the first and third

/// segments.

/// - [4,7) is colored {7,9} (with a sum of 16) from the second and third

/// segments.

///

/// Example 2:

/// Input: segments = [[1,7,9],[6,8,15],[8,10,7]]

/// Output: [[1,6,9],[6,7,24],[7,8,15],[8,10,7]]

/// Explanation: The painting can be described as follows:

/// - [1,6) is colored 9 from the first segment.

/// - [6,7) is colored {9,15} (with a sum of 24) from the first and second

/// segments.

/// - [7,8) is colored 15 from the second segment.

/// - [8,10) is colored 7 from the third segment.

///

/// Example 3:

/// Input: segments = [[1,4,5],[1,4,7],[4,7,1],[4,7,11]]

/// Output: [[1,4,12],[4,7,12]]

/// Explanation: The painting can be described as follows:

/// - [1,4) is colored {5,7} (with a sum of 12) from the first and second

/// segments.

/// - [4,7) is colored {1,11} (with a sum of 12) from the third and fourth

/// segments.

/// Note that returning a single segment [1,7) is incorrect because the

/// mixed color sets are different.

///

/// Constraints:

/// 1. 1 <= segments.length <= 2 \* 10^4

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

/// 3. 1 <= starti < endi <= 10^5

/// 4. 1 <= colori <= 10^9

/// 5. Each colori is distinct.

/// </summary>

vector<vector<long long>> LeetCodeGreedy::splitPainting(vector<vector<int>>& segments)

{

map<long long, long long> colors;

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

{

colors[segments[i][0]] += segments[i][2];

colors[segments[i][1]] -= segments[i][2];

}

vector<vector<long long>> result;

vector<pair<long long, long long>> paints;

for (auto itr = colors.begin(); itr != colors.end(); ++itr)

{

auto next\_itr = next(itr);

if (next\_itr == colors.end()) break;

next\_itr->second += itr->second;

if (itr->second == 0) continue;

result.push\_back({ itr->first, next\_itr->first, itr->second });

}

return result;

}

## 2021. Brightest Position on Street

Medium

A perfectly straight street is represented by a number line. The street has street lamp(s) on it and is represented by a 2D integer array lights. Each lights[i] = [positioni, rangei] indicates that there is a street lamp at position positioni that lights up the area from [positioni - rangei, positioni + rangei] (**inclusive**).

The **brightness** of a position p is defined as the number of street lamp that light up the position p.

Given lights, return *the****brightest****position on the street. If there are multiple brightest positions, return the****smallest****one.*

**Example 1:**

Chart

Description automatically generated

**Input:** lights = [[-3,2],[1,2],[3,3]]

**Output:** -1

**Explanation:**

The first street lamp lights up the area from [(-3) - 2, (-3) + 2] = [-5, -1].

The second street lamp lights up the area from [1 - 2, 1 + 2] = [-1, 3].

The third street lamp lights up the area from [3 - 3, 3 + 3] = [0, 6].

Position -1 has a brightness of 2, illuminated by the first and second street light.

Positions 0, 1, 2, and 3 have a brightness of 2, illuminated by the second and third street light.

Out of all these positions, -1 is the smallest, so return it.

**Example 2:**

**Input:** lights = [[1,0],[0,1]]

**Output:** 1

**Explanation:**

The first street lamp lights up the area from [1 - 0, 1 + 0] = [1, 1].

The second street lamp lights up the area from [0 - 1, 0 + 1] = [-1, 1].

Position 1 has a brightness of 2, illuminated by the first and second street light.

Return 1 because it is the brightest position on the street.

**Example 3:**

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

**Output:** -1

**Explanation:**

The first street lamp lights up the area from [1 - 2, 1 + 2] = [-1, 3].

Positions -1, 0, 1, 2, and 3 have a brightness of 1, illuminated by the first street light.

Out of all these positions, -1 is the smallest, so return it.

**Constraints:**

* 1 <= lights.length <= 105
* lights[i].length == 2
* -108 <= positioni <= 108
* 0 <= rangei <= 108

### Analysis:

Light on at left edge, light off at right edge + 1.

/// <summary>

/// Leet Code 2021. Brightest Position on Street

///

/// Medium

///

/// A perfectly straight street is represented by a number line. The

/// street has street lamp(s) on it and is represented by a 2D integer

/// array lights. Each lights[i] = [positioni, rangei] indicates that

/// there is a street lamp at position positioni that lights up the

/// area from [positioni - rangei, positioni + rangei] (inclusive).

///

/// The brightness of a position p is defined as the number of street

/// lamp that light up the position p.

///

/// Given lights, return the brightest position on the street. If

/// there are multiple brightest positions, return the smallest one.

///

/// Example 1:

/// Input: lights = [[-3,2],[1,2],[3,3]]

/// Output: -1

/// Explanation:

/// The first street lamp lights up the area from

/// [(-3) - 2, (-3) + 2] = [-5, -1].

/// The second street lamp lights up the area from

/// [1 - 2, 1 + 2] = [-1, 3].

/// The third street lamp lights up the area from

/// [3 - 3, 3 + 3] = [0, 6].

///

/// Position -1 has a brightness of 2, illuminated by the first and second

/// street light.

/// Positions 0, 1, 2, and 3 have a brightness of 2, illuminated by the

/// second and third street light.

/// Out of all these positions, -1 is the smallest, so return it.

///

/// Example 2:

/// Input: lights = [[1,0],[0,1]]

/// Output: 1

/// Explanation:

/// The first street lamp lights up the area from [1 - 0, 1 + 0] = [1, 1].

/// The second street lamp lights up the area from

/// [0 - 1, 0 + 1] = [-1, 1].

///

/// Position 1 has a brightness of 2, illuminated by the first and second

/// street light.

/// Return 1 because it is the brightest position on the street.

///

/// Example 3:

/// Input: lights = [[1,2]]

/// Output: -1

/// Explanation:

/// The first street lamp lights up the area from [1 - 2, 1 + 2] = [-1, 3].

///

/// Positions -1, 0, 1, 2, and 3 have a brightness of 1, illuminated by

/// the first street light.

/// Out of all these positions, -1 is the smallest, so return it.

///

/// Constraints:

/// 1. 1 <= lights.length <= 10^5

/// 2. lights[i].length == 2

/// 3. -10^8 <= positioni <= 10^8

/// 4. 0 <= rangei <= 10^8

/// </summary>

int LeetCodeGreedy::brightestPosition(vector<vector<int>>& lights)

{

map<int, int> street;

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

{

int left = lights[i][0] - lights[i][1];

int right = lights[i][0] + lights[i][1] + 1;

street[left]++;

street[right]--;

}

int result = street.begin()->first;

int prev = street.begin()->second;

int sum = 0;

for (auto& itr : street)

{

sum += itr.second;

if (sum > prev)

{

result = itr.first;

prev = sum;

}

}

return result;

}

## 554. Brick Wall

Medium

There is a rectangular brick wall in front of you with n rows of bricks. The ith row has some number of bricks each of the same height (i.e., one unit) but they can be of different widths. The total width of each row is the same.

Draw a vertical line from the top to the bottom and cross the least bricks. If your line goes through the edge of a brick, then the brick is not considered as crossed. 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.

Given the 2D array wall that contains the information about the wall, return *the minimum number of crossed bricks after drawing such a vertical line*.

**Example 1:**

A picture containing text

Description automatically generated

**Input:** wall = [[1,2,2,1],[3,1,2],[1,3,2],[2,4],[3,1,2],[1,3,1,1]]

**Output:** 2

**Example 2:**

**Input:** wall = [[1],[1],[1]]

**Output:** 3

**Constraints:**

* n == wall.length
* 1 <= n <= 104
* 1 <= wall[i].length <= 104
* 1 <= sum(wall[i].length) <= 2 \* 104
* sum(wall[i]) is the same for each row i.
* 1 <= wall[i][j] <= 231 - 1

### Analysis:

For this problem you do not need to sweep line, just record all edge point by count them in hash table, the maximum edge point meet, is the place you should cut.

/// <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 LeetCodeGreedy::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;

}

# Advanced Problems

## 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. Given the locations and heights of all the buildings, return *the****skyline****formed by these buildings collectively*.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

* lefti is the x coordinate of the left edge of the ith building.
* righti is the x coordinate of the right edge of the ith building.
* heighti is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

**Note:** 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],...]

**Example 1:**

Chart

Description automatically generated

**Input:** buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

**Output:** [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]

**Explanation:**

Figure A shows the buildings of the input.

Figure B shows the skyline formed by those buildings. The red points in figure B represent the key points in the output list.

**Example 2:**

**Input:** buildings = [[0,2,3],[2,5,3]]

**Output:** [[0,3],[5,0]]

**Constraints:**

* 1 <= buildings.length <= 104
* 0 <= lefti < righti <= 231 - 1
* 1 <= heighti <= 231 - 1
* buildings is sorted by lefti in non-decreasing order.

### Analysis:

On every left edge of the building you add this building, on every right edge of the building you take it out. If you see same level as before, skip it, if you see ground, skip it.

/// <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>> LeetCodeGreedy::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;

}

## 1589. Maximum Sum Obtained of Any Permutation

Medium

We have an array of integers, nums, and an array of requests where requests[i] = [starti, endi]. The ith request asks for the sum of nums[starti] + nums[starti + 1] + ... + nums[endi - 1] + nums[endi]. Both starti and endi are *0-indexed*.

Return *the maximum total sum of all requests****among all permutations****of* nums.

Since the answer may be too large, return it **modulo** 109 + 7.

**Example 1:**

**Input:** nums = [1,2,3,4,5], requests = [[1,3],[0,1]]

**Output:** 19

**Explanation:** One permutation of nums is [2,1,3,4,5] with the following result:

requests[0] -> nums[1] + nums[2] + nums[3] = 1 + 3 + 4 = 8

requests[1] -> nums[0] + nums[1] = 2 + 1 = 3

Total sum: 8 + 3 = 11.

A permutation with a higher total sum is [3,5,4,2,1] with the following result:

requests[0] -> nums[1] + nums[2] + nums[3] = 5 + 4 + 2 = 11

requests[1] -> nums[0] + nums[1] = 3 + 5 = 8

Total sum: 11 + 8 = 19, which is the best that you can do.

**Example 2:**

**Input:** nums = [1,2,3,4,5,6], requests = [[0,1]]

**Output:** 11

**Explanation:** A permutation with the max total sum is [6,5,4,3,2,1] with request sums [11].

**Example 3:**

**Input:** nums = [1,2,3,4,5,10], requests = [[0,2],[1,3],[1,1]]

**Output:** 47

**Explanation:** A permutation with the max total sum is [4,10,5,3,2,1] with request sums [19,18,10].

**Constraints:**

* n == nums.length
* 1 <= n <= 105
* 0 <= nums[i] <= 105
* 1 <= requests.length <= 105
* requests[i].length == 2
* 0 <= starti <= endi < n

### Analysis:

This is a very tricky problem, you need to process from the request, not the numbers, given overlap of request, you can see which index is most overlapped, give the maximum number to it then process in desc order of overlap layers.

/// <summary>

/// Leet code #1589. Maximum Sum Obtained of Any Permutation

///

/// Medium

///

/// We have an array of integers, nums, and an array of requests where

/// requests[i] = [starti, endi]. The ith request asks for the sum of

/// nums[starti] + nums[start[i] + 1] + ... + nums[end[i] - 1] +

/// nums[end[]i]. Both start[i] and end[i] are 0-indexed.

///

/// Return the maximum total sum of all requests among all

/// permutations of nums.

///

/// Since the answer may be too large, return it modulo 10^9 + 7.

/// Example 1:

//

/// Input: nums = [1,2,3,4,5], requests = [[1,3],[0,1]]

/// Output: 19

/// Explanation: One permutation of nums is [2,1,3,4,5] with the following

/// result:

/// requests[0] -> nums[1] + nums[2] + nums[3] = 1 + 3 + 4 = 8

/// requests[1] -> nums[0] + nums[1] = 2 + 1 = 3

/// Total sum: 8 + 3 = 11.

/// A permutation with a higher total sum is [3,5,4,2,1] with the

/// following result:

/// requests[0] -> nums[1] + nums[2] + nums[3] = 5 + 4 + 2 = 11

/// requests[1] -> nums[0] + nums[1] = 3 + 5 = 8

/// Total sum: 11 + 8 = 19, which is the best that you can do.

///

/// Example 2:

/// Input: nums = [1,2,3,4,5,6], requests = [[0,1]]

/// Output: 11

/// Explanation: A permutation with the max total sum is [6,5,4,3,2,1]

/// with request sums [11].

///

/// Example 3:

/// Input: nums = [1,2,3,4,5,10], requests = [[0,2],[1,3],[1,1]]

/// Output: 47

/// Explanation: A permutation with the max total sum is [4,10,5,3,2,1]

/// with request sums [19,18,10].

///

/// Constraints:

/// 1. n == nums.length

/// 2. 1 <= n <= 10^5

/// 3. 0 <= nums[i] <= 10^5

/// 4. 1 <= requests.length <= 10^5

/// 5. requests[i].length == 2

/// 6. 0 <= start[i] <= end[i] < n

/// </summary>

int LeetCodeGreedy::maxSumRangeQuery(vector<int>& nums, vector<vector<int>>& requests)

{

int n = nums.size();

vector<int> dp(n);

priority\_queue<pair<int, int>> positions;

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

{

int start = requests[i][0];

int end = requests[i][1];

dp[start]++;

if (end + 1 < n)

{

dp[end + 1] --;

}

}

for (size\_t i = 1; i < dp.size(); i++)

{

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

if (dp[i] != dp[i - 1] && dp[i] > 0)

{

positions.push(make\_pair(dp[i], i));

}

}

if (dp[0] != 0) positions.push(make\_pair(dp[0], 0));

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

int M = 1000000007;

int result = 0;

int index = n - 1;

while (!positions.empty())

{

pair<int, int> pos = positions.top();

positions.pop();

for (int i = pos.second; i < n; i++)

{

if (dp[i] == pos.first)

{

result = (result + nums[index] \* pos.first) % M;

index--;

}

else

{

break;

}

}

}

return result;

}