1981. Minimize the Difference Between Target and Chosen Elements

Medium | Array | Dynamic Programming | Matrix

Problem Description

In this problem, we are given a matrix `mat` which has `m` rows and `n` columns, and a target integer called `target`. Our goal is to choose exactly one number from each row of the matrix such that the sum of these numbers is as close as possible to the target. We need to minimize the absolute difference between this sum and the target. Absolute difference means we consider the positive distance between two numbers, regardless of their order. The output should be this minimum absolute difference.

Intuition

To solve this problem efficiently, we need to keep track of all possible sums that can be generated by choosing one number from each row up to the current row being processed. Here, dynamic programming comes into play. We start with an initial set that contains only one element, 0, representing the sum before any numbers have been chosen. For each row in the matrix, we update this set by adding each element of the current row to each of the sums we have accumulated so far.

This way, by the time we finish the last row, our set contains all possible sums that could be generated by picking one number from each row. After this, to find the minimum absolute difference, we simply iterate over these possible sums and calculate the absolute difference between each of them and the target. The smallest absolute difference found during this process is the result we're looking for.

Although this approach seems to have a high computational complexity due to generating all possible sums, in practice, we can optimize this by trimming the set of sums to only include relevant sums that could lead to a result close to the target. This is not shown in the given code, but it is a useful optimization for very large matrices.

Solution Approach

The solution uses a straightforward dynamic programming approach. Let's walk through the algorithms, data structures, or patterns used in the provided solution code:

1. Initialization of a Set: The variable `f` is initialized as a set with one element, `0`. This set represents all possible sums we've seen so far, starting with no numbers having been chosen.

```
f = {0}
```

2. Iteration Over Rows: The outer loop goes through each row of the matrix:

```
for row in mat:
```

3. Generating New Sums: Inside the loop, for each row, we generate new sums by adding every element in the current row to each of the sums in `f`. This is done using a set comprehension:

```
f = set(a + b for a in f for b in row)
```

   Here, for every sum `a` existing in `f` and for each element `b` in the current row, we add `a + b` to the new set. This set comprehension is executed for each row, effectively updating `f` with all possible sums after choosing an element from each row so far.

4. Finding Minimum Absolute Difference: After processing all rows, the final set `f` encompasses all possible sums that could be made by picking one number from each row. The task now is to find the sum that is closest to the target, thus minimizing the absolute difference. This is achieved by iterating over each value `v` in the final set `f`:

```
return min(abs(v - target) for v in f)
```

   We calculate the absolute difference `abs(v - target)` and use the `min` function to find the smallest difference, which is our desired result.

The solution effectively leverages Python's set datatype to avoid duplicate sums, ensuring that only unique sums are generated and subsequently considered when calculating the minimum absolute difference from the target. This approach, due to the nature of the problem, may not be the most efficient in terms of time or space complexity especially if there are many large numbers in the rows, but it provides a correct and easily understandable solution.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Assume we have the following matrix `mat` with `m` = 3 rows and `n` = 3 columns, and our target integer is `target` = 7.

```
mat = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
```

Following the steps of the solution approach:

1. **Initialization of a Set**: We initialize our set `f` with a single element `0`.

```
f = {0}
```

2. **Iteration Over Rows - Row 1**: The first row is `[1, 2, 3]`. We add each number in this row to all elements in set `f`.

   New sums will be:

   - 0 + 1 = 1
   - 0 + 2 = 2
   - 0 + 3 = 3

   So, `f` now becomes `{1, 2, 3}`.

3. **Iteration Over Rows - Row 2**: The second row is `[4, 5, 6]`. Now, we add each number in this row to each sum in the current set `f`.

   New sums will be:

   - 1 + 4 = 5, 1 + 5 = 6, 1 + 6 = 7
   - 2 + 4 = 6, 2 + 5 = 7, 2 + 6 = 8
   - 3 + 4 = 7, 3 + 5 = 8, 3 + 6 = 9

   After adding these amounts, `f` now becomes `{5, 6, 7, 8, 9}`.

4. **Iteration Over Rows - Row 3**: The third row is `[7, 8, 9]`. Again, add each number in this row to each sum in the current set `f`.

   Calculating the new sums, we get a multitude of values, but we're only interested in the unique ones:

   - 5 + 7 = 12, 5 + 8 = 13, 5 + 9 = 14
   - 6 + 7 = 13, 6 + 8 = 14, 6 + 9 = 15
   - 7 + 7 = 14, 7 + 8 = 15, 7 + 9 = 16
   - and so on for sums 8 and 9...

   The resulting set `f` now consists of `{12, 13, 14, 15, 16, ...}` along with other sums from the additions involving 8 and 9 from the third row.

5. **Finding Minimum Absolute Difference**: Now we have a set of all possible sums. We must find the one which is closest to our `target` of 7.

   The absolute differences between each sum `v` in set `f` and the target are calculated as `abs(v - 7)`. The minimum difference here can be seen to be 0, which occurs for sum 7 (which is directly in our set after adding the first two rows).

   The closest sum to the target is the sum itself when it is present in the set. So, the solution returns 0, as this is the smallest absolute difference we can achieve.

Solution Implementation

Python

```python
class Solution:
    def minimizeTheDifference(self, mat: List[List[int]], target: int) -> int:
        # Initialize a set containing just the element 0
        # This set will hold all possible sums up till the current row
        possible_sums = {0}

        # Iterate over each row in the matrix
        for row in mat:
            # Use a set comprehension to generate all possible sums
            # by adding each element in the current row to all possible sums calculated in the previous step
            # This ensures we only consider sums that include exactly one element from each row
            possible_sums = {elem + row_elem for elem in possible_sums for row_elem in row}

        # Find the minimum absolute difference between any possible sum and the target
        # This is done by iterating over all possible sums and computing the absolute difference with the target
        min_difference = min(abs(sum_val - target) for sum_val in possible_sums)

        # Return the minimum difference found
        return min_difference
```

Java

```java
class Solution {
    public int minimizeTheDifference(int[][] mat, int target) {
        // Initialize an array to keep track of possible sums using the first row.
        boolean[] possibleSums = {true};

        // Loop over each row in the matrix.
        for (int[] row : mat) {
            // Variable to keep track of the maximum element in the current row.
            int maxElement = 0;
            for (int element : row) {
                maxElement = Math.max(maxElement, element);
            }

            // Extend the possible sums array to accommodate the new maximum element.
            boolean[] newPossibleSums = new boolean[possibleSums.length + maxElement];

            // Update new possible sums by adding each element in the current row
            // to the sums calculated so far.
            for (int element : row) {
                for (int sumIndex = element; sumIndex < possibleSums.length + element; ++sumIndex) {
                    newPossibleSums[sumIndex] |= possibleSums[sumIndex - element];
                }
            }

            // Move to the next set of possible sums.
            possibleSums = newPossibleSums;
        }

        // Initialize the answer to a large number (infinity equivalent).
        int minimumDifference = Integer.MAX_VALUE;

        // Find the smallest difference between any possible sum and the target.
        for (int sumIndex = 0; sumIndex < possibleSums.length; ++sumIndex) {
            if (possibleSums[sumIndex]) {
                minimumDifference = Math.min(minimumDifference, Math.abs(sumIndex - target));
            }
        }

        // Return the minimum difference found.
        return minimumDifference;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum absolute difference between the sum of elements from each row and the target value.
    int minimizeTheDifference(vector<vector<int>>& mat, int target) {
        // Initialize a vector 'possibleSums' representing possible sums with just one element set to 1 (meaning sum 0 is possible).
        vector<int> possibleSums = {1};

        // Iterate over each row in the matrix.
        for (auto& row : mat) {
            // Find the maximum element in the current row to determine the new size of 'nextPossibleSums'.
            int maxElementInRow = *max_element(row.begin(), row.end());
            vector<int> nextPossibleSums(possibleSums.size() + maxElementInRow);

            // Compute the next state of possible sums by adding each element of the current row to the 'possibleSums'.
            for (int element : row) {
                for (int j = element; j < possibleSums.size() + element; ++j) {
                    nextPossibleSums[j] |= possibleSums[j - element];
                }
            }
            // Move 'nextPossibleSums' to 'possibleSums' to use in the next iteration.
            possibleSums = move(nextPossibleSums);
        }

        // Initialize 'minDifference' to a large number to find the minimum difference.
        int minDifference = 1 << 30; // Equivalent to a large number using bit left shift.

        // Iterate over all possible sums to calculate the smallest difference to 'target'.
        for (int sum = 0; sum < possibleSums.size(); ++sum) {
            if (possibleSums[sum]) {
                minDifference = min(minDifference, abs(sum - target));
            }
        }

        // Return the minimum absolute difference found.
        return minDifference;
    }
};
```

TypeScript

```typescript
// Function to find the minimum element in an array using the reduce method.
function maxElement(array: number[]): number {
    return array.reduce((max, currentValue) => Math.max(max, currentValue));
}

// Function to find the minimum absolute difference between the sum of elements from each row and the target value.
function minimizeTheDifference(mat: number[][], target: number): number {
    // Initialize an array 'possibleSums' representing possible sums with just zero sum possible initially.
    let possibleSums: number[] = [1];

    // Iterate over each row in the matrix.
    mat.forEach(row => {
        // Find the maximum element in the current row to determine the new size of 'nextPossibleSums'.
        const maxElementInRow = maxElement(row);
        const nextPossibleSums: number[] = new Array(possibleSums.length + maxElementInRow).fill(0);

        // Compute the next state of possible sums by adding each element of the current row to the 'possibleSums'.
        row.forEach(element => {
            for (let j = element; j < possibleSums.length + element; ++j) {
                if (possibleSums[j - element]) {
                    nextPossibleSums[j] = 1;
                }
            }
        });

        // Update 'possibleSums' with newly computed 'nextPossibleSums'.
        possibleSums = nextPossibleSums;
    });

    // Initialize 'minDifference' with a large number to ensure it is higher than any possible sum.
    let minDifference = Infinity;

    // Iterate over all possible sums to calculate the smallest difference to the 'target'.
    possibleSums.forEach((value, sum) => {
        if (value) {
            minDifference = Math.min(minDifference, Math.abs(sum - target));
        }
    });

    // Return the minimum absolute difference found.
    return minDifference;
}
```


```python
class Solution:
    def minimizeTheDifference(self, mat: List[List[int]], target: int) -> int:
        # Initialize a set containing just the element 0
        # This set will hold all possible sums up till the current row
        possible_sums = {0}

        # Iterate over each row in the matrix
        for row in mat:
            # Use a set comprehension to generate all possible sums
            # by adding each element in the current row to all possible sums calculated in the previous step
            # This ensures we only consider sums that include exactly one element from each row
            possible_sums = {elem + row_elem for elem in possible_sums for row_elem in row}

        # Find the minimum absolute difference between any possible sum and the target
        # This is done by iterating over all possible sums and computing the absolute difference with the target
        min_difference = min(abs(sum_val - target) for sum_val in possible_sums)

        # Return the minimum difference found
        return min_difference
```


Time and Space Complexity

The time complexity of the given code can be understood by analyzing the nested loops and the set comprehension. In the worst-case scenario, the set `f` contains all possible sums that can be formed by adding the elements from each previous row of `mat` to the existing sums in `f`. If the maximum number of elements a row can contribute to the set is `m`, and there are `n` rows, the set can grow by a factor of up to `m` with each iteration (each new row of `mat`). Therefore, in the worst case, the time complexity is `O(m^n)`. However, since each element of `f` represents a unique sum and `m` can be large, this is still a considerable amount of computation.

The space complexity is influenced by the size of the set `f`. At any given point, `f` holds all unique sums that can be formed with the current rows of `mat` being processed. In the worst case, `f` can grow to `O(m^n)` unique sums. Therefore, the space complexity is also `O(m^n)`.

To sum up:

- Time Complexity: `O(m^n)`
- Space Complexity: `O(m^n)`