1380. Lucky Numbers in a Matrix

Problem Description

The problem provides us with a matrix of distinct numbers that is `m` rows by `n` columns in size. We are asked to identify all the "lucky numbers" in this matrix. A lucky number is defined as one that is the smallest number in its row and also the largest number in its column. The challenge is to return a list of these lucky numbers in any order.

To approach this problem, we need to iterate through each element in the matrix to compare it with other elements in the same row and column. The uniqueness of each number simplifies the task since there will be no duplicates to consider.

Intuition

The intuition behind the proposed solution is to optimize the search for lucky numbers by avoiding unnecessary comparisons. Here's the thought process:

- For each row in the matrix, we find the smallest element since a lucky number must be the minimum in its row.
- Similarly, we determine the largest element for each column, as a lucky number must also be the maximum in its column.

With these two lists of minimum row values and maximum column values, we can find the intersection set that contains elements that are both the minimum in their row and the maximum in their column. These are our "lucky numbers."

The implementation uses Python's list comprehensions and set intersection to achieve this efficiently:

- To find the row minimums, we generate a set by iterating through each row and apply the `min` function.
- To find the column maximums, we generate another set by transposing the matrix with `zip(*matrix)` and again use the `max` function for each column.
- Finally, by intersecting these two sets, we get the lucky numbers.

The solution assumes that the rows and columns property of the matrix object are the minimum and maximum functions applied as shown in the solution.

Solution Approach

The solution uses Python list comprehensions and the `min` and `max` functions to create two sets, one containing the minimum element of each row and the other containing the maximum element of each column. These two sets are then intersected to find the lucky numbers.

Here's how the code works:

1. `rows = min(row for row in matrix)` – This line iterates over each row in the matrix and finds the smallest element in that row. The `min` function applies within the comprehension to each row. This results in a set of minimum values, one for each row.
2. `cols = max(col for col in zip(*matrix))` – This line transposes the matrix using `zip(*matrix)`, which groups elements from each row into columns. By iterating over this transposed matrix, the code finds the maximum element in each of the original columns. This creates a set of the maximum column values.
3. `return list(rows & cols)` – Finally, the intersection (`&`) of the two sets is determined, which gives us the lucky numbers. Since a lucky number is defined as being both the minimum in its row and maximum in its column, intersection checks which values are present in both sets. Only those that meet both criteria will be in the result. The intersection is then converted back to a list before being returned.

The algorithm is efficient because it takes linear time with respect to the elements in the matrix ($O(mn)$, where m is the number of rows and n is the number of columns). This is far more efficient than a brute force approach that would require a comparison of each element with every other element in its row and column.

The use of sets for storing the minimum row values and maximum column values is a crucial part of the approach because it allows for the intersection operation to be performed swiftly.

Note: The solution given seems to assume that `rows` and `cols` would be sets. However, the current implementation with list comprehensions actually creates lists. To get the correct result, the implementation would need to be corrected to use the set of minimum row values and the set of maximum column values before the intersection. The implementation as stated would cause a `TypeError` because Python lists do not support intersection. Here's the corrected implementation:

```python
class Solution:
    def luckyNumbers(self, matrix: List[List[int]]) -> List[int]:
        rows = {min(row) for row in matrix}  # Use a set comprehension
        cols = {max(col) for col in zip(*matrix)}  # Use a set comprehension
        return list(rows & cols)
```

With this adjustment, the solution should work correctly, finding the lucky numbers in the matrix.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Consider the following matrix:

3	7	8
9	11	13
15	16	17

According to the problem description, we need to find the numbers that are the smallest in their row and the largest in their column.

First, we find the smallest number in each row:

- For the first row, the smallest number is `3`.
- For the second row, the smallest number is `9`.
- For the third row, the smallest number is `15`.

Now, we have the set of minimum row values: `{3, 9, 15}`.

Next, we find the largest number in each column by transposing the matrix:

- The transposed columns become `[3, 9, 15]`, `[7, 11, 16]`, and `[8, 13, 17]`.
- The largest number in the first transposed column is `15`.
- The largest number in the second transposed column is `16`.
- The largest number in the third transposed column is `17`.

From this, we get the set of maximum column values: `{15, 16, 17}`.

The final step is to find the intersection of these two sets:

- Our minimum row values set is `{3, 9, 15}`.
- Our maximum column values set is `{15, 16, 17}`.
- The intersection of these two sets is `{15}`.

Therefore, the lucky number in this matrix is `15` because it is the only number that satisfies both criteria, and our final result is `[15]`.

Python Solution

```python
class Solution:
    def luckyNumbers(self, matrix: List[List[int]]) -> List[int]:
        # Find the minimum element in each row and store as a set
        min_in_rows = {min(row) for row in matrix}

        # Transpose the matrix to work with columns as rows and find the maximum
        # element in each original column, also store as a set
        max_in_columns = {max(col) for col in zip(*matrix)}

        # Find the intersection (common elements) between the sets of minimum elements in rows and
        # the maximum elements in columns, which represents the "lucky numbers"
        lucky_numbers = list(min_in_rows & max_in_columns)

        # Return the list of lucky numbers
        return lucky_numbers
```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;

class Solution {

    // Method to find all lucky numbers in the matrix.
    // A lucky number is defined as a number which is the minimum of its row and maximum of its column.
    public List<Integer> luckyNumbers(int[][] matrix) {

        // m represents the number of rows
        // n represents the number of columns
        int m = matrix.length, n = matrix[0].length;

        // 'minInRows' will hold the minimum values for each row
        // 'maxInCols' will hold the maximum values for each column
        int[] minInRows = new int[m];
        int[] maxInCols = new int[n];

        // Initialize 'minInRows' with a maximum valid integer value
        Arrays.fillWithInRows, Integer.MAX_VALUE);

        // Nested loops to calculate minInRows and maxInCols
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++) {
                // Update the minimum value in the current row
                minInRows[i] = Math.minminInRows[i], matrix[i][j]);
                // Update the maximum value in the current column
                maxInCols[j] = Math.maxmaxInCols[j], matrix[i][j]);
            }

        // List to store all the lucky numbers found
        List<Integer> luckyNumbers = new ArrayList<>();

        // Nested loops to find common elements in 'minInRows' and 'maxInCols'
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // Check if the current number is a lucky number
                if (minInRows[i] == maxInCols[j]) {
                    // Add the number to the list of lucky numbers
                    luckyNumbers.addWithInRows[i]);
                }
            }
        }

        // Return the list of lucky numbers
        return luckyNumbers;
    }
}
```

C++ Solution

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

class Solution {
public:
    std::vector<int> luckyNumbers (std::vector<std::vector<int>>& matrix) {
        int numRows = matrix.size(); // Number of rows in the matrix
        int numCols = matrix[0].size(); // Number of columns in the matrix
        std::vector<int> minInRows(numRows, INT_MAX); // Initialize vector to store min values for every row
        std::vector<int> maxInCols(numCols, INT_MIN); // Initialize vector to store max values for every column

        // Find the minimum value in each row and the maximum value in each column
        for(int i = 0; i < numRows; ++i) {
            for(int j = 0; j < numCols; ++j) {
                // Update the min for the ith row if the current element is less than the stored min
                minInRows[i] = std::min(minInRows[i], matrix[i][j]);
                // Update the max for the jth column if the current element is greater than the stored max
                maxInCols[j] = std::max(maxInCols[j], matrix[i][j]);
            }
        }

        std::vector<int> luckyNumbers; // Vector to store the lucky numbers
        // Check for lucky numbers - values that are the minimum in their rows and maximum in their columns
        for(int i = 0; i < numRows; ++i) {
            for(int j = 0; j < numCols; ++j) {
                // If the number is equal to the min of its row and the max of its column, it's a lucky number
                if(matrix[i][j] == minInRows[i] && matrix[i][j] == maxInCols[j]) {
                    luckyNumbers.push_back(matrix[i][j]);
                }
            }
        }

        return luckyNumbers; // Return the list of lucky numbers
    }
};
```

Typescript Solution

```typescript
function luckyNumbers(matrix: number[][]): number[] {
  // Get the number of rows (m) and columns (n) from the matrix.
  const numRows = matrix.length;
  const numCols = matrix[0].length;

  // Initialize arrays to store the minimum values of each row and the maximum values of each column.
  const minRowValues: number[] = new Array(numRows).fill(Infinity);
  const maxColValues: number[] = new Array(numCols).fill(0);

  // Find the minimum and maximum values for rows and columns, respectively.
  for (let row = 0; row < numRows; ++row) {
    for (let col = 0; col < numCols; col++) {
      minRowValues[row] = Math.minminRowValues[row], matrix[row][col]);
      maxColValues[col] = Math.maxmaxColValues[col], matrix[row][col]);
    }
  }

  // Initialize array to store the lucky numbers.
  const luckyNumbers: number[] = [];

  // Check each element in the minimum row values against the maximum column values.
  // If any value is both the minimum in its row and the maximum in its column, it's considered lucky.
  for (let row = 0; row < numRows; ++row) {
    for (let col = 0; col < numCols; col++) {
      if (minRowValues[row] === maxColValues[col]) {
        luckyNumbers.push(minRowValues[row]);
      }
    }
  }

  // Return the array of lucky numbers found in the matrix.
  return luckyNumbers;
}
```

Time and Space Complexity

Time Complexity

The `luckyNumbers` method consists of several operations. Let's analyze them step by step:

1. `min(row for row in matrix)`: This operation computes the minimum of each row in the matrix. Since it examines each element of the row to find the minimum, this operation takes $O(n)$ time for each row, where n is the number of columns. Since there are m rows in the matrix, the total time taken for this step is $O(m * n)$.

2. `max(col for col in zip(*matrix))`: This operation first transposes the matrix using `zip(*matrix)`, which is $O(1)$ since it returns an iterator. Then, it computes the maximum for each column (originally row in the transposed matrix). Computing the maximum for one column takes $O(m)$, and since there are n columns, the total time for this step is $O(n * m)$.

3. `list(rows & cols)`: The intersection operation (`&`) between two sets happens in $O(min(len(rows), len(cols)))$ time in the average case. Since `rows` is a set with at most n elements (one for each row) and `cols` is a set with at most m elements (one for each column), this step will take $O(min(n, m))$ time.

Combining these steps, the dominant term is the one with $O(m * n)$, as this is the largest factor in the time complexity.

Therefore, the overall time complexity is $O(m * n)$.

Space Complexity

Now let's look at the space complexity:

1. `rows`: Stores the minimum element of each row, so it has at most n elements.
2. `cols`: Stores the maximum element of each column, so it has at most m elements.

The space taken by the `rows` and `cols` sets is $O(n)$ and $O(m)$ respectively. There is no additional significant space usage.

Hence, the total space complexity is $O(m + n)$, as we need to store the minimum and the maximum elements of the rows and columns separately.