2137. Pour Water Between Buckets to Make Water Levels Equal

Medium | Array | Binary Search | Leetcode Link

Problem Description

You are given an array `buckets` where the `i`-th bucket contains `buckets[i]` gallons of water, and the objective is to equalize the water in all buckets. However, when you pour `x` gallons of water from one bucket to another, a certain percentage of the water (`loss` percent) is spilled and lost. The challenge is to calculate the maximum amount of water that could be in each bucket after distributing the water as evenly as possible while taking into account the water loss during each pour. The expected output is a floating-point number that represents the maximum water level achievable in each bucket, where the result is considered correct if it's within `10^-5` of the actual answer.

Intuition

To solve this problem, we use a binary search method because we are trying to find a value (the maximum amount of water that can be in each bucket) within a certain range (from 0 to the maximum volume in any of the buckets). The intuition behind using binary search in this context comes from understanding that:

1. If it's possible to make the water level `v` in all buckets without violating the conditions, then any value lower than `v` would also be possible because you would just spill less water.
2. If it's impossible to achieve the water level `v`, then any value higher than `v` would also be impossible as it would require even more water to be transferred and hence, more would be lost due to spilling.

With each midpoint value calculated during the binary search, we simulate the process of equalizing the water to that level and check if it would be possible considering the spilling loss. If it's possible to achieve the water level at the midpoint, it becomes the new lower bound; otherwise, it becomes the new upper bound. We stop the binary search process when the difference between the upper and lower bounds is smaller than `10^-5`, and we return the lower bound as the maximum water level possible in each bucket.

The `check` function within the solution plays a critical role - it calculates the total water that can be removed from buckets with excess water (`a`) and the total water required to fill buckets with less water than the midpoint value (`b`), applying the `loss` percentage to each transfer. If `a` is greater than or equal to `b`, then it's possible to achieve the equalized water level `v`.

Solution Approach

The solution provided uses binary search to find the maximum level of water that can be achieved in each bucket. Here's an explanation of the key components of the implementation:

- **Binary Search:** Binary search is an efficient algorithm for finding an item from a sorted list of items. In our case, though the list is not explicitly sorted, we know that the maximum level lies between a range – from 0 to the maximum number of gallons in any of the buckets.
- **Check Function:** The core of the solution is the `check` function which determines if a certain water level `v` is achievable. It calculates total excess water (`a`) that can be moved from buckets with water levels above `v` and the total required water (`b`) to be poured into buckets below the level `v`, accounting for the loss during transfer.
- **Mathematical Calculations:** When transferring water from a bucket with excess water, there's no spillage, hence we subtract directly. But when we need to add water to a bucket with less water, we calculate how much extra water is needed given that there will be a `loss` percent spillage. This is calculated with the formula `(v - x) * 100 / (100 - loss)` where `x` is the target level, `v` is the current level, and `loss` is the percentage lost.
- **Convergence Criterion:** The solution keeps adjusting the bounds of the binary search until the range between the lower (`l`) and upper (`r`) bounds is less than `1e-5`. At this point, the value of `l` is close enough to the true answer that it can be returned.

Here's a step-by-step breakdown of how the code implements the solution using the binary search algorithm:

1. Initialize the lower (`l`) and upper (`r`) bounds for the binary search. `l` starts at 0, and `r` starts at the maximum value in `buckets`.
2. Perform the binary search: a. Calculate the midpoint (`mid`) between the current bounds `l` and `r`. b. Utilize the `check` function to see if it's possible to equalize water to the level `mid`. The function does this by simulating the water pouring and spillage process. c. If the `check` function returns `True`, it means it's possible to equalize at least to the level `mid`. Hence, update the lower bound to `mid`. d. If the `check` function returns `False`, update the upper bound to `mid`.
3. Repeat the binary search process until the difference between `l` and `r` is less than `1e-5`. The lower bound `l` will then give us the maximum achievable level with the given loss percentage.
4. Return the lower bound `l` as the final answer.

By repeatedly narrowing the search range, the binary search algorithm efficiently zeroes in on the maximum level of water that can be achieved in each bucket.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the array `buckets = [10, 15, 20]` and a loss percentage `loss = 5`. We are to calculate the maximum amount of water that could be in each bucket after transferring water while accounting for the spillage.

Step 1: Initialize `l` (lower bound) to 0 and `r` (upper bound) to the maximum value in `buckets`, which is 20.

Step 2: Begin binary search:

- Calculate midpoint `mid` between `l` and `r`. Initially, `mid = (0 + 20) / 2 = 10`.
- Use the `check` function to determine if we can equalize all buckets to level `10` after considering the 5% loss in transfers.

Step 3: Apply the `check` function:

- Start with the total excess water `a` and required water `b` set to 0.
- Bucket 1 (`buckets[0]`): Has 10 gallons, which is equal to `mid`, no action needed.
- Bucket 2 (`buckets[1]`): Has 15 gallons, which is more than `mid`. We can move `(15 - 10) = 5` gallons out. No spillage occurs when removing water, so `a += 5`.
- Bucket 3 (`buckets[2]`): Has 20 gallons, also more than `mid`. We can move `(20 - 10) = 10` gallons out. Again, no spillage when removing water, so `a += 10`.
- Total excess `a` is equal to 15 gallons of water.

Next, calculate needed water `b` to reach the level `mid` for the buckets below `mid`, which in this case, there are none. Since no bucket has less than 10 gallons, `b = 0`.

Check whether `a` can cover `b` after accounting for the 5% loss. Here, the `a` is more than enough to cover `b`, as `a` is 0.

Then, update our bounds:

- Since we could equalize to level `10`, let's move the lower bound up to `mid`. Set `l` to `mid` which is now 10.

Step 4: Repeat the binary search process with the new bounds:

- Now `l = 10` and `r = 20`, calculate a new `mid = (10 + 20) / 2 = 15`.
- Running the `check` function with `mid = 15`, we find:
  - From Bucket 1 (`buckets[0]`), we now need `(15 - 10) * 100 / (100 - 5) ≈ 5.26` gallons considering 5% loss.
  - From Bucket 2 (`buckets[1]`), we have an excess of 0 gallons, since it's now equal to `mid`.
  - From Bucket 3 (`buckets[2]`), we can take away `(20 - 15) = 5` gallons with no spillage.
  - Total excess `a` is 5 gallons, but we need approximately `5.26` gallons for Bucket 1, so `a < b`.

Since `a` isn't sufficient to cover `b`, we adjust the upper bound:

- Set `r` to `mid`.

Step 5: Keep adjusting `l` and `r`, recalculating `mid`, and using the `check` function until the difference between `l` and `r` is less than `1e-5`.

By iterating over this process, the binary search algorithm eventually narrows down to the maximum water level that can be achieved with the spillage taken into account. For this example, let's say the binary search converges to a water level of `approx. 12.62` gallons.

Finally, we return the value that `l` has converged to, which in our example would be roughly `12.82 gallons`, as the maximum achievable water level in each bucket.

Python Solution

```python
from typing import List  # Import List from typing module for type hinting

class Solution:
    def equalizeWater(self, buckets: List[int], loss: int) -> float:
        # Define an internal helper function that checks
        # if it is possible to equalize the water to volume v
        def check(v):
            water_to_remove = 0  # Water to remove from larger buckets
            water_to_add = 0  # Water to add to smaller buckets

            for water_in_bucket in buckets:
                if water_in_bucket >= v:  # If current bucket has more water than v
                    # Calculate the water to remove without loss
                    water_to_remove += water_in_bucket - v
                else:  # If current bucket has less water than v
                    # Calculate the water to add considering the loss
                    water_to_add += (v - water_in_bucket) * 100 / (100 - loss)

            return water_to_remove >= water_to_add  # We can equalize if we have enough water to redistribute

        # Initialize the binary search boundaries
        left = 0
        right = max(buckets)  # The maximum possible volume is the largest bucket

        # Continue the loop until the precision of 1e-5 is reached
        while right - left > 1e-5:
            mid = (left + right) / 2  # Consider the mid-volume between left and right
            # Check if it's possible to equalize water to this mid volume
            if check(mid):  # If possible,
                left = mid  # try to see if there's room for more water
            else:
                right = mid  # Otherwise, we reduce the target volume

        # Final volume after equalization, rounded to 4 decimal places
        return round(left, 4)
```

Java Solution

```java
class Solution {
    // Function to equalize the water in the buckets considering the loss in pouring.
    public double equalizeWater(int[] buckets, int loss) {
        // Initialize left to the lowest possible amount of water, which is 0.
        double left = 0;
        // The maximum amount of water can only be as much as the fullest bucket.
        double right = Arrays.stream(buckets).max().getAsInt();

        // Perform binary search within the precision of 1e-5.
        while (right - left > 1e-5) {
            double mid = (left + right) / 2;
            // If it's possible to equalize to mid liters with the allowed loss, search higher.
            if (canEqualize(buckets, loss, mid)) {
                left = mid;
            }
            // Otherwise, search lower.
            else {
                right = mid;
            }
        }

        // Return the maximum possible equal amount of water.
        return left;
    }

    // Helper function to check if it is possible to equalize to `targetVolume` liters.
    private boolean canEqualize(int[] buckets, int loss, double targetVolume) {
        double excessWater = 0; // Water to be poured out from fuller buckets.
        double requiredWater = 0; // Water needed to fill the less full buckets.

        // Iterate through each bucket.
        for (int waterInBucket : buckets) {
            // If the bucket has more water than the target, calculate the excess.
            if (waterInBucket > targetVolume) {
                excessWater += waterInBucket - targetVolume;
            }
            // If the bucket has less water than the target, calculate the required amount.
            else {
                requiredWater += (targetVolume - waterInBucket) * 100 / (100 - loss);
            }
        }

        // We can equalize the water if we have enough excess to cover the required amount after considering the loss.
        return excessWater >= requiredWater;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum equal water level that can be achieved
    // in all buckets after accounting for water loss due to transferring.
    double equalizeWater(vector<int>& buckets, int loss) {
        // Initialize the search space for the binary search
        double left = 0;
        double right = *max_element(buckets.begin(), buckets.end());

        // Function to check if a given water level is achievable
        auto isAchievable = [&](double targetVolume) {
            double excessWater = 0; // Water to be removed from fuller buckets
            double requiredWater = 0; // Water needed for emptier buckets, adjusted for loss
            for (int waterInBucket : buckets) {
                if (waterInBucket > targetVolume) {
                    // Calculate excess water in buckets higher than target level
                    excessWater += waterInBucket - targetVolume;
                } else {
                    // Calculate water needed for buckets lower than target level
                    // Adjust for loss during transfer
                    requiredWater += (targetVolume - waterInBucket) * 100 / (100 - loss);
                }
            }
            // It's achievable if there is enough excess water to account for losses
            // when refilling the emptier buckets.
            return excessWater >= requiredWater;
        };

        // Perform binary search to find the precise water level that is achievable
        while (right - left > 1e-5) { // Loop until a precision of 1e-5
            double mid = (left + right) / 2;
            if (isAchievable(mid)) {
                // If it's possible to achieve 'mid' level, try for a higher level
                left = mid;
            } else {
                // If 'mid' level is not achievable, try for a lower level
                right = mid;
            }
        }

        // After binary search, left is the highest water level that can be achieved
        return left;
    }
};
```

Typescript Solution

```typescript
function equalizeWater(buckets: number[], loss: number): number {
    // Define lower and upper bounds for binary search
    let lowerBound = 0;
    let upperBound = Math.max(...buckets);

    // Helper function to check if it is possible to equalize at volume `targetVolume`
    // with the given loss percentage
    const canEqualizeToVolume = (targetVolume: number): boolean => {
        let excessWater = 0; // Water to be removed
        let totalRequiredWater = 0;
        for (const bucketVolume of buckets) {
            if (bucketVolume >= targetVolume) {
                // Add excess water from buckets above the targetVolume,
                // we add the excess water to totalExcessWater.
                excessWater += bucketVolume - targetVolume;
            } else {
                // If the current bucket has less water than targetVolume,
                // we calculate how much water considering loss is required to reach
                // targetVolume and add it to totalRequiredWater.
                totalRequiredWater += (targetVolume - bucketVolume) * 100 / (100 - loss);
            }
        }
        // Return true if there is enough excess water to make up for the required water after loss
        return totalExcessWater >= totalRequiredWater;
    };

    // Perform binary search to find the maximum water level possible
    while (upperBound - lowerBound > 1e-5) {
        const midAttemptVolume = (lowerBound + upperBound) / 2;

        // Use the helper function to decide which half of the search range to keep
        if (canEqualizeToVolume(midAttemptVolume)) {
            lowerBound = midAttemptVolume;
        } else {
            upperBound = midAttemptVolume;
        }
    }

    // Return the lower bound as the result, which is the maximum water level possible after trimming
    // off the extra places up to the fifth (which is the precision of our binary search)
    return lowerBound;
}
```

Time and Space Complexity

The given Python code performs a binary search on the range of possible water levels to find the correct water level such that the total amount of water after adjusting all buckets equals that level even with water loss during the transfer. Let's break down the time and space complexity:

Time Complexity

The time complexity of this function is determined by the binary search and the loop inside the `check` function.

- **Binary Search:** The binary search runs while the difference between `l` (low) and `r` (high) is greater than `1e-5`. In each iteration, this range is halved. Therefore, the number of iterations needed for the binary search to converge is $O(\log(\text{Range}/1e-5))$, where `Range` represents the initial range (`max(buckets) - min(buckets)`).
- **Check Function:** Inside each binary search iteration, a `check` function is called that iterates through all buckets to compare their volume against the mid value. This function has a loop that runs once for each bucket. Therefore, it has a $O(n)$ complexity, where n is the number of buckets.

Multiplying these together, the overall time complexity is $O(n * \log(\text{Range}/1e-5))$. Since `1e-5` is a constant, we can simplify this to $O(n * \log(\text{Range}))$.

Space Complexity

The space complexity is simpler to analyze.

- There are no additional data structures that grow with the input size. The only extra space used is for a constant number of variables (`a`, `b`, `l`, `r`, `mid`, and the space to hold the `check` function).
- As such, the space complexity is $O(1)$ - constant space complexity.

In summary:

- Time Complexity: $O(n * \log(\text{Range}))$
- Space Complexity: $O(1)$