2931. Maximum Spending After Buying Items

Hard Greedy Array Matrix Sorting Weekly Contest 370 Heap (Priority Queue)

Problem Description

In this problem, we are given a 0-indexed matrix `values` of integers representing the prices of different items in several shops. Each shop carries `n` items, with the prices sorted in non-increasing order—that is, each item's price is greater than or equal to the price of the next item in the shop.

Our task is to buy all `m * n` items in such a way that across `m * n` days, we spend the maximum amount of money possible. On each day `d`, we choose one shop and buy the rightmost available item (the one that has not been purchased yet) from that shop at a price of `values[i][j] * d`. The goal is to find a strategy that allows us to maximize the total money spent.

Intuition

The key intuition behind solving this problem is that we must buy cheaper items earlier and save the expensive ones for later, as the cost to buy an item increases with each passing day. Since we multiply the base price of an item by the current day to get the final price, we want to minimize the base price for earlier purchases.

Hence, we should start by buying the smallest available item from any shop on day 1. This can be easily done by using a priority queue (min-heap), where we can efficiently keep track of the smallest item available across all shops.

Initially, we push the last (and smallest) item of each shop into the priority queue. Each day, we pop the smallest item from the priority queue and calculate the price. We then multiply this price by the day number to get the final cost and add it to the total spending. After buying an item from a shop, we push the next smallest item from the same shop (moving one step to the left) into the priority queue.

This process continues until we have purchased all items, i.e., when the priority queue is empty.

Solution Approach

The solution approach utilizes a greedy algorithm combined with a priority queue (min-heap) to efficiently select the right item to purchase each day. Here's a step-by-step breakdown of the implementation:

1. **Initialization**: We create a priority queue to store the available items from each shop along with the shop index and the item index. We only store the smallest (rightmost) value from each shop initially, i.e., `values[i][len(values[i]) - 1]` along with `i` (the shop index) and `len(values[i]) - 1` (the item index).
2. **Heapify**: Convert the list of minimum items into a min-heap using the `heapify` function from Python's `heapq` module. This ensures that the item with the smallest value is at the front of the priority queue.
3. **Buying Items**: With the priority queue set up, we start buying items. For each day `d` starting from 1, we:
   - Pop the top element from the min-heap using `heappop`, which gives us the item with the smallest value that has not been purchased yet, along with its shop and item indices (`v, i, j`).
   - Calculate the cost of the item for the given day as `cost = value * day`, and add this cost to the total amount `ans`.
   - Check if the bought item was not the only item left in the shop. If there are items left (i.e., `j > 0`), push the next smallest item from the same shop (one index to the left) into the priority queue using `heappush`.
4. **Repeating the Process**: Continue this process, incrementing `d` by 1 each day, until all items have been purchased. This happens when the priority queue is empty, since we remove an item from the queue each day.
5. **Returning the Result**: Once the priority queue is empty, all items have been purchased, and the process terminates. We return the total amount `ans` as the final result.

This solution guarantees to spend the maximum amount of money because it always buys the cheapest available item each day and leverages the non-decreasing cost multiplier—by buying the more expensive items later, they will cost significantly more and lead to a higher total expenditure.

The time complexity of this approach is primarily dictated by the operations on the priority queue, which are `O(log(m))` for both insertion and removal. Since we perform these operations `m * n` times, the overall time complexity is `O(m * n * log(m))`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have two shops (m=2) and each shop carries three items (n=3). Here are the prices for each shop in non-increasing order:

```
Shop 1: [5, 4, 1]
Shop 2: [6, 3, 2]
```

Following the solution approach step by step:

1. **Initialization**: Create a priority queue and push the smallest value from each shop:
   - From Shop 1, push (1, 0, 2) where 1 is the value, 0 is the shop index, and 2 is the item index.
   - From Shop 2, push (2, 1, 2).
   The priority queue is now: `[(1, 0, 2), (2, 1, 2)]`.
2. **Heapify**: The priority queue is already a min-heap, so the element with the smallest value is at the front.
3. **Buying Items**: Now we start purchasing items, one for each day.
   - Day 1: Pop the smallest item (`(1, 0, 2)`), calculate the cost as `1 * 1 = 1`, and add it to the total amount. Then, push the next item from Shop 1, which is `(4, 0, 1)` into the priority queue.
   - Day 2: Pop the smallest item (`(2, 1, 2)`), calculate the cost as `2 * 2 = 4`, and add it to the total amount. Then, push the next item from Shop 2, which is `(3, 1, 1)` into the priority queue.
   - Day 3: Pop the now smallest item (`(3, 1, 1)`), calculate the cost as `3 * 3 = 9`, and add it to the total amount. Then, push the next item from Shop 2, which is `(6, 1, 0)` into the priority queue.
   - Day 4: Pop the smallest item (`(4, 0, 1)`), calculate the cost as `4 * 4 = 16`, and add it to the total amount. Then, push the next item from Shop 1, which is `(5, 0, 0)` into the priority queue.
   - Day 5: Pop the smallest item (`(5, 0, 0)`), calculate the cost as `5 * 5 = 25`, and add it to the total amount. Shop 1 is now empty, so there's nothing more to add to the queue from this shop.
   - Day 6: Pop the smallest item (`(6, 1, 0)`), calculate the cost as `6 * 6 = 36`, and add it to the total amount. Shop 2 is now empty as well.
4. **Repeating the Process**: The process is now complete as we have purchased all items and the priority queue is empty.
5. **Returning the Result**: The total amount spent is `1 + 4 + 9 + 16 + 25 + 36 = 91`.

In this example, you can see how each day we buy the cheapest available item and delay purchasing the more expensive items until later days. This maximizes the total money spent because of the increasing multiplier effect applied to the base prices of the items with each subsequent day.

Solution Implementation

Python

```python
from typing import List
import heapq  # Using the heapq module for a priority queue.

class Solution:
    def maxSpending(self, values: List[List[int]]) -> int:
        # Initialize the number of columns.
        column_count = len(values[0])

        # Create a priority queue with tuples containing the last value
        # of each row, the row index, and the last column index.
        priority_queue = [(row[-1], row_idx, column_count - 1) for row_idx, row in enumerate(values)]

        # Transform the list into a heap.
        heapq.heapify(priority_queue)

        # Initialize the answer and day count.
        total_spending = 0
        day_count = 1

        # While there are elements in the priority queue:
        while priority_queue:
            # Increment the day count.
            day_count += 1

            # Get the next highest value, along with its row and column index.
            value, row_idx, col_idx = heapq.heappop(priority_queue)

            # Add the value multiplies by the current day to the total spending.
            total_spending += value * day_count

            # If there's a previous column, push the value from the previous
            # column of the same row back into the priority queue.
            if col_idx:
                heapq.heappush(priority_queue, (values[row_idx][col_idx - 1], row_idx, col_idx - 1))

        # Return the total spending.
        return total_spending
```

Java

```java
import java.util.PriorityQueue;

public class Solution {
    public long maxSpending(int[][] values) {
        int rows = values.length; // Number of rows in the input array
        int cols = values[0].length; // Number of columns in the input array

        // Priority queue to store the values along with their respective row and column indices
        // The queue is sorted in ascending order based on the values
        PriorityQueue<int[]> priorityQueue = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // Offer the last element of each row into the priority queue
        for (int i = 0; i < rows; ++i) {
            priorityQueue.offer(new int[] {values[i][cols - 1], i, cols - 1});
        }

        long totalSpending = 0; // Initialize variable to keep track of the total spending.

        // Indexing variable to rearesent the 'days'
        for (int day = 1; !priorityQueue.isEmpty(); ++day) {
            int[] current = priorityQueue.poll(); // Retrieve and remove the smallest element in the priority queue
            int value = current[0], row = current[1], col = current[2];

            // Add the value multiplied by the day index to the total spending
            totalSpending += (long) value * day;

            // If we are not at the first column, offer the previous element in the row to the priority queue
            if (col > 0) {
                priorityQueue.offer(new int[] {values[row][col - 1], row, col - 1});
            }
        }

        // Return the calculated total spending
        return totalSpending;
    }
}
```

C++

```cpp
#include <vector>
#include <queue>
#include <tuple>

class Solution {
public:
    long long maxSpending(vector<vector<int>>& values) {
        // A min-heap priority queue that stores tuples in ascending order by their first element.
        // Each tuple contains a value, a row index, and a column index.
        priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<tuple<int, int, int>>> minHeap;

        // Get the dimensions of the `values` matrix.
        int rowCount = values.size();
        int columnCount = values[0].size();

        // Initialize the priority queue with the last column's values from the matrix.
        for (int row = 0; row < rowCount; ++row) {
            minHeap.emplace(values[row][columnCount - 1], row, columnCount - 1);
        }

        long long totalSpending = 0; // This will hold the calculated maximum spending.

        // Loop to process all values from the priority queue.
        for (int day = 1; !minHeap.empty(); ++day) {
            // Get the top (smallest) element from the priority queue.
            auto [value, i, j] = minHeap.top();
            minHeap.pop();

            // Accumulate the spending by multiplying value with the day count.
            totalSpending += static_cast<long long>(value) * day;

            // If there is a previous column, add the value from the previous column to the priority queue.
            if (j > 0) {
                minHeap.emplace(values[i][j - 1], i, j - 1);
            }
        }

        return totalSpending; // Return the computed total spending.
    }
};
```

TypeScript

```typescript
// Define a type for the PriorityQueue entry to improve clarity.
type PriorityQueueEntry = [number, number, number];

function maxSpending(values: number[][]): number {
    // The number of rows in the `values` array
    const rows = values.length;
    // The number of columns in the `values` array
    const columns = values[0].length;
    // Initialize a priority queue with a comparator for sorting based on the value
    const priorityQueue = new PriorityQueue<PriorityQueueEntry>({ compare: (a, b) => a[0] - b[0] });

    // Enqueue the last value of each row along with its row and column index
    for (let i = 0; i < rows; ++i) {
        priorityQueue.enqueue([values[i][columns - 1], i, columns - 1]);
    }

    let totalSpending = 0; // Initialize a variable to keep track of the total spending
    // Process the queue until it is empty
    for (let day = 1; !priorityQueue.isEmpty(); ++day) {
        // Dequeue the top element (lowest value)
        const [value, rowIndex, columnIndex] = priorityQueue.dequeue()!;
        // Accumulate the total spending, weighting it by the day
        totalSpending += currentValue * day;
        // If there's a previous column, enqueue the value from the previous column (to the left)
        if (columnIndex > 0) {
            priorityQueue.enqueue([values[rowIndex][columnIndex - 1], rowIndex, columnIndex - 1]);
        }
    }

    return totalSpending; // Return the total spending after processing all elements
}
```

```python
from typing import List
import heapq  # Using the heapq module for a priority queue.

class Solution:
    def maxSpending(self, values: List[List[int]]) -> int:
        # Initialize the number of columns.
        column_count = len(values[0])

        # Create a priority queue with tuples containing the last value
        # of each row, the row index, and the last column index.
        priority_queue = [(row[-1], row_idx, column_count - 1) for row_idx, row in enumerate(values)]

        # Transform the list into a heap.
        heapq.heapify(priority_queue)

        # Initialize the answer and day count.
        total_spending = 0
        day_count = 0

        # While there are elements in the priority queue:
        while priority_queue:
            # Increment the day count.
            day_count += 1

            # Get the next highest value, along with its row and column index.
            value, row_idx, col_idx = heapq.heappop(priority_queue)

            # Add the value multiplies by the current day to the total spending.
            total_spending += value * day_count

            # If there's a previous column, push the value from the previous
            # column of the same row back into the priority queue.
            if col_idx:
                heapq.heappush(priority_queue, (values[row_idx][col_idx - 1], row_idx, col_idx - 1))

        # Return the total spending.
        return total_spending
```

Time and Space Complexity

The time complexity of the given code is `O(m * n * log m)`. The code begins with initializing a priority queue with all the last elements from each row, which will take `O(m)` (since all the last elements in the rows are being pushed at once). After this, the while loop pops from the priority queue, does a constant amount of work, and potentially pushes another element onto the priority queue. This will happen `O(n)` times for each row, because we add `n` elements for each of the `m` rows. Each push or pop operation on the priority queue is `O(log m)` since there are at most `m` elements in the priority queue. Therefore, since we pop and then possibly push for each of the `n` column indexes and for each of the `m` rows, the total time complexity is indeed `O(m * n * log m)`.

The space complexity is `O(m)` because, at any point, the priority queue holds at most one element from each of the `m` rows, regardless of the number of columns `n`. Therefore, the maximum size of the data structure that varies with the input size is proportional to `m`, leading to a space complexity of `O(m)`.