

Problem 1253: Reconstruct a 2-Row Binary Matrix

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

Given the following details of a matrix with

n

columns and

2

rows :

The matrix is a binary matrix, which means each element in the matrix can be

0

or

1

.

The sum of elements of the 0-th(upper) row is given as

upper

.

The sum of elements of the 1-st(lower) row is given as

lower

.

The sum of elements in the i-th column(0-indexed) is

colsum[i]

, where

colsum

is given as an integer array with length

n

.

Your task is to reconstruct the matrix with

upper

,

lower

and

colsum

.

Return it as a 2-D integer array.

If there are more than one valid solution, any of them will be accepted.

If no valid solution exists, return an empty 2-D array.

Example 1:

Input:

upper = 2, lower = 1, colsum = [1,1,1]

Output:

[[1,1,0],[0,0,1]]

Explanation:

[[1,0,1],[0,1,0]], and [[0,1,1],[1,0,0]] are also correct answers.

Example 2:

Input:

upper = 2, lower = 3, colsum = [2,2,1,1]

Output:

[]

Example 3:

Input:

upper = 5, lower = 5, colsum = [2,1,2,0,1,0,1,2,0,1]

Output:

[[1,1,1,0,1,0,0,1,0,0],[1,0,1,0,0,0,1,1,0,1]]

Constraints:

$1 \leq \text{colsum.length} \leq 10^5$

0 <= upper, lower <= colsum.length

0 <= colsum[i] <= 2

Code Snippets

C++:

```
class Solution {
public:
    vector<vector<int>> reconstructMatrix(int upper, int lower, vector<int>&
    colsum) {

    }
};
```

Java:

```
class Solution {
    public List<List<Integer>> reconstructMatrix(int upper, int lower, int[]
    colsum) {

    }
}
```

Python3:

```
class Solution:
    def reconstructMatrix(self, upper: int, lower: int, colsum: List[int]) ->
    List[List[int]]:
```

Python:

```
class Solution(object):
    def reconstructMatrix(self, upper, lower, colsum):
        """
        :type upper: int
        :type lower: int
        :type colsum: List[int]
        :rtype: List[List[int]]
        """
```

JavaScript:

```
/**
 * @param {number} upper
 * @param {number} lower
 * @param {number[]} colsum
 * @return {number[][]}
 */
var reconstructMatrix = function(upper, lower, colsum) {

};
```

TypeScript:

```
function reconstructMatrix(upper: number, lower: number, colsum: number[]):
number[][] {

};
```

C#:

```
public class Solution {
    public IList<IList<int>> ReconstructMatrix(int upper, int lower, int[]
colsum) {

    }
}
```

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** reconstructMatrix(int upper, int lower, int* colsum, int colsumSize,
int* returnSize, int** returnColumnSizes) {

}
```

Go:

```

func reconstructMatrix(upper int, lower int, colsum []int) [][]int {

}

```

Kotlin:

```

class Solution {
    fun reconstructMatrix(upper: Int, lower: Int, colsum: IntArray):
        List<List<Int>> {

    }
}

```

Swift:

```

class Solution {
    func reconstructMatrix(_ upper: Int, _ lower: Int, _ colsum: [Int]) ->
        [[Int]] {

    }
}

```

Rust:

```

impl Solution {
    pub fn reconstruct_matrix(upper: i32, lower: i32, colsum: Vec<i32>) ->
        Vec<Vec<i32>> {

    }
}

```

Ruby:

```

# @param {Integer} upper
# @param {Integer} lower
# @param {Integer[]} colsum
# @return {Integer[][]}
def reconstruct_matrix(upper, lower, colsum)

end

```

PHP:

```

class Solution {

    /**
     * @param Integer $upper
     * @param Integer $lower
     * @param Integer[] $colsum
     * @return Integer[][]
     */
    function reconstructMatrix($upper, $lower, $colsum) {

    }

}

```

Dart:

```

class Solution {
    List<List<int>> reconstructMatrix(int upper, int lower, List<int> colsum) {

    }

}

```

Scala:

```

object Solution {
    def reconstructMatrix(upper: Int, lower: Int, colsum: Array[Int]):
    List[List[Int]] = {

    }

}

```

Elixir:

```

defmodule Solution do
    @spec reconstruct_matrix(upper :: integer, lower :: integer, colsum ::
    [integer]) :: [[integer]]
    def reconstruct_matrix(upper, lower, colsum) do

    end

end

```

Erlang:

```
-spec reconstruct_matrix(Upper :: integer(), Lower :: integer(), Colsum ::
[integer()]) -> [[integer()]].
reconstruct_matrix(Upper, Lower, Colsum) ->
.
```

Racket:

```
(define/contract (reconstruct-matrix upper lower colsum)
  (-> exact-integer? exact-integer? (listof exact-integer?) (listof (listof
exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
 * Problem: Reconstruct a 2-Row Binary Matrix
 * Difficulty: Medium
 * Tags: array, greedy
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
public:
    vector<vector<int>> reconstructMatrix(int upper, int lower, vector<int>&
colsum) {

    }

};
```

Java Solution:

```
/**
 * Problem: Reconstruct a 2-Row Binary Matrix
 * Difficulty: Medium
 * Tags: array, greedy
 */
```

```

* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public List<List<Integer>> reconstructMatrix(int upper, int lower, int[]
colsum) {

}

}

```

Python3 Solution:

```

"""
Problem: Reconstruct a 2-Row Binary Matrix
Difficulty: Medium
Tags: array, greedy

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def reconstructMatrix(self, upper: int, lower: int, colsum: List[int]) ->
List[List[int]]:
# TODO: Implement optimized solution
pass

```

Python Solution:

```

class Solution(object):
def reconstructMatrix(self, upper, lower, colsum):
"""
:type upper: int
:type lower: int
:type colsum: List[int]
:rtype: List[List[int]]
"""

```

JavaScript Solution:

```
/**
 * Problem: Reconstruct a 2-Row Binary Matrix
 * Difficulty: Medium
 * Tags: array, greedy
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

/**
 * @param {number} upper
 * @param {number} lower
 * @param {number[]} colsum
 * @return {number[][]}
 */
var reconstructMatrix = function(upper, lower, colsum) {

};
```

TypeScript Solution:

```
/**
 * Problem: Reconstruct a 2-Row Binary Matrix
 * Difficulty: Medium
 * Tags: array, greedy
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

function reconstructMatrix(upper: number, lower: number, colsum: number[]):
number[][] {

};
```

C# Solution:

```
/*
 * Problem: Reconstruct a 2-Row Binary Matrix
```

```

* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public IList<IList<int>> ReconstructMatrix(int upper, int lower, int[]
colsum) {

}

}

```

C Solution:

```

/*
* Problem: Reconstruct a 2-Row Binary Matrix
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* 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** reconstructMatrix(int upper, int lower, int* colsum, int colsumSize,
int* returnSize, int** returnColumnSizes) {

}

```

Go Solution:

```

// Problem: Reconstruct a 2-Row Binary Matrix
// Difficulty: Medium
// Tags: array, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func reconstructMatrix(upper int, lower int, colsum []int) [][]int {

}

```

Kotlin Solution:

```

class Solution {
    fun reconstructMatrix(upper: Int, lower: Int, colsum: IntArray):
        List<List<Int>> {

    }
}

```

Swift Solution:

```

class Solution {
    func reconstructMatrix(_ upper: Int, _ lower: Int, _ colsum: [Int]) ->
        [[Int]] {

    }
}

```

Rust Solution:

```

// Problem: Reconstruct a 2-Row Binary Matrix
// Difficulty: Medium
// Tags: array, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
    pub fn reconstruct_matrix(upper: i32, lower: i32, colsum: Vec<i32>) ->

```

```
Vec<Vec<i32>> {

}

}
```

Ruby Solution:

```
# @param {Integer} upper
# @param {Integer} lower
# @param {Integer[]} colsum
# @return {Integer[][]}
def reconstruct_matrix(upper, lower, colsum)

end
```

PHP Solution:

```
class Solution {

    /**
     * @param Integer $upper
     * @param Integer $lower
     * @param Integer[] $colsum
     * @return Integer[][]
     */
    function reconstructMatrix($upper, $lower, $colsum) {

    }

}
```

Dart Solution:

```
class Solution {
  List<List<int>> reconstructMatrix(int upper, int lower, List<int> colsum) {

  }
}
```

Scala Solution:

```
object Solution {
  def reconstructMatrix(upper: Int, lower: Int, colsum: Array[Int]):
```

```
List[List[Int]] = {  
  
}  
}
```

Elixir Solution:

```
defmodule Solution do  
  @spec reconstruct_matrix(upper :: integer, lower :: integer, colsum ::  
    [integer]) :: [[integer]]  
  def reconstruct_matrix(upper, lower, colsum) do  
  
  end  
end
```

Erlang Solution:

```
-spec reconstruct_matrix(Upper :: integer(), Lower :: integer(), Colsum ::  
  [integer()]) -> [[integer()]].  
reconstruct_matrix(Upper, Lower, Colsum) ->  
.
```

Racket Solution:

```
(define/contract (reconstruct-matrix upper lower colsum)  
  (-> exact-integer? exact-integer? (listof exact-integer?) (listof (listof  
    exact-integer?)))  
)
```