2218. Maximum Value of K Coins From Piles

Dynamic Programming Prefix Sum

Problem Description

Array

Hard

piles, where each element piles[i] is itself a list that contains integers representing the coins' values in the ith pile from the top to the bottom of the pile. The goal is to maximize the total value of coins you can collect by picking coins from the tops of these piles. However, there's a

You are given n piles of coins, each pile containing a stack of coins with varying denominations. The piles are represented by a list

catch: you can only make a total of k moves, and in each move, you can only pick one coin from the top of any pile. Once you take a coin from a pile, you can't skip coins; you can only take another coin from the same pile if it's now the new top coin of that pile.

moves and do so in the most optimal way possible. Intuition

Knowing this, you need to find the maximum total value of coins you can amass in your wallet assuming you make exactly k

(DP) because it involves making a series of decisions that rely on the outcomes of previous decisions.

The intuition behind the solution involves two key insights: 1. We need to track the maximum value we can achieve for any number of moves up to k, as we progress through the piles. 2. We must consider all possible options for each pile, i.e., we can choose 0 coins from the pile, or we can take 1 coin, or we can take 2 coins, and so on, and for each of these choices, we need to update the maximum value possible considering the new total number of moves.

To approach this problem effectively, we need a strategy that allows us to make local decisions that contribute to our overall goal

of maximizing the value of the coins we collect. This is a classic optimization problem that hints at using dynamic programming

array is initialized with zeros, as initially, no coins are taken.

Mathematically, this is represented as follows:

A list of lists presum for the prefix sums of each pile.

• <u>Dynamic programming</u> for optimal substructure and overlapping subproblems.

• Prefix sums to quickly calculate the sum of taking a certain number of coins from a pile.

First, we calculate the prefix sum for each pile to make summing coins efficient.

dp[j] = max(dp[j], dp[j - idx] + v)

The implementation uses a 2D dynamic programming approach, which is effectively compressed into a 1D DP array dp for space

efficiency. The dp array stores the maximum value that can be achieved for each possible number of moves up to k. The presum array contains the prefix sums for each pile, which allows us to easily compute the total value gained by taking the

The trick is to iterate through all piles and all possible coin counts we might take from each pile, update our dp array to store the best result. The updating is done in reverse to ensure that we don't overwrite a DP state before we're done using it for the current

pile. At the end of the iteration, dp[k] will hold our answer, which is the maximum value we can collect with k coins picked.

Solution Approach The solution involves dynamic programming (DP), a technique that solves problems by breaking them down into simpler

subproblems. We use a 1D DP array, dp, where each index j represents the maximum value achievable with j moves. The DP

For each index j of our dp array, we loop through all possible coin counts, idx, that we could take from the current pile,

For each pile s in our presum array, where presum is a list of prefix sums (cumulative sums from the start of the pile to the current

represented by the different values in the prefix sum. If taking idx coins is feasible (i.e., j is greater than or equal to idx), then we

update our dp[j] by choosing the max between its current value and the sum of dp[j - idx] and the value of the idx-th coin, v.

index), we iterate through our dp array in reverse. We do this because we want to update our dp values for using a certain number of coins, idx, from the current pile without affecting the other values of dp that we haven't processed yet.

first i coins from a pile.

This line of the code represents the core of the DP transition equation. It updates the dp[j] to the maximum value we could have if we did not take any coins from the current pile or if we took idx coins from the current pile.

The process continues until we have processed all piles and considered all possible moves up to k. The last element of the dp

array, dp [-1] or dp [k], will contain the maximum total value achievable by taking k coins across all piles. **Data Structures:** A list dp for the DP array.

By carefully iterating and updating our DP array, we ensure that at the completion of the algorithm, the dp[k] index will hold the

Iterating in reverse to preserve DP states for processing without interference.

Algorithms and Patterns:

- **Example Walkthrough**
- Let's walk through a small example to illustrate the solution approach. Suppose we have the following piles of coins and can make

maximum value of coins we can collect out of k moves.

We want to find the maximum value we can collect by making exactly two moves.

Step 1: Create Prefix Sums for Each Pile

piles = [[1, 2], [2, 1]]

k = 2 moves:

coins gives 3). For the second pile [2, 1], the prefix sums are [2, 3].

In each iteration, idx is considered as the number of coins we pick from the current pile (hence the added values from the prefix

After iterating through all piles, the last entry in the dp array gives us the answer. For k = 2, dp[k] is 3, so the maximum value we

For the first pile [1, 2], the prefix sums are [1, 3] (taking 0 coins gives 0, taking the first coin gives 1, and taking the first two

Step 3: Iterate Over Piles and Update DP Array

dp = [0, 0, 0]

Step 2: Initialize the DP Array

 \circ For j = 1: dp[1] = max(dp[1], dp[1 - 0] + 1) = max(0, 0 + 1) = 1 (using 1 coin from the pile) \circ For j = 1: dp[1] = max(dp[1], dp[1 - 1] + 1) = max(1, 0 + 2) = 2 (opting to instead use 2 coins from the pile)

 \circ For j = 1: dp[1] = max(dp[1], dp[1 - 0] + 2) = max(2, 0 + 2) = 2 (no change)

 \circ For j = 2: dp[2] = max(dp[2], dp[2 - 2] + 3) = max(3, 0 + 3) = 3 (no change)

Initialize the DP array with 0's; size k+1 for the 'zero' case

in the calculation of another value in the same step

This is to ensure that we do not use a value from this step

it does not exceed the remaining_coin limit

Enumerate over the prefix sums providing index and value

Make sure when taking 'index' coins from this pile,

 \circ For j = 2: dp[2] = max(dp[2], dp[2 - 1] + 1) = max(0, 1 + 2) = 3 (taking 1 coin previously and 1 coin now)

 \circ For j = 1: dp[1] = max(dp[1], dp[1 - 1] + 2) = max(2, 0 + 2) = 2 (opting to instead use 2 coins from the pile)

 \circ For j = 2: dp[2] = max(dp[2], dp[2 - 1] + 2) = max(3, 2 + 1) = 3 (taking 1 coin from the first pile and 1 from this one)

• For j = 2: dp[2] = max(dp[2], dp[2 - 2] + 3) = max(3, 0 + 3) = 3 (taking both coins from this pile)

Next, we initialize a dp array with a length of k+1 (for 0 to k moves), starting with all zeros:

After processing the first pile, our dp array is [0, 2, 3]. For the second pile's prefix sums [2, 3], we perform similar updates:

Step 4: Collect the Result

can collect is 3.

Python

Solution Implementation

dp = [0] * (k + 1)

for sums in prefix_sums:

int numPiles = piles.size();

int pileSize = pile.size();

prefixSums.add(prefixSum);

int[] dp = new int[k + 1];

Iterate over each pile's prefix sums

Iterate over the DP values in reverse

for remaining_coins in range(k, -1, -1):

for index, value in enumerate(sums):

if remaining_coins >= index:

public int maxValueOfCoins(List<List<Integer>> piles, int k) {

prefixSum[i + 1] = prefixSum[i] + pile.get(i);

// Initialize DP array to store the maximum value we can achieve picking i coins

// Iterate through the possible number of coins to take in reverse

// Iterate through each index in the current prefix sum vector

sum[i + 1] = sum[i] + pile[i]; // Calculate the prefix sum for the current pile

// If the current number of coins to take is greater or equal to the index

// If the current number of coins to take is greater or equal to the index

dp[coinsToTake] = max(dp[coinsToTake], dp[coinsToTake - idx] + sums[idx]); // Update the DP value with ma

dp[coinsToTake] = Math.max(dp[coinsToTake], dp[coinsToTake - idx] + sums[idx]); // Update the DP value with 1

for (int coinsToTake = k; coinsToTake >= 0; --coinsToTake) {

for (int idx = 0; idx < sums.size(); ++idx) {</pre>

if (coinsToTake >= idx)

// Return the maximum value for taking k coins

function maxValueOfCoins(piles: number[][], k: number): number {

// Create an array to store the prefix sums for all piles

let sum: number[] = new Array(pileSize + 1).fill(0);

for (let idx = 0; idx < sums.length; idx++) {</pre>

if (coinsToTake >= idx)

prefixSums.push(sum); // Store it in the array of prefix sums

// Create a DP array to store the maximum value of coins for each number up to k

// Iterate through each index in the current prefix sum array

List<int[]> prefixSums = new ArrayList<>();

int[] prefixSum = new int[pileSize + 1];

for (int i = 0; i < pileSize; ++i) {</pre>

// Iterate through each pile's prefix sums

After processing the second pile, our dp array is still [0, 2, 3].

We iterate over each prefix sum array and update our dp:

For the first pile's prefix sums [1, 3], we compare:

sums). We choose the largest possible value to assign to dp[j].

- In this example, the best strategy is to take one coin with a value of 2 from the first pile and one coin with a value of 1 from the second pile, or to take the top coin from each pile, as both strategies yield the total maximum value of 3.
- from itertools import accumulate class Solution: def maxValueOfCoins(self, piles: List[List[int]], k: int) -> int: # Precompute the prefix sums for each pile # Each prefix sum list starts with 0 for convenience prefix_sums = [list(accumulate(pile, initial=0)) for pile in piles]

Update the DP value with the maximum between the current value

and the value achievable by taking 'index' coins from this pile

The last element of dp is the maximum value achievable by taking k coins return dp[-1] Java

dp[remaining_coins] = max(dp[remaining_coins], dp[remaining_coins - index] + value)

// Calculate prefix sums for each pile to have quick access to the sum of the first i coins for (List<Integer> pile : piles) {

class Solution {

```
for (int[] prefixSum : prefixSums) {
            // Iterate over the dp array in reverse to avoid overwriting values we still need to read
            for (int j = k; j \ge 0; ---j) {
                // Try taking 0 to pileSize coins from the current pile and update dp values
                for (int idx = 0; idx < prefixSum.length; ++idx) {</pre>
                    if (j >= idx) {
                        // Update the dp value if taking idx coins gives us a better result
                        dp[j] = Math.max(dp[j], dp[j - idx] + prefixSum[idx]);
        // Finally, return the maximum value that can be achieved by taking k coins
        return dp[k];
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    // Function to calculate the maximum value of coins we can obtain by selecting at most k coins from several piles
    int maxValueOfCoins(vector<vector<int>>& piles, int k) {
        // Create a vector to store the prefix sums for all piles
        vector<vector<int>> prefixSums;
        for (auto& pile : piles) {
            int pileSize = pile.size();
            vector<int> sum(pileSize + 1);
            for (int i = 0; i < pileSize; ++i)</pre>
                sum[i + 1] = sum[i] + pile[i]; // Calculate the prefix sum for the current pile
            prefixSums.push_back(sum); // Store it in the vector of prefix sums
        // Create a DP vector to store the maximum value of coins for each k
       vector<int> dp(k + 1);
        // Iterate through the prefix sums of each pile
        for (auto& sums : prefixSums) {
```

```
let dp: number[] = new Array(k + 1).fill(0);
// Iterate through the prefix sums of each pile
for (let sums of prefixSums) {
   // Iterate through the possible number of coins to take in reverse
    for (let coinsToTake = k; coinsToTake >= 0; coinsToTake--) {
```

return dp[k];

let prefixSums: number[][] = [];

let pileSize: number = pile.length;

for (let i = 0; i < pileSize; i++)</pre>

for (let pile of piles) {

};

TypeScript

```
// Return the maximum value for taking k coins
      return dp[k];
from itertools import accumulate
class Solution:
   def maxValueOfCoins(self, piles: List[List[int]], k: int) -> int:
       # Precompute the prefix sums for each pile
       # Each prefix sum list starts with 0 for convenience
        prefix_sums = [list(accumulate(pile, initial=0)) for pile in piles]
       # Initialize the DP array with 0's; size k+1 for the 'zero' case
       dp = [0] * (k + 1)
       # Iterate over each pile's prefix sums
        for sums in prefix_sums:
           # Iterate over the DP values in reverse
           # This is to ensure that we do not use a value from this step
           # in the calculation of another value in the same step
           for remaining_coins in range(k, -1, -1):
               # Enumerate over the prefix sums providing index and value
                for index, value in enumerate(sums):
                   # Make sure when taking 'index' coins from this pile,
                   # it does not exceed the remaining_coin limit
                   if remaining_coins >= index:
                       # Update the DP value with the maximum between the current value
                       # and the value achievable by taking 'index' coins from this pile
                       dp[remaining_coins] = max(dp[remaining_coins], dp[remaining_coins - index] + value)
       # The last element of dp is the maximum value achievable by taking k coins
        return dp[-1]
Time and Space Complexity
  The time complexity of the provided code can be analyzed by breaking down its operations. We need to consider the number of
  iterations in the nested loops along with the cost of constructing the prefix sum presum.
  The first operation is the construction of the prefix sums for each pile, which is O(n) per pile where n is the number of coins in the
  pile. If there are m piles, this step will be 0(m * n).
```

Next, we look at the nested loops. The outermost loop iterates over each pile's prefix sum list, which will happen m times (the

omitted in Big O notation, it simplifies to 0(m * k).

number of piles). The middle loop will always iterate k times since it goes from k to 0. The innermost loop depends on the length of the current prefix sum list, which can be a maximum of k+1, because we will never take more than k coins from a single pile. Combining the total iterations for the nested loops, we have m (number of piles) multiplied by k (maximum coins we can pick)

multiplied by k+1 (coinciding with the prefix sums length). Therefore, the total time complexity of the nested loops is $0(m * k^2)$. Hence, the overall time complexity of the algorithm is dominated by the nesting loops, resulting in $0(m * k^2)$.

The space complexity of the algorithm can be determined by looking at the extra space used. The presum variable will store up to k+1 elements for each of the m piles, which means that its space complexity is 0(m * (k+1)). However, since constants are

The dp array will have k+1 elements no matter what, leading to a space complexity of O(k).

Since dp does not grow with m, the overall space complexity will be dominated by presum, so the overall space complexity is 0(m *