1439. Find the Kth Smallest Sum of a Matrix With Sorted Rows

### **Problem Description**

Array

Hard

The given problem presents a challenge where we are dealing with a matrix called mat with dimensions m x n, where each row is sorted in a non-decreasing order. The task here is to create arrays by choosing exactly one element from each row of the matrix. The aim is to find the kth smallest sum that can be obtained from all the possible arrays created in this way. An array's sum is the total of all the elements within that array.

Binary Search Matrix Heap (Priority Queue)

selected).

Intuition

Here's how we conceptualize our approach: Begin Small: Initialize a list, let's call it pre, with a single element 0, which represents the smallest initial sum (with no elements

To approach this problem, we utilize a step by step strategy that builds upon the smallest possible sums array by array, while

looking ahead by only considering the top k smallest sums at each step to reduce computational complexity.

maintained to be sorted with only k elements. Thus, pre[-1] yields the desired result.

elements chosen so far, and with no elements having been considered, the sum is simply 0.

- **Build Up:** For each row in the matrix, we combine the values in pre with the new values from the current row. This helps us to generate all possible sums with one more element added to the arrays. Stay Within Bounds: To maintain performance and not generate an unnecessarily large number of sums, we only keep the
- top k smallest sums at any given step. This is done by sorting the generated sums and slicing the list to retain only the first k elements.
- Iterate Through Rows: Repeat step 2 and 3 for each row of the matrix. With each iteration, pre grows to contain the smallest sums resulting from picking exactly one element from each of the rows processed so far. **Final Answer**: After processing all the rows, the last element in pre will be the kth smallest sum, because pre is always
- By employing this incremental approach and limiting the consideration to k elements at each step, we efficiently find the desired sum without having to explore all possible combinations, which would be infeasible for large matrices.
- The solution employs an efficient approach utilizing simple data structures and Python's list comprehension to handle the potentially large search space in a way that is both manageable and scalable to larger datasets.

Here's a step-by-step breakdown of the implementation: Initialization: We begin by creating a list called pre, initialized with a single element, 0. This list represents the sum of

for cur in mat:

return pre[-1]

**Example Walkthrough** 

Let mat be:

exactly one element from each row.

pre = [0]

Solution Approach

Iterative Combination: For every row cur in the matrix mat, we use a nested loop through list comprehension to create new sums. This is where we explore every combination of the previously accumulated sums in pre with the new elements from the current row cur.

new sum a + b. The slicing cur[:k] ensures that we don't consider more elements than necessary, which is crucial for performance optimization.

pre = [a + b for a in pre for b in cur[:k]]

list for the next iteration. pre = sorted(pre)[:k] Repeat Process: This process is repeated iteratively for each row of the matrix, each time updating the pre list with the smallest sums obtained from choosing exactly one element from each row processed up to that point.

The kth smallest value, which is the answer we need to find, will be the last element of this list.

*Inner Loop Explanation*: For every element a in pre and for every element b in the top k elements of the current cur, compute a

**Sorting and Trimming:** After combining sums from pre and the current row, this new list could be quite large, larger than k.

We need to only keep the smallest k sums. We do this by first sorting the new sums and then slicing the list to retain only the

first k elements. The sorting operation here not only ensures that we are keeping the smallest sums but also prepares the pre

Extracting Result: After the last iteration, the final list pre will be sorted and contain the k smallest sums of all possible arrays.

By utilizing this approach, we effectively minimize the number of possible sums we have to consider at each step, which is essential for handling inputs where the number of combinations can be extraordinarily high. The algorithm leverages the sorted property of the matrix's rows and Python's efficient list operations to deliver a solution that meets the problem's constraints.

Let's consider a matrix mat with two rows and three columns, and suppose we want to find the 2nd smallest sum by choosing

We want the 2nd smallest sum when picking one element from each row, meaning k=2 in this context. **Step 1: Initialization** We initialize pre with a single element, 0, representing the empty sum. pre = [0]

# We now have all possible sums that can be generated by choosing one element from the first row with pre only keeping the

smallest 2 sums.

pre = sorted(pre)[:2]

elements.

# Second row

return pre[-1]

**Python** 

# pre becomes [0+1, 0+3] = [1, 3]

# The 2nd smallest sum is hence

Solution Implementation

from typing import List

prev\_row\_sums = [0]

for row in matrix:

class Solution:

class Solution {

# First row

Repeat the iterative combination with the next row [2, 5, 7]. Step 4: Repeat Process Take the next row, and combine each element with the sums in pre while considering only the first k

Again, sort pre and keep only the first k elements. pre = sorted(pre)[:2] # After sorting, pre becomes [3, 5], so we keep these as the smallest sums.

Step 5: Extracting Result After processing all rows, the final list pre contains the k smallest sums.

pre = [a + b for a in pre for b in [2, 5, 7][:2]] # Again, take the first 2 elements only for <math>k=2.

Step 2: Iterative Combination Starting with the first row [1, 3, 4], combine pre with this row.

Step 3: Sorting and Trimming Sort pre and keep only the first k elements.

# pre becomes [1+2, 1+5, 3+2, 3+5] = [3, 6, 5, 8] (combinations of sums)

# Since pre is already [1, 3], no change is made in this case.

pre = [a + b for a in pre for b in [1, 3, 4][:2]] # We only use the first 2 elements for k=2.

in this list, which is 5. This sum comes from the array [1, 4], the second smallest array sum by choosing one element from each row of the given mat.

def kthSmallest(self, matrix: List[List[int]], k: int) -> int:

# Iterate through each row in the matrix.

public int kthSmallest(int[][] matrix, int k) {

// Get the dimensions of the matrix.

int rows = matrix.length;

int cols = matrix[0].length;

# Initialize a list with a single element 0 to start the accumulation.

# in the next iteration or to decide the final result.

// Initialize a list to store the previous row's computations.

// Initialize a list to store the current row's computations.

// Start with 0 as the only element for an empty prefix sum.

// Add the sum to the current list.

currentRowSums.add(prevSum + value);

List<Integer> currentRowSums = new ArrayList<>(cols \* k);

List<Integer> previousRowSums = new ArrayList<>(k);

for (int prevSum : previousRowSums) {

for (int value : row) {

Collections.sort(currentRowSums);

for (int j = 0; j < size; ++j) {

current[index++] = previous[j] + value;

// After combining all rows, the k-th smallest sum is at index k-1

// Sort the 'current' array to bring the smallest sums to the front

// Only keep the 'k' smallest sums in 'previous' array for next iteration

// Initialize an array to store new sums that include elements from the current row

for (int value : row) {

sort(current, current + index);

for (int j = 0; j < size; ++j) {

previous[j] = current[j];

size = min(index, k);

return previous[k - 1];

for (const currentRow of matrix) {

const newRowSums: number[] = [];

# Generate new sums by adding each element in the current row to each

# previously computed sum. Since we're only interested in the k smallest sums,

# Then, we sort the resulting sums and keep only the k smallest sums to use

# we only consider the first k elements in the row to avoid unnecessary computation.

// Combine each element from the previous list with each element of the current row.

// Sort the current list to prepare for selecting the smallest k elements.

// Clear the previous sums list to reuse it for the next iteration.

prev\_row\_sums = sorted(sum\_1 + sum\_2 for sum\_1 in prev\_row\_sums for sum\_2 in row[:k])[:k]

# pre[-1] gives 5, which is the 2nd smallest sum possible.

# After processing all rows, the k-th smallest sum is the last element # in the list of k smallest sums. return prev\_row\_sums[-1] Java

Therefore, by following the approach, we end up with a pre list that contains [3, 5], and the 2nd smallest sum is the last element

```
previousRowSums.add(0);
// Iterate through each row of the matrix.
for (int[] row : matrix) {
    // Clear the current sums list for new calculations.
    currentRowSums.clear();
```

```
previousRowSums.clear();
            // Take the first k elements from the current list, or the entire list if it's smaller than k.
            for (int i = 0; i < Math.min(k, currentRowSums.size()); ++i) {</pre>
                // Add each of the smallest elements to the previous sums list.
                previousRowSums.add(currentRowSums.get(i));
       // Return the k-th smallest sum.
       // Subtract 1 from k because of 0-based indexing in Java lists.
        return previousRowSums.get(k - 1);
C++
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
       // Initialize the 'previous' array with all zeros and max length as 'k'
       int previous[k];
       memset(previous, 0, sizeof(previous));
       // initial size of the combined list is 1 because we start with zero elements
       int size = 1;
       // initialize the 'current' array with length up to 'matrix[0].size() * k'
       // since each addition of a row can at most add that many combinations
       int current[matrix[0].size() * k]; // This could be optimized using dynamic arrays(Dynamic array/vector is recommended)
       // Iterate through each row of the matrix
        for (auto& row : matrix) {
            int index = 0; // Index to store the sum of elements in 'current' array
            // Generate sums for all the possible combinations with the current row
```

```
// Defines a function to find the kth smallest element in a sorted matrix
function kthSmallest(matrix: number[][], k: number): number {
    // Initialize a preliminary array with zero, which will keep track of the sum permutations
    let previousRowSums: number[] = [0];
    // Iterate over each row of the matrix
```

**TypeScript** 

**}**;

```
// Iterate over each sum in the previous sums array
          for (const sum of previousRowSums) {
              // Add the current element to each of the previous sums
              for (const element of currentRow) {
                  newRowSums.push(sum + element);
          // Sort the sums array and keep only the smallest k sums for the next iteration
          previousRowSums = newRowSums.sort((a, b) => a - b).slice(0, k);
      // Return the kth smallest element
      return previousRowSums[k - 1];
from typing import List
class Solution:
   def kthSmallest(self, matrix: List[List[int]], k: int) -> int:
       # Initialize a list with a single element 0 to start the accumulation.
       prev_row_sums = [0]
       # Iterate through each row in the matrix.
        for row in matrix:
           # Generate new sums by adding each element in the current row to each
            # previously computed sum. Since we're only interested in the k smallest sums,
            # we only consider the first k elements in the row to avoid unnecessary computation.
            # Then, we sort the resulting sums and keep only the k smallest sums to use
            # in the next iteration or to decide the final result.
            prev_row_sums = sorted(sum_1 + sum_2 for sum_1 in prev_row_sums for sum_2 in row[:k])[:k]
       # After processing all rows, the k-th smallest sum is the last element
       # in the list of k smallest sums.
```

## **Time Complexity** The time complexity of the given code is determined by the operations executed in the nested loops.

through the iterations.

return prev\_row\_sums[-1]

Time and Space Complexity

matrix, we have a maximum of k elements considered in cur. Thus, in the inner loop, the number of sums will be on the order of len(pre) \* k. Initially, len(pre) = 1 and can grow up to k after sorting and truncation. • Sorting the list pre can take 0(k log k) time at most since we only keep the smallest k elements.

The worst-case computational load at each iteration is dominated by the sorting step, which is O(k log k). Thus, the total time complexity of the algorithm is  $0(m * k * \log k)$ .

• The outer loop iterates over each row of the matrix mat, which we can denote as m, where m is the number of rows in the matrix.

• The inner loop essentially computes all pairwise sums between elements in pre and cur[:k]. If n represents the number of columns in the

**Space Complexity** The space complexity of the code is mainly due to the storage of the list pre, which maintains a length of at most k elements

• The generation of pairwise sums (a + b for a in pre for b in cur[:k]) creates an intermediary list of size up to len(pre) \* k, but since it's

immediately sorted and truncated to size k, the space consumption does not accumulate over iterations. Hence, the space complexity of the code is O(k).

At each step of the outer loop, pre is replaced by a new list of length at most k.