799. Champagne Tower

Medium Dynamic Programming Leetcode Link

Problem Description

This problem presents a scenario where glasses are stacked in a pyramid fashion to form a champagne tower. Each row in the pyramid has a number of glasses equal to the row number, starting from 1 for the first row, 2 for the second row, and continuing up to the 100th row. Each glass can hold one cup of champagne. When we pour champagne into the top glass, it fills glasses below it in the following manner: once the current glass is full, any extra champagne splits equally between the two glasses directly below it. This process continues down the pyramid until the champagne either fills glasses or spills on the floor if it reaches the last row.

The actual problem is to determine how full a specific glass, given by its row (`query_row`) and position in the row (`query_glass`), is after pouring a certain number of cups (`poured`) of champagne into the top glass of the tower.

Intuition

To solve this problem, we can use a dynamic programming approach where we simulate the pouring process. We will create a 2D array that represents the champagne glasses and track the amount of champagne in each glass. When we pour champagne into the glass at the top, we continue to distribute it down the rows. If a glass receives more than one cup of champagne, it overflows, and the excess amount is shared equally between the two glasses directly below it.

We're only interested in the distribution of champagne up to and including the `query_row`, so we can limit the simulation to that. By initializing the overflow process from the top and proceeding row by row, we can efficiently calculate how much champagne is in each glass when the overflow process ceases - either when we reach the desired row or when no more champagne can overflow.

When representing the overflow, any glass that has more than one cup of champagne will distribute the excess. This is calculated by subtracting 1 cup (the glass's capacity) from its champagne amount, dividing the remainder by 2, and adding the result to the two glasses below. This is iterated for each row until we get to the row we need to query, which contains the glass of interest.

Finally, we query the amount of champagne in the specific glass. If the glass is full, it will have one cup of champagne; if it is not full, it will have an amount equal to whatever was poured into it through the overflow process. Since glasses can't hold more than one cup, the answer will be 1 if the calculated amount is over 1 cup, or the calculated amount itself if it's less than or equal to 1 cup.

Solution Approach

The solution uses dynamic programming to solve the problem efficiently. Here's how the implementation goes:

1. **Create a 2D Array**: We initialize a 2D array `f` with dimensions 101×101, which represents the champagne glasses. We use a size of 101 because that accounts for the 100th row potentially overflowing into an additional row below it.
2. **Base Condition**: We set the champagne in the top glass `f[0][0]` to be equal to the `poured` amount.
3. **Simulate Pouring Process**: We iterate through the rows up to `query_row + 1`. For each row `i`, we iterate through the glasses `j` in that row up to `i + 1`, because the number of glasses in each row is equal to the row number.
4. **Check for Overflow**: Inside the nested loop, we check if a glass has more than one cup of champagne by checking if `f[i][j]` is greater than `1`.
5. **Distribute Excess Champagne**: If the glass overflows, we calculate the excess amount that will flow to the glasses below by subtracting one cup and dividing the remainder by two (this is `half`).
6. **Update Adjacent Glasses**: We update the two glasses below the current glass (positions `[i + 1][j]` and `[i + 1][j + 1]` in the `f` array) by adding the `half` to both. This represents the flow of excess champagne to lower-level glasses.
7. **Cap Glass Fullness**: We set the current glass `f[i][j]` to `1` because a glass can't hold more than one cup of champagne.
8. **Query Result**: After completing the simulation up to the `query_row`, we simply return the value at `f[query_row][query_glass]`. If it's more than `1`, it means the glass is overflowing, and we return `1`. Otherwise, we return the actual value since that's how full the glass is.

The patterns used in the solution are dynamic programming (memoization of intermediate results to avoid recomputation) and simulation (simulating the behavior of champagne pouring over the glasses).

The algorithm iteratively updates the state of the champagne tower as champagne is poured until it reaches the query condition. This allows us to only perform necessary calculations and retrieve the exact fullness of the requested glass in an efficient manner. The dynamic programming pattern reduces the overall time complexity as compared to a naive recursive approach.

Here's the code snippet that performs these steps:

```python
class Solution:
    def champagneTower(self, poured: int, query_row: int, query_glass: int) -> float:
        f = [[0] * 101 for _ in range(101)]
        f[0][0] = poured
        for i in range(query_row + 1):
            for j in range(i + 1):
                if f[i][j] > 1:
                    half = (f[i][j] - 1) / 2
                    f[i][j] = 1
                    f[i + 1][j] += half
                    f[i + 1][j + 1] += half
        return f[query_row][query_glass]
```

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we pour `3` cups of champagne into the top glass of a tower and want to find out how full the glass at row `2`, position `1` (zero-indexed) is.

Here's how the process works step-by-step:

- **Step 1 (Initialization)**: We create the 2D array `f` to represent our glasses, all initially with `0` champagne.
- **Step 2 (Top Glass)**: We pour `3` cups into the top glass `f[0][0]`, so it becomes `3`.
- **Step 3 (First Row)**: As there is an overflow (since `3` > `1`), we calculate the excess champagne, which is `3 - 1 = 2` cups. We then divide this by `2`, so each glass below will receive `1` cup.

  At this point, the array will look like this:

  ```
  Row 0:  3
  Row 1:  1   1
  Row 2:  0   0   0
  ```

- **Step 4 (Second Row)**: Now we look at each glass in the first row. Each glass has `1` cup, so no overflow occurs.

  The array now:

  ```
  Row 0:  1
  Row 1:  1   1
  Row 2:  0   0   0
  ```

- **Step 5 (Third Row)**: Since there's no overflow from the first row, the second row's glasses remain as they are.

  The final state of the array up to row `2`:

  ```
  Row 0:  1
  Row 1:  1   1
  Row 2:  0   0   0
  ```

- **Step 6 (Result Query)**: Now we query the fullness of the glass at row `2`, position `1`, which corresponds to `f[2][1]`. As we can see, it is `0`, meaning no champagne has reached this glass. Therefore, the return value would be `0`.

In conclusion, after pouring `3` cups of champagne, the glass at row `2`, position `1` is empty. The code provided efficiently simulates the entire pouring process, accounts for any overflows, and gives us the correct amount of champagne in any given glass.

Python Solution

```python
class Solution:
    def champagneTower(self, poured: int, query_row: int, query_glass: int) -> float:
        # Initialize a Pascal's triangle with 101 rows and columns as the champagne tower, and only 0 is filled.
        tower = [[0.0] * 101 for _ in range(101)]

        # The champagne poured into the top glass.
        tower[0][0] = poured

        # Simulating the pouring process up to the query_row + 1 because we need values from the given query row.
        for row in range(query_row + 1):
            for col in range(row + 1):
                # If there's an overflow in the current glass...
                if tower[row][col] > 1:
                    # Calculate the champagne that flows to each glass below.
                    overflow = (tower[row][col] - 1) / 2.0
                    # Reset the current glass to full after overflow.
                    tower[row][col] = 1
                    # Distribute the overflowed champagne to the two glasses below.
                    tower[row + 1][col] += overflow
                    tower[row + 1][col + 1] += overflow

        # Return the amount of champagne in the glass at query_row and query_glass, capped at 1.
        return min(1, tower[query_row][query_glass])

```

Java Solution

```java
class Solution {

    // Computes the amount of champagne in a glass located at (queryRow, queryGlass)
    // after pouring a certain amount into the top glass of the tower.
    public double champagneTower(int poured, int queryRow, int queryGlass) {
        // Initialize a 2D array to hold the quantity of champagne in each glass
        double[][] champagneLevel = new double[101][101];

        // Pour champagne into the top glass
        champagneLevel[0][0] = poured;

        // Fill the glasses for each row till the queried row
        for (int row = 0; row <= queryRow; row++) {
            for (int glass = 0; glass <= row; glass++) {
                // Check if the current glass is overflowing
                if (champagneLevel[row][glass] > 1) {
                    // Calculate the amount of champagne that overflows, to be divided between the two glasses below
                    double overflow = (champagneLevel[row][glass] - 1) / 2.0;

                    // Current glass should not have more than 1 unit of champagne after overflowing
                    champagneLevel[row][glass] = 1;

                    // Distribute the overflowing champagne to the two glasses below it
                    champagneLevel[row + 1][glass] += overflow;
                    champagneLevel[row + 1][glass + 1] += overflow;
                }
            }
        }

        // Return the amount of champagne in the queried glass
        // (it will be at most 1 since any excess would have overflowed)
        return champagneLevel[queryRow][queryGlass];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    double champagneTower(int poured, int queryRow, int queryGlass) {
        // Initialize the array that will store the quantity of champagne in each glass.
        // Only need 100 rows according to the problem statement.
        double glasses[100][100] = {0.0};

        // Pour the champagne into the top glass.
        glasses[0][0] = poured;

        // Start from the top and fill down to the queried row.
        for (int row = 0; row <= queryRow; ++row) {
            for (int glass = 0; glass <= row; ++glass) {
                // Check if there is any overflow in the current glass.
                if (glasses[row][glass] > 1) {
                    // Calculate the amount of champagne that flows to the glasses below.
                    double overflow = (glasses[row][glass] - 1.0) / 2.0;
                    // Ensure the current glass is filled to its capacity.
                    glasses[row][glass] = 1;
                    // Distribute the overflow to the two glasses below equally.
                    glasses[row + 1][glass] += overflow;
                    glasses[row + 1][glass + 1] += overflow;
                }
            }
        }
        // Return the amount of champagne in the queried glass.
        // If the glass is full or under, it will contain the exact amount.
        // If it did not receive enough champagne, it will contain the remaining amount.
        return glasses[queryRow][queryGlass];
    }
};

```

Typescript Solution

```typescript
// Defines a function to simulate pouring champagne into a tower and queries the amount of champagne
// in a specific glass at a specific row after the champagne is poured.
function champagneTower(poured: number, queryRow: number, queryGlass: number): number {
    // Initialize the first row with the amount of champagne poured into the first glass
    let currentRow = [poured];

    // Iterate over the rows of the champagne tower until the specified query row is reached
    for (let rowIndex = 1; rowIndex <= queryRow; rowIndex++) {
        // Initialize the next row with zeros which will represent the empty glasses
        const nextRow = new Array(rowIndex + 1).fill(0);

        // Iterate over each glass in the current row
        for (let glassIndex = 0; glassIndex < rowIndex; glassIndex++) {
            // If the current glass has more than one unit of champagne, it overflows
            if (currentRow[glassIndex] > 1) {
                // Calculate the amount of champagne that overflows from the current glass and distribute it
                // equally to the two glasses below it in the next row
                const overflow = (currentRow[glassIndex] - 1) / 2;
                nextRow[glassIndex] += overflow;
                nextRow[glassIndex + 1] += overflow;
            }
        }
        // Set the nextRow to be the currentRow for the next iteration
        currentRow = nextRow;
    }

    // Return the amount of champagne in the specified glass, capped at 1 because glasses can't hold
    // more than one unit of champagne
    return Math.min(1, currentRow[queryGlass]);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the nested for loops. In the worst case, the loop runs for `query_row + 1` iterations, and for each iteration `i`, it runs `i + 1` times, because the inner loop ranges up to the current row number. Thus, the number of operations can be approximated as a sum of an arithmetic series, which calculates to roughly `1 + 2 + ... + (query_row + 1)`, which is `((query_row + 1) * (query_row + 2)) / 2`. Simplifying this arithmetic series results in O(query_row^2). Hence, the time complexity is `O(query_row^2)`.

Space Complexity

The space complexity is determined by the size of the 2D list `f`, which is created to store the quantity of champagne in each glass. Since the list is initialized with dimensions 101×101, this is essentially a constant space allocation, not varying with the size of the input. Therefore, the space complexity is `O(1)` in terms of the input `poured`, `query_row`, and `query_glass`; however, if considering the size of the grid as part of the complexity, it would be `O(101 * 101)` which simplifies to `O(1)` under Big O notation since 101 is a constant.