Problem 2015: Average Height of Buildings in Each Segment

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

A perfectly straight street is represented by a number line. The street has building(s) on it and is represented by a 2D integer array

buildings

, where

buildings[i] = [start

i

, end

i

, height

i

]

. This means that there is a building with

height

i

in the

half-closed segment

[start

i

, end

i

)

.

You want to

describe

the heights of the buildings on the street with the

minimum

number of non-overlapping

segments

. The street can be represented by the 2D integer array

street

where

street[j] = [left

j

, right

j

, average

j

]

describes a

half-closed segment

[left

j

, right

j

)

of the road where the

average

heights of the buildings in the

segment

is

average

j

.

For example, if

buildings = [[1,5,2],[3,10,4]],

the street could be represented by

street = [[1,3,2],[3,5,3],[5,10,4]]

because:

From 1 to 3, there is only the first building with an average height of

2 / 1 = 2

.

From 3 to 5, both the first and the second building are there with an average height of

(2+4) / 2 = 3

.

From 5 to 10, there is only the second building with an average height of

4 / 1 = 4

.

Given

buildings

, return

the 2D integer array

street

as described above (

excluding

any areas of the street where there are no buldings). You may return the array in

any order

.

The

average

of

n

elements is the

sum

of the

n

elements divided (

integer division

) by

n

.

A

half-closed segment

[a, b)

is the section of the number line between points

a

and

b

including

point

a

and

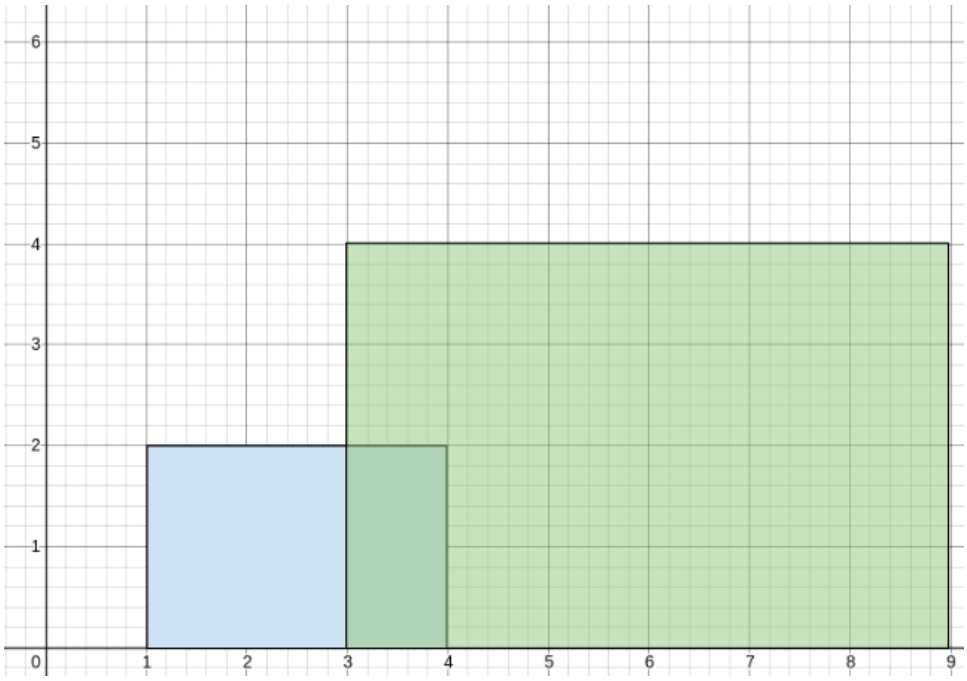
not including

point

b

.

Example 1:



Input:

buildings = [[1,4,2],[3,9,4]]

Output:

[[1,3,2],[3,4,3],[4,9,4]]

Explanation:

From 1 to 3, there is only the first building with an average height of 2 / 1 = 2. From 3 to 4, both the first and the second building are there with an average height of (2+4) / 2 = 3. From 4 to 9, there is only the second building with an average height of 4 / 1 = 4.

Example 2:

Input:

buildings = [[1,3,2],[2,5,3],[2,8,3]]

Output:

[[1,3,2],[3,8,3]]

Explanation:

From 1 to 2, there is only the first building with an average height of 2 / 1 = 2. From 2 to 3, all three buildings are there with an average height of (2+3+3) / 3 = 2. From 3 to 5, both the second and the third building are there with an average height of (3+3) / 2 = 3. From 5 to 8, there is only the last building with an average height of 3 / 1 = 3. The average height from 1 to 3 is the same so we can group them into one segment. The average height from 3 to 8 is the same so we can group them into one segment.

Example 3:

Input:

buildings = [[1,2,1],[5,6,1]]

Output:

[[1,2,1],[5,6,1]]

Explanation:

From 1 to 2, there is only the first building with an average height of 1 / 1 = 1. From 2 to 5, there are no buildings, so it is not included in the output. From 5 to 6, there is only the second building with an average height of 1 / 1 = 1. We cannot group the segments together because an empty space with no buildings seperates the segments.

Constraints:

1 <= buildings.length <= 10

5

buildings[i].length == 3

0 <= start

i

< end

i

<= 10

8

1 <= height

i

<= 10

5

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> averageHeightOfBuildings(vector<vector<int>>& buildings)
{

}
};
```

Java:

```
class Solution {
public int[][] averageHeightOfBuildings(int[][] buildings) {

}
}
```

Python3:

```
class Solution:
def averageHeightOfBuildings(self, buildings: List[List[int]]) ->
List[List[int]]:
```

Python:

```
class Solution(object):
def averageHeightOfBuildings(self, buildings):
"""
:type buildings: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} buildings
* @return {number[][]}
*/
var averageHeightOfBuildings = function(buildings) {

};
```

TypeScript:

```
function averageHeightOfBuildings(buildings: number[][]): number[][] {

};
```

C#:

```
public class Solution {
public int[][] AverageHeightOfBuildings(int[][] buildings) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** averageHeightOfBuildings(int** buildings, int buildingsSize, int*
buildingsColSize, int* returnSize, int** returnColumnSizes) {
```

```
}
```

Go:

```
func averageHeightOfBuildings(buildings [][]int) [][]int {

}
```

Kotlin:

```
class Solution {
fun averageHeightOfBuildings(buildings: Array<IntArray>): Array<IntArray> {

}
}
```

Swift:

```
class Solution {
func averageHeightOfBuildings(_ buildings: [[Int]]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn average_height_of_buildings(buildings: Vec<Vec<i32>>) -> Vec<Vec<i32>>
{

}
}
```

Ruby:

```
# @param {Integer[][]} buildings
# @return {Integer[][]}
def average_height_of_buildings(buildings)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $buildings
* @return Integer[][]
*/
function averageHeightOfBuildings($buildings) {

}
}
```

Dart:

```
class Solution {
List<List<int>> averageHeightOfBuildings(List<List<int>> buildings) {

}
}
```

Scala:

```
object Solution {
def averageHeightOfBuildings(buildings: Array[Array[Int]]): Array[Array[Int]]
= {

}
}
```

Elixir:

```
defmodule Solution do
@spec average_height_of_buildings(buildings :: [[integer]]) :: [[integer]]
def average_height_of_buildings(buildings) do

end
end
```

Erlang:

```
-spec average_height_of_buildings(Buildings :: [[integer()]]) ->
[[integer()]].
```

```
average_height_of_buildings(Buildings) ->
.
```

Racket:

```
(define/contract (average-height-of-buildings buildings)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<vector<int>> averageHeightOfBuildings(vector<vector<int>>& buildings)
{

}
};
```

Java Solution:

```
/**
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public int[][] averageHeightOfBuildings(int[][] buildings) {

}
}
```

Python3 Solution:

```
"""
Problem: Average Height of Buildings in Each Segment
Difficulty: Medium
Tags: array, tree, greedy, sort, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def averageHeightOfBuildings(self, buildings: List[List[int]]) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def averageHeightOfBuildings(self, buildings):
"""
:type buildings: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
```

```
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number[][]} buildings
* @return {number[][]}
*/
var averageHeightOfBuildings = function(buildings) {

};
```

TypeScript Solution:

```
/**
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function averageHeightOfBuildings(buildings: number[][]): number[][] {

};
```

C# Solution:

```
/*
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
```

```
*/

public class Solution {
public int[][] AverageHeightOfBuildings(int[][] buildings) {

}
}
```

C Solution:

```
/*
* Problem: Average Height of Buildings in Each Segment
* Difficulty: Medium
* Tags: array, tree, greedy, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** averageHeightOfBuildings(int** buildings, int buildingsSize, int*
buildingsColSize, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Average Height of Buildings in Each Segment
// Difficulty: Medium
// Tags: array, tree, greedy, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height
```

```
func averageHeightOfBuildings(buildings [][]int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun averageHeightOfBuildings(buildings: Array<IntArray>): Array<IntArray> {

}
}
```

Swift Solution:

```
class Solution {
func averageHeightOfBuildings(_ buildings: [[Int]]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Average Height of Buildings in Each Segment
// Difficulty: Medium
// Tags: array, tree, greedy, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn average_height_of_buildings(buildings: Vec<Vec<i32>>) -> Vec<Vec<i32>>
{

}
}
```

Ruby Solution:

```
# @param {Integer[][]} buildings
# @return {Integer[][]}
```

```
def average_height_of_buildings(buildings)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $buildings
* @return Integer[][]
*/
function averageHeightOfBuildings($buildings) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> averageHeightOfBuildings(List<List<int>> buildings) {

}
}
```

Scala Solution:

```
object Solution {
def averageHeightOfBuildings(buildings: Array[Array[Int]]): Array[Array[Int]]
= {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec average_height_of_buildings(buildings :: [[integer]]) :: [[integer]]
def average_height_of_buildings(buildings) do

end
```

```
end
```

Erlang Solution:

```
-spec average_height_of_buildings(Buildings :: [[integer()]]) ->
[[integer()]].
average_height_of_buildings(Buildings) ->
.
```

Racket Solution:

```
(define/contract (average-height-of-buildings buildings)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```