1937. Maximum Number of Points with Cost

Medium | Array | Dynamic Programming | Leetcode Link

Problem Description

The problem presents a scenario where we are working with an `m x n` integer matrix called `points`. The goal is to maximize our total points by selecting one cell from each row of the matrix. When we select a cell at (r, c), we add the points value from `points[r][c]` to our score.

However, there's a catch: we must also account for the positional difference from the cells we select in adjacent rows. Specifically, if we choose a cell at (r, c1) and then a cell at (r+1, c2) in the next row, our score is reduced by `abs(c1 - c2)`, which represents the absolute difference between the column indices of the selected cells in consecutive rows.

The challenge is to figure out which cells to choose from each row to end up with the highest possible score after accounting for both the points from each cell and the penalties incurred for the distance between cells in adjacent rows.

Intuition

To solve this problem efficiently, dynamic programming can be used to avoid redundant calculations while considering each possible choice. The intuition behind the solution lies in the observation that the penalty for moving horizontally between columns can affect our choice of picking the optimal cell. We need to balance the points gained against the potential loss from moving too far horizontally in the next row.

The solution keeps track of an array `f`, which represents the best score we can achieve up to the current row for every possible column choice. The solution iterates over each row of the `points` matrix, calculates the best scores for choosing each column in that row, and prepares the array `f` for the next row.

There are two main parts in the iteration over each row:

1. Traverse left-to-right: calculating the value of `lmx`, which maintains the maximum score possible up to the current cell while moving from left to right.
2. Traverse right-to-left: calculating the value of `rmx`, which keeps track of the maximum score possible up to the current cell when moving from right to left.

These two sweeps are necessary because moving to a cell on the right might be advantageous when seen from one direction but not from the other. Therefore, for every cell, we calculate the best possible score from the `f` array value for that row taking into account the movement cost, both from the left and the right.

By the end of these two sweeps, we have an updated array `f` that contains the best score we can have after we have finished processing the current row.

In essence, the algorithm balances selecting cells with high values versus the cost of moving to each cell from the cells chosen in previous rows. After going through all the rows, the maximum value in the `f` array is the desired maximum number of points we can achieve.

Solution Approach

The provided solution implements a dynamic programming strategy that utilizes a single-dimensional array `f` to keep track of the maximum scores achievable for each column up to the current row.

Initially, `f` is a copy of the first row of the `points` matrix since there is no previous row to consider, and thus no penalty cost.

As we iterate over each subsequent row, we use two temporary arrays - `lmx` and `rmx` standing for "left max" and "right max," respectively. These arrays are actually not explicitly created in the code; rather they represent the maxima calculated on-the-fly during left-to-right and right-to-left sweeps.

Here's a step-by-step breakdown:

1. **Left-to-right sweep:** Iterate through the columns from left to right. We calculate the running `lmx`, which keeps track of the maximum score that could have been obtained from choosing any of the previous cells in the row (including the cost of moving to that cell). As we move right, we add the index `j` to the `f[j]` value because if we move further right in the next row, the cost will increase, which is why the cost `j` is proactively added to the score. For each cell, we calculate the possible score if we move there from the left as `p[j] + lmx - j` and store it in the temporary array `g`.
2. **Right-to-left sweep:** This is similar to the left-to-right sweep but in the opposite direction. We maintain a running `rmx` to track the maximum score that could have been obtained if moving from any previous cell to the right. While moving to the left, we subtract the index `j` when calculating rmx because the cost to move leftward in the next row will decrease. For each cell, we then calculate the possible score if we move there from the right as `p[j] + rmx + j`.
3. **Combining the results:** After both sweeps, for each column, the temporary array `g` holds the maximum of the scores obtainable from either direction. Then we update `f` with `g` to use in the next row's calculations.
4. **Return the maximum value:** Once we reach the last row, we've calculated the maximum possible scores for that row considering all rows above it. The final result is the maximum value in `f`.

To summarize, the pattern used here involves iteratively updating a dynamic programming table (`f`) while considering the movement cost in both directions (left-to-right and right-to-left). It elegantly captures the trade-offs between picking high-value cells and minimizing the penalty for moving between cells of different columns in adjacent rows. The sweeps are essential to factor in the cost impact from both sides before deciding on the best cells to choose.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Say we have the following `3 x 4` matrix called `points`:

```
points = [
  [1, 2, 2, 4],
  [3, 1, 2, 1],
  [4, 1, 1, 3]
]
```

Our goal is to choose one cell from each row to maximize the score, subtracting the absolute difference of column indices between chosen cells of consecutive rows.

Initialization

We start by copying the first row into our `f` array since there is no previous row to consider:

```
f = [1, 2, 2, 4]
```

Iterating over the second row

We will perform both left-to-right and right-to-left sweeps:

1. **Left-to-right sweep:**

Starting at `j=0`:

- `lmx` is `f[0]` since this is the first element in the `f` array. `lmx = 1`.
- For `g[0]`, we take `points[1][0] + lmx - 0 = 3 + 1 - 0 = 4`.

Continuing to `j=1` and so on:

- We update `lmx` to `max(lmx, f[j] - j)`.
- Calculate `g[j]` as `points[1][j] + lmx - j`.

After the sweep, `lmx` values are calculated and they are `[4, 4, 4, 4]`.

2. **Right-to-left sweep:**

Starting at `j=3` and going left:

- `rmx` is initialized to `f[3] - 3 = 4 - 3 = 1`.
- For `g[3]`, since g already has a value from the left-to-right sweep, we take `max(g[3], points[1][3] + rmx + 3)`.

Continuing to `j=2` and so on:

- We update `rmx` to `max(rmx, f[j] + j)`.
- Update `g[j]` to `max(g[j], points[1][j] + rmx + j)`.

After the sweep, the combined maximum `g` values are `[4, 4, 4, 5]`.

Iterating over the third and final row

Repeat the same steps as above. After the left-to-right and right-to-left sweeps, the final `g` values might look like `[8, 8, 7, 8]`.

Conclusion

After processing all rows, `g` holds the maximum scores possible for each cell in the last row including the penalties for moving between columns. The maximum score of `g` is our answer, which in this case is `8`.

Thus, our dynamic programming approach has allowed us to navigate through the matrix, weighing the value of each point against the penalty for moving between columns, to find the maximum total score we can achieve.

Python Solution

```python
from typing import List

class Solution:
    def maxPoints(self, points: List[List[int]]) -> int:
        # Get the number of columns in the points matrix.
        num_columns = len(points[0])

        # Initialize dp array with the first row of points.
        dp = points[0][:]

        # Iterate over each row in points starting from the second row.
        for row in points[1:]:
            # Create a new row for dp, to store the maximum points for the current row.
            new_dp = [0] * num_columns

            # Initialize left max value, this tracks the maximum from the left side including the current position.
            left_max = float('-inf')
            # Left pass: Fill in the new_dp array considering the left side of each column.
            for j in range(num_columns):
                left_max = max(left_max, dp[j] + j)
                new_dp[j] = max(new_dp[j], row[j] + left_max - j)

            # Initialize right max value, this tracks the maximum from the right side including the current position.
            right_max = float('-inf')
            # Right pass: Update the new_dp array considering the right side of each column.
            for j in range(num_columns - 1, -1, -1):
                right_max = max(right_max, dp[j] - j)
                new_dp[j] = max(new_dp[j], row[j] + right_max + j)

            # Update dp array with the new_dp for the next iteration.
            dp = new_dp

        # Return the maximum points that can be collected, which is the max value in the last dp row.
        return max(dp)

```

Java Solution

```java
class Solution {
    public long maxPoints(int[][] points) {
        // 'n' denotes the length of columns in 'points'
        int n = points[0].length;

        // 'currentRowMaxPoints' stores the maximum points achievable for the current row
        long[] currentRowMaxPoints = new long[n];

        // 'infinity' is a large number to be considered as negative infinity
        final long infinity = 1L << 60;

        // Loop through all the rows of the 'points' matrix
        for (int[] row : points) {
            // 'nextRowMaxPoints' is a temporary array to store the maximum points for the next row
            long[] nextRowMaxPoints = new long[n];

            // 'leftMax' tracks the maximum points from the left up to the current position
            long leftMax = -infinity;

            // 'rightMax' tracks the maximum points from the right up to the current position
            long rightMax = -infinity;

            // Forward pass to calculate the max points when coming from the left
            for (int j = 0; j < n; ++j) {
                leftMax = Math.max(leftMax, currentRowMaxPoints[j] + j);
                nextRowMaxPoints[j] = Math.max(nextRowMaxPoints[j], row[j] + leftMax - j);
            }

            // Backward pass to calculate the max points when coming from the right
            for (int j = n - 1; j >= 0; --j) {
                rightMax = Math.max(rightMax, currentRowMaxPoints[j] - j);
                nextRowMaxPoints[j] = Math.max(nextRowMaxPoints[j], row[j] + rightMax + j);
            }

            // Update 'currentRowMaxPoints' with the max points so far for the next iteration
            currentRowMaxPoints = nextRowMaxPoints;
        }

        // 'maxPoints' will store the overall maximum from 'currentRowMaxPoints'
        long maxPoints = 0;
        for (long pointsValue : currentRowMaxPoints) {
            maxPoints = Math.max(maxPoints, pointsValue);
        }

        // The final answer is the maximum points we can collect from the matrix
        return maxPoints;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    long long maxPoints(std::vector<std::vector<int>>& pointsMatrix) {
        // Alias for long long type to be used later in the function
        using ll = long long;

        // Number of columns in the input matrix
        int n = pointsMatrix[0].size();

        // Initial row (f) will hold the maximum points for each column
        std::vector<ll> prevRow(n);

        // Representing infinity as a large number, useful to ensure
        // that if a path includes an unfeasible option, it is ignored
        const ll inf = 1e18;

        // Loop over each row in the points matrix
        for (auto& points : pointsMatrix) {
            // Temporary row (g) to calculate new scores considering the transition
            std::vector<ll> currentRow(n);

            // Variables to track the left and right maximum of previous row
            ll leftMax = -inf, rightMax = -inf;

            // Forward iteration: calculate max points from the left to the current position
            for (int j = 0; j < n; ++j) {
                leftMax = std::max(leftMax, prevRow[j] + j);
                currentRow[j] = std::max(currentRow[j], points[j] + leftMax - j);
            }

            // Backward iteration: calculate max points from the right to the current position
            for (int j = n - 1; j >= 0; --j) {
                rightMax = std::max(rightMax, prevRow[j] - j);
                currentRow[j] = std::max(currentRow[j], points[j] + rightMax + j);
            }

            // Update the previous row to the current row for next iteration
            prevRow = std::move(currentRow);
        }

        // Return the maximum points that can be achieved
        return *std::max_element(prevRow.begin(), prevRow.end());
    }
};

```

Typescript Solution

```typescript
function maxPoints(points: number[][]): number {
    // n represents the number of columns in the points grid.
    const columns = points[0].length;
    // Initialize the dynamic programming array to store maximum points.
    const dp: number[] = new Array(columns).fill(0);

    // Iterate through each row of points.
    for (const rowPoints of points) {
        // Initialize a temporary array to store the current row's maximum points.
        const currentMax: number[] = new Array(columns).fill(0);
        // Variables to keep track of the left and right maximum with modifiers.
        let leftMax = -Infinity;
        let rightMax = -Infinity;

        // Traverse from left to right to calculate the leftMax contributions.
        for (let j = 0; j < columns; ++j) {
            // Update leftMax by considering the previously calculated dp.
            leftMax = Math.max(leftMax, dp[j] + j);
            // Calculate maximum points for the current position from the left side.
            currentMax[j] = Math.max(currentMax[j], rowPoints[j] + leftMax - j);
        }

        // Traverse from right to left to calculate the rightMax contributions.
        for (let j = columns - 1; j >= 0; --j) {
            // Update rightMax by considering the previously calculated dp.
            rightMax = Math.max(rightMax, dp[j] - j);
            // Calculate maximum points for the current position from the right side.
            currentMax[j] = Math.max(currentMax[j], rowPoints[j] + rightMax + j);
        }

        // Prepare dp for the next iteration by copying the values from currentMax.
        dp.splice(0, columns, ...currentMax);
    }

    // After processing all rows, the maximum points will be the max value in dp.
    return Math.max(...dp);
}

```

Time and Space Complexity

The provided code is designed to solve the problem of finding the maximum points you can obtain by walking on a grid, where `points[i][j]` represents the points you get from the i-th row and j-th column. The algorithm maintains two arrays `f` and `g` to store the intermediate results.

Time Complexity

The time complexity of the code can be determined as follows:

- There is a nested loop structure where the outer loop iterates over the rows of the input `points` list, and the inner loops iterate over the columns.
- The outer loop runs `len(points) - 1` times because it starts from the second row in `points`. Since the inner loops iterate `n` times each (where `n` is the number of columns which is `len(points[0])`), we have two separate iterations for each row - one from left to right and the other from right to left.
- Inside each iteration of the inner loops, a constant time operation is performed to compute the maximum points.

Therefore, the time complexity is `O(m * n)` where `m` is the number of rows in `points` and `n` is the number of columns.

Space Complexity

The space complexity of the algorithm is determined as follows:

- It uses two additional arrays `f` and `g`, both of size `n` (`len(points[0])`), to store intermediate results.
- No other significant data structures are used that grow with the size of the input.

Hence, the space complexity is `O(n)`.

In summary, the time complexity is `O(m * n)` and the space complexity is `O(n)` where `m` is the number of rows and `n` is the number of columns in the `points` list.