1007. Minimum Domino Rotations For Equal Row

Medium Greedy Array

Leetcode Link

Problem Description

In this problem, we are given two arrays, `tops` and `bottoms`, each representing the top and bottom halves of a stack of domino tiles, where each tile may have numbers from 1 to 6. We have the option to rotate any number of dominos to make either all top or all bottom numbers the same. Our goal is to find the minimum number of rotations required to achieve this uniformity of numbers on either side. If it's impossible to reach a state where all numbers are the same on any one side, we must return -1.

Imagine a row of dominoes lined up on a table, where we can see the top and bottom numbers of each tile. We can flip any domino, swapping its top and bottom numbers, trying to get all the top numbers to match or all the bottom numbers to match. The challenge is to do this in the fewest flips possible or determine if it's not possible at all.

Intuition

To solve this problem efficiently, we employ a greedy strategy. The key insight is that if a uniform number can be achieved on either the top or bottom of the dominoes, that number must be the same as either the top or bottom of the first domino. This is because if the first domino does not contain the number we want to match everywhere, we can never make that number appear on both the first top and bottom by any rotation.

With this in mind, we define a function, `f(x)`, that calculates the minimum number of rotations required to make all the values in either `tops` or `bottoms` equal to the number `x`. We only need to consider two cases for `x`: the number on the top and the number on the bottom of the first domino.

The function `f(x)` works by counting how many times the number `x` appears on the top (`cnt1`) and how many times on the bottom (`cnt2`). If the number `x` is not present in either side of a domino, it means we cannot make all values equal to `x` with rotations alone, and we return infinity (`inf`) to indicate that it is impossible for `x`.

If the number `x` is present, we calculate the rotations as the total number of dominos minus the maximum appearance count (`max(cnt1, cnt2)`). This represents the side with fewer appearances of `x`, which would need to be flipped to make all the numbers match `x`. We take the minimum value from applying function `f(x)` to both `tops[0]` and `bottoms[0]` to find the overall minimum rotations needed.

This approach is greedy because it seeks a local optimization—focusing on aligning all numbers to match either `tops[0]` or `bottoms[0]`, to give a global solution which is the minimum number of rotations required. If neither number can be aligned, the function recognizes this by returning infinity, and thus the overall solution returns -1.

Solution Approach

The solution implements a simple yet clever approach utilizing a helper function that encapsulates the core logic required to solve the problem.

Let's walk through the steps:

1. We have a helper function `f(x)` which takes `x` as a parameter. This function is responsible for determining the minimum number of rotations needed to make all values in `tops` or all values in `bottoms` equal to `x`.
2. Inside `f(x)`, we initiate two counters, `cnt1` and `cnt2`, which count the occurrences of `x` in `tops` and `bottoms` arrays respectively. These counters are important as they help to evaluate the current state with respect to our target value `x`.
3. A loop runs over both `tops` and `bottoms` simultaneously using `zip(tops, bottoms)` which pairs the top and bottom values of each domino. For each pair `(a, b)`:
   - If `a` is not found in either `a` or `b`, it implies that no rotations can make the `i`th domino's top or bottom value `x`. Hence, we return `inf` which denotes an impossible situation for `x`.
   - If `x` is found, we increment `cnt1` if `a` (top value) equals `x`, and similarly, we increment `cnt2` if `b` (bottom value) equals `x`.
4. After iterating over all dominos, the minimum rotations needed is calculated by taking the length of the tops array (which is the total number of dominos) and subtracting the larger of `cnt1` or `cnt2`. This is because to make all numbers equal to `x`, we simply need to rotate the dominos which currently do not have `x` on the desired side (`tops` or `bottoms`). The side with a larger count of `x` will need fewer rotations since more dominos are already showing `x`.
5. Our main function `minDominoRotations` now calls `f(x)` for `tops[0]` and `bottoms[0]` and calculates the minimum between these two results. If we get `inf`, it means rotating won't help us achieve a uniform side, hence we return `-1`.
6. Otherwise, we return the minimum of the two values which represents the minimum number of rotations required for making all `tops` or all `bottoms` values equal.

This solution uses the **greedy algorithmic paradigm** where we take the local optimum (minimum rotations for `tops[0]` or `bottoms[0]`) to reach a global solution. Essential to this approach are the **conditional checks** and **value counts** which allow us to decide quickly the feasibility and cost (number of rotations) for each potential solution.

Additionally, by using Python's `zip` function to iterate over `tops` and `bottoms` simultaneously and its expressive syntax, the implementation remains clean, intuitive, and efficient.

Example Walkthrough

Let's consider small arrays for `tops` and `bottoms` for an illustrative example:

Suppose `tops = [2, 1, 2, 4, 2]` and `bottoms = [5, 2, 2, 2, 2]`. We want to make all numbers in the top or bottom uniform with the minimum number of rotations.

- Firstly, we choose `x` to be `tops[0]`, which is 2.
- Applying function `f(x)`:
  - We initialize `cnt1` and `cnt2` both to zero, which will count the occurrences of 2 in `tops` and `bottoms` respectively.
  - As we iterate:
    - For the first domino, `tops[0]` is 2 so we increment `cnt1` to 1.
    - For the second domino, `bottoms[1]` is 2, so we increment `cnt2` to 1.
    - The third domino has 2 on both `tops` and `bottoms`, so we increment both `cnt1` and `cnt2`.
    - The fourth domino does not have 2 on the top, but it is on the bottom, so we increment `cnt2`.
    - The fifth domino has 2 on the top, so we increment `cnt1`.
  - After completing the iteration, `cnt1` is 4 (since `tops` has the number 2 in four positions) and `cnt2` is 4 as well (as `bottoms` also has the number 2 in four positions).
  - The minimum number of rotations f(2) will be the total number of dominos (5) minus the maximum occurrences of 2 (4), which gives us 1. We can achieve this by flipping the fourth domino.
- Secondly, we choose `x` to be `bottoms[0]`, which is 5.
- Applying function `f(x)` with 5:
  - We find that 5 does not appear in `tops` at all and only appears once in `bottoms`. This means we would have to flip every domino except the first one to get 5 on the top everywhere. This gives us a count of 4 flips.

When we compare the results, flipping to get all 2's requires only 1 rotation, while flipping to get all 5's requires 4 rotations. Therefore, the minimum number of rotations required is 1, and the target number is 2.

Hence, `minDominoRotations` would return 1, which indicates that we should flip the fourth domino for all dominos to show the number 2 on either `tops` or `bottoms`. Since both `tops` and `bottoms` have the number 2 on four out of five positions already, the target number 2 is the optimal choice for achieving uniformity with the minimum rotations possible.

Python Solution

```python
from typing import List

class Solution:
    def minDominoRotations(self, tops: List[int], bottoms: List[int]) -> int:
        # Helper function that tries to make all the dominos show x on top.
        # It returns the minimum rotations needed, or infinity if it's not possible.
        def check_value(x: int) -> int:
            rotations_top = rotations_bottom = 0

            # Compare each pair of top and bottom values.
            for top_value, bottom_value in zip(tops, bottoms):
                # If the value x is not in either the top or bottom, return infinity
                # since it's impossible to make all dominoes show x.
                if x not in (top_value, bottom_value):
                    return float('inf')

                # Count how many times value x appears on the top and bottom.
                rotations_top += top_value == x
                rotations_bottom += bottom_value == x

            # Return the minimum rotations needed, which is the total number of dominoes
            # minus the maximum appearance of the value x on either side.
            return len(tops) - max(rotations_top, rotations_bottom)

        # Try making all dominoes show the first value of tops or bottoms.
        # The min function will choose the smallest result from the two calls,
        # which represents the minimum rotations to achieve the goal.
        min_rotations = min(check_value(tops[0]), check_value(bottoms[0]))

        # If min_rotations is infinity, it means it's not possible to make all
        # dominoes show the same number. Otherwise, return the minimum rotations.
        return -1 if min_rotations == float('inf') else min_rotations
```

Java Solution

```java
class Solution {
    private int arrayLength;
    private int[] tops;
    private int[] bottoms;

    /**
     * Finds the minimum number of rotations to have all the values in either top or bottom equal.
     *
     * @param tops    - array representing the top values of each domino.
     * @param bottoms - array representing the bottom values of each domino.
     * @return minimum number of rotations, or -1 if it is not possible.
     */
    public int minDominoRotations(int[] tops, int[] bottoms) {
        // Set the global variables for easy access in the helper function.
        arrayLength = tops.length;
        this.tops = tops;
        this.bottoms = bottoms;

        // Check for the possibility of all dominos having the first top value or the first bottom value.
        int rotations = Math.min(findRotationsForValue(tops[0]), findRotationsForValue(bottoms[0]));

        // If the number of rotations is greater than the array length, it means it's impossible to achieve the goal.
        return rotations > arrayLength ? -1 : rotations;
    }

    /**
     * Helper function to calculate the rotations needed to make all the values of tops or bottoms equal to x.
     *
     * @param value - the value to match across all tops or bottoms.
     * @return number of rotations needed or a value greater than n if not possible.
     */
    private int findRotationsForValue(int value) {
        int countTops = 0, countBottoms = 0;

        // Traverse through all dominos.
        for (int i = 0; i < arrayLength; ++i) {
            // If the current domino does not have the desired value on either side, the configuration is not possible.
            if (tops[i] != value && bottoms[i] != value) {
                return arrayLength + 1;
            }

            // Increment count of occurrence of value in tops and bottoms.
            if (tops[i] == value) {
                ++countTops;
            }
            if (bottoms[i] == value) {
                ++countBottoms;
            }
        }

        // The minimum rotations is the length of the array minus
        // the maximum count of the value in either tops or bottoms.
        return arrayLength - Math.max(countTops, countBottoms);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minDominoRotations(vector<int>& tops, vector<int>& bottoms) {
        int size = tops.size();

        // Helper function to calculate the minimum rotations needed to make all dominoes show number 'x' on top
        // If it's impossible to make all the numbers 'x', the function returns more than 'size' which is an invalid number of rots
        auto checkRotations = [&](int x) {
            int topCount = 0, bottomCount = 0;
            for (int i = 0; i < size; ++i) {
                // If neither the top nor the bottom of the i-th domino is 'x', it's not possible to make all numbers 'x'
                if (tops[i] != x && bottoms[i] != x) {
                    return size + 1;
                }
                // Count how many tops are already 'x'
                topCount += tops[i] == x;
                // Count how many bottoms are already 'x'
                bottomCount += bottoms[i] == x;
            }
            // You want to preserve the side which already has the most 'x's and rotate the other side
            // Hence you need 'size' - max(topCount, bottomCount) rotations
            return size - max(topCount, bottomCount);
        };

        // Compute the minimum rotations for the first number on the top and bottom to make the entire row uniform
        int minRotations = min(checkRotations(tops[0]), countMinRotations(bottoms[0]));

        // If the computed rotations is greater than 'size', it's not possible to make the row uniform, returns -1
        return minRotations > size ? -1 : minRotations;
    }
};
```

Typescript Solution

```typescript
function minDominoRotations(tops: number[], bottoms: number[]): number {
    const length = tops.length;

    // Function that tries to make all dominos to show number x on the top
    const calculateRotationsForNumber = (targetNumber): number => {
        let topCount = 0;    // Count of targetNumber in the 'tops' array
        let bottomCount = 0;   // Count of targetNumber in the 'bottoms' array

        // Iterate over the domino pieces
        for (let i = 0; i < length; ++i) {
            // If the current top and bottom do not contain the target number, return invalid result
            if (tops[i] !== targetNumber && bottoms[i] !== targetNumber) {
                return length + 1;
            }
            // Count occurrences of targetNumber in tops
            if (tops[i] === targetNumber) {
                topCount++;
            }
            // Count occurrences of targetNumber in bottoms
            if (bottoms[i] === targetNumber) {
                bottomCount++;
            }
        }
        // Return minimum rotations needed by subtracting the max occurrences from the total length
        return length - Math.max(topCount, bottomCount);
    };

    // Calculate the minimum rotations needed for the first elements of tops and bottoms
    const rotations = Math.min(
        calculateRotationsForNumber(tops[0]),
        calculateRotationsForNumber(bottoms[0])
    );

    // If rotations are greater than length, that means we cannot make all values in one side equal; return -1
    return rotations > length ? -1 : rotations;
}
```

Time and Space Complexity

The time complexity of the code provided is $O(n)$, where n is the length of the arrays `tops` and `bottoms`. This is because the function `f(x)` goes through all the elements in `tops` and `bottoms` once, using a single loop with `zip(tops, bottoms)`, to count the occurrences. The function `f(x)` is called at most twice, once for `tops[0]` and once for `bottoms[0]`, regardless of the size of the input. Therefore, the total number of operations is proportional to the size of the arrays, thus the $O(n)$ time complexity.

The space complexity is $O(1)$. This is because the extra space used by the function does not depend on the size of the input arrays. The variables `cnt1`, `cnt2`, and `ans` use a constant amount of space, and no additional data structures that grow with the input size are used.