2017. Grid Game

Medium Array Matrix Prefix Sum

Problem Description

In this problem, two robots are playing a game on a `2 x n` grid matrix where each cell contains a certain number of points. The grid is 0-indexed which means that the rows and columns are numbered starting from 0.

Both robots start from the top-left corner at position `(0, 0)` and their goal is to reach the bottom-right corner at position `(1, n-1)`. They can only move right (from `(r, c)` to `(r, c + 1)`) or down (from `(r, c)` to `(r + 1, c)`).

The game is played in two stages:

1. The first robot moves first and collects points from each cell it passes through, setting the points in those cells to 0 after passing through them.
2. After the first robot finishes its path, the second robot makes its way from `(0, 0)` to `(1, n-1)`, collecting remaining points.

The first robot aims to minimize the points that the second robot can collect by choosing an optimal path, while the second robot aims to maximize its points by also choosing an optimal path after the first robot has completed its route.

The objective is to determine the number of points that the second robot will collect if both robots play optimally.

Intuition

The solution is based on the idea that we need to find the path for the first robot that minimizes the maximum points that the second robot can collect. One approach is to simulate the path of the first robot and keep track of the points that will be left for the second robot.

To find the optimal path for the first robot, we need to consider two possible scenarios for the second robot:

1. The second robot could take the maximum points from the top row after the first robot completes its path.
2. The second robot could take the maximum points from the bottom row after the first robot finishes its path.

We can keep track of two running sums: `s1`, for the sum of points left on the top row, and `s2`, for the sum of points collected from the bottom row. As we iterate through each column:

- We subtract the points collected by the first robot from `s1`, as those points are no longer available for the second robot.
- We then calculate the maximum points the second robot could collect, which is either the remaining points on the top row (`s1`) or the points it has accumulated from the bottom row (`s2`), whichever is larger.
- We keep track of the minimum such value (`ans`) across all the column iterations since we are looking for the path for the first robot that minimizes the maximum points that the second robot can collect.

The answer we want is this minimum of the maximum points, which is the best that the second robot can do if the first robot plays optimally.

Solution Approach

The key algorithm used in this solution is essentially a greedy approach coupled with dynamic programming concepts. We need to keep track of two key metrics as we iterate through the grid: the sum of the remaining points on the top row (`s1`) after the first robot moves, and the sum of points on the bottom row that the second robot can potentially collect (`s2`). Here's a breakdown of how the algorithm proceeds:

1. Initialize `s1` to the sum of all elements in the top row of the grid (this represents the maximum number of points available to the second robot if the first robot drops straight down to the bottom row from the beginning).
2. Initialize `s2` to 0 as the second robot starts from the first column and has not collected any points yet.
3. Initialize `ans` to `inf` which stands for infinity. This variable will hold the minimum number of points the second robot can collect after the first robot has chosen its path.
4. Iterate through each column (element) in the top row.
    - Subtract the current element's value from `s1` since the first robot will collect these points and they will no longer be available for the second robot.
    - Calculate the maximum points the second robot can collect after the first robot's move, which is the maximum of `s1` and `s2`. This represents the worst-case points that the second robot can get if the first robot moves right on the current step.
    - Update `ans` with the minimum of itself and the maximum points from the previous step. This effectively stores the best (minimum) outcome for the second robot so far considering all the columns processed.
    - Add the current element's value from the bottom row to `s2`. This is because the second robot would collect points from the bottom row if the first robot moves right.
5. After the loop, `ans` stores the minimum of the maximum points that the second robot can collect, given that both robots play optimally. It is the required answer.

Using this approach, we iterate through the grid only once, yielding a time complexity of `O(n)` where `n` is the number of columns in the grid. No additional data structures are used, so the space complexity is `O(1)` as we only store a fixed number of variables.

Here's the part of the solution implementing the above steps:

```
s1, s2 = sum(grid[0]), 0
for j, v in enumerate(grid[0]):
    s1 -= v
    ans = min(ans, max(s1, s2))
    s2 += grid[1][j]
return ans
```

In this code snippet, `s1` and `s2` are updated within the loop, `v` is the value of the current top-row cell, and `grid[1][j]` is the value at the current bottom-row cell.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Consider a `2 x 3` grid matrix where the top row is `[3, 1, 2]` and the bottom row is `[4, 1, 5]`. We want to minimize the points that the second robot can collect, assuming both robots are playing optimally.

1. **Initialization**
    - `s1` (points on the top row) = `3 + 1 + 2 = 6`
    - `s2` (points on the bottom row) = `0` (to start)
    - `ans` (ans to track the optimal outcome for the second robot) = `infinity`
2. **First column iteration (`j = 0, v = 3`)**
    - We subtract the first robot's collected points from `s1`: `s1 = 6 - 3 = 3`
    - The max points the second robot can have now are `max(s1, s2) = max(3, 0) = 3`
    - We update `ans` with the min value: `ans = min(infinity, 3) = 3`
    - The first robot moves down, collecting points from the bottom row: `s2 = 0 + 4 = 4`
3. **Second column iteration (`j = 1, v = 1`)**
    - Now `s1 = 3 - 1 = 2`
    - The max points the second robot can have are `max(s1, s2) = max(2, 4) = 4`
    - Update `ans` with the new min: `ans = min(3, 4) = 3`
    - The first robot moves right, collecting bottom points: `s2 = 4 + 1 = 5`
4. **Third column iteration (`j = 2, v = 2`)**
    - `s1 = 2 - 2 = 0`
    - The max points are `max(s1, s2) = max(0, 5) = 5`
    - Update `ans` again: `ans = min(3, 5) = 3`
    - After the first robot moves right: `s2 = 5 + 5 = 10` (this step is actually not needed as we've reached the last column)

At the completion of our loop, `ans = 3` which is the optimal number of points the second robot can collect if the first robot plays optimally. Thus, the result of the given algorithm for our example would be `3`.

Solution Implementation

Python

```
class Solution:
    def gridGame(self, grid: List[List[int]]) -> int:
        # Initialize the answer to an infinite value since we want to minimize it later
        min_max_score = float('inf')

        # Sum of the top row's elements
        top_sum = sum(grid[0])
        # Initialize bottom sum to 0 since the robot hasn't moved yet
        bottom_sum = 0

        # Iterate through the elements of the top row
        for index, value in enumerate(grid[0]):
            # Robot moves down, so remove the current value from the top row sum
            top_sum -= value
            # Calculate the maximum of the remaining sums after removing the current column
            min_max_score = min(min_max_score, max(top_sum, bottom_sum))
            # Add the current value from the bottom row to its sum as the robot can take it
            bottom_sum += grid[1][index]

        # Return the minimum value found among the maximum sums after each possible move
        return min_max_score
```

Java

```
class Solution {
    public long gridGame(int[][] grid) {
        // Initialize the answer to the maximum possible value.
        long answer = Long.MAX_VALUE;
        // Variables to store the sum of the top row (sumTopRow) and the sum of the bottom row (sumBottomRow).
        long sumTopRow = 0, sumBottomRow = 0;

        // Calculate the initial sum of the top row.
        for (int value : grid[0]) {
            sumTopRow += value;
        }

        // Find the length of the grid rows.
        int numberOfColumns = grid[0].length;

        // Iterate over every column to decide the best path.
        for (int column = 0; column < numberOfColumns; ++column) {
            // Subtract the current value of the top row because the robot will move right from here.
            sumTopRow -= grid[0][column];
            // Calculate the minimum of the maximum of the two paths (top and bottom).
            answer = Math.min(answer, Math.max(sumTopRow, sumBottomRow));
            // Add the current value to the sum of the bottom row as the robot can move down.
            sumBottomRow += grid[1][column];
        }
        // Return the minimum result after traversing all columns.
        return answer;
    }
}
```

C++

```
#include <vector>
#include <algorithm>
#include <climits>

using ll = long long; // Define 'll' as an alias for 'long long' for simplicity

class Solution {
public:
    long long gridGame(std::vector<std::vector<int>>& grid) {
        // The function to calculate the minimal points the second player can obtain

        ll answer = LONG_MAX; // Initialize the answer variable to maximum possible long long value
        int numColumns = grid[0].size(); // Number of columns in the grid
        ll upperSum = 0, lowerSum = 0; // Variables to keep track of the prefix sums of the top and bottom rows

        // Calculate the initial prefix sum of the top row
        for (int value : grid[0]) {
            upperSum += value;
        }

        // Iterate through the grid to find the minimal points the second player will end up with
        for (int columnIndex = 0; columnIndex < numColumns; ++columnIndex) {
            // Decrease the upperSum by the current top grid value since the robot will pass it
            upperSum -= grid[0][columnIndex];

            // Take the maximum of the remaining values in the upperSum and lowerSum, as it's the value the second player is guaranteed to get.
            // Then take the minimum of this and answer to find the minimum points the second player will have to collect through this path.
            answer = std::min(answer, std::max(upperSum, lowerSum));

            // Increase the lowerSum by the current bottom grid value as the robot can collect it
            lowerSum += grid[1][columnIndex];
        }

        // Return the final answer which is the minimal points the second player will get
        return answer;
    }
};
```

TypeScript

```
function gridGame(grid: number[][]): number {
    // Initialize the answer to the maximum safe integer value because we're looking for a minimum.
    let answer = Number.MAX_SAFE_INTEGER;

    // Calculate the sum of values on the top row (robot 1's path) to start with.
    let topRowSum = grid[0].reduce((accumulator, currentValue) => accumulator + currentValue, 0);

    // Initialize sum for the bottom row (robot 2's path) to be 0 since we haven't started summing it yet.
    let bottomRowSum = 0;

    // Iterate through each column.
    for (let columnIndex = 0; columnIndex < grid[0].length; ++columnIndex) {
        // Subtract the current top cell value as robot 1 moves right, no longer able to collect this cell's points.
        topRowSum -= grid[0][columnIndex];

        // Update the minimum answer by comparing the maximum of the two sums after robot 1 moves.
        answer = Math.min(answer, Math.max(topRowSum, bottomRowSum));

        // Add the current bottom cell value to the bottomRowSum as robot 2 moves right, since it can now collect this cell's points.
        bottomRowSum += grid[1][columnIndex];
    }

    // Return the minimum answer which indicates the maximum points robot 2 can score when robot 1 takes the optimal path.
    return answer;
}
```

```
class Solution:
    def gridGame(self, grid: List[List[int]]) -> int:
        # Initialize the answer to an infinite value since we want to minimize it later
        min_max_score = float('inf')

        # Sum of the top row's elements
        top_sum = sum(grid[0])
        # Initialize bottom sum to 0 since the robot hasn't moved yet
        bottom_sum = 0

        # Iterate through the elements of the top row
        for index, value in enumerate(grid[0]):
            # Robot moves down, so remove the current value from the top row sum
            top_sum -= value
            # Calculate the maximum of the remaining sums after removing the current column
            min_max_score = min(min_max_score, max(top_sum, bottom_sum))
            # Add the current value from the bottom row to its sum as the robot can take it
            bottom_sum += grid[1][index]

        # Return the minimum value found among the maximum sums after each possible move
        return min_max_score
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)` where `n` is the number of columns in the input `grid`. This is because there is a single loop that iterates through the elements of the first row of the grid. During each iteration, it performs constant-time operations: updating the summation variables `s1`, `s2` and computing the minimum of `ans` with `max(s1, s2)`.

Space Complexity

The space complexity of the given code is `O(1)`. No additional data structures that grow with the size of the input are being used. The variables `ans`, `s1`, `s2`, `j`, and `v` use a constant amount of space, irrespective of the input size.