<https://leetcode.ca/2022-01-29-2137-Pour-Water-Between-Buckets-to-Make-Water-Levels-Equal/>

You have n [buckets](https://leetcode.ca/2022-01-29-2137-Pour-Water-Between-Buckets-to-Make-Water-Levels-Equal/#) each containing some gallons of water in it, represented by a **0-indexed** integer array buckets, where the ith [bucket](https://leetcode.ca/2022-01-29-2137-Pour-Water-Between-Buckets-to-Make-Water-Levels-Equal/#) contains buckets[i] gallons of water. You are also given an integer loss.

You want to make the amount of water in each bucket equal. You can pour any amount of water from one bucket to another bucket (not necessarily an integer). However, every time you pour k gallons of water, you spill loss **percent** of k.

Return *the****maximum****amount of water in each bucket after making the amount of water equal.*Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

**Input:** buckets = [1,2,7], loss = 80

**Output:** 2.00000

**Explanation:** Pour 5 gallons of water from buckets[2] to buckets[0].

5 \* 80% = 4 gallons are spilled and buckets[0] only receives 5 - 4 = 1 gallon of water.

All buckets have 2 gallons of water in them so return 2.

**Example 2:**

**Input:** buckets = [2,4,6], loss = 50

**Output:** 3.50000

**Explanation:** Pour 0.5 gallons of water from buckets[1] to buckets[0].

0.5 \* 50% = 0.25 gallons are spilled and buckets[0] only receives 0.5 - 0.25 = 0.25 gallons of water.

Now, buckets = [2.25, 3.5, 6].

Pour 2.5 gallons of water from buckets[2] to buckets[0].

2.5 \* 50% = 1.25 gallons are spilled and buckets[0] only receives 2.5 - 1.25 = 1.25 gallons of water.

All buckets have 3.5 gallons of water in them so return 3.5.

**Example 3:**

**Input:** buckets = [3,3,3,3], loss = 40

**Output:** 3.00000

**Explanation:** All buckets already have the same amount of water in them.

**Constraints:**

1 <= buckets.length <= 10^5

0 <= buckets[i] <= 10^5

0 <= loss <= 99

**Attempt 1: 2025-04-22**

**Solution 1: Binary Search + Math (60 min)**

**Refer to**

<https://github.com/doocs/leetcode/blob/main/solution/2100-2199/2137.Pour%20Water%20Between%20Buckets%20to%20Make%20Water%20Levels%20Equal/README_EN.md>

**Solution 1: Binary Search for Floating-Point Numbers**

We notice that if a water volume x meets the condition, then all water volumes less than x also meet the condition. Therefore, we can use binary search to find the maximum water volume that satisfies the condition.

We define the left boundary of the binary search as l=0 and the right boundary as r=max(buckets). During each binary search iteration, we take the midpoint mid of l and r, and check if mid meets the condition. If it does, we update l to mid; otherwise, we update r to mid. After the binary search concludes, the maximum water volume that satisfies the condition is l.

The key to the problem is to determine if a water volume v meets the condition. We can iterate through all buckets, and for each bucket, if its water volume is greater than v, then we need to pour out x−v water volume; if its water volume is less than v, then we need to pour in (v−x)×100/(100−loss) water volume. If the total volume poured out is greater than or equal to the volume poured in, then v meets the condition.

The time complexity is O(n×log⁡M), where n and M are the length and the maximum value of the array buckets, respectively. The time complexity of binary search is O(log⁡M), and each binary search iteration requires traversing the array buckets, with a time complexity of O(n). The space complexity is O(1).

class Solution {

public double equalizeWater(int[] buckets, int loss) {

double l = 0, r = Arrays.stream(buckets).max().getAsInt();

while (r - l > 1e-5) {

double mid = (l + r) / 2;

if (check(buckets, loss, mid)) {

l = mid;

} else {

r = mid;

}

}

return l;

}

private boolean check(int[] buckets, int loss, double v) {

double a = 0;

double b = 0;

for (int x : buckets) {

if (x > v) {

a += x - v;

} else {

b += (v - x) \* 100 / (100 - loss);

}

}

return a >= b;

}

}

**Refer to Deepseek**

**Problem Understanding**

The problem requires us to equalize water levels across buckets by pouring water between them, accounting for spillage during transfers. Each pour operation loses a fixed percentage of the transferred amount .

**Key Insights**

Binary Search Applicability: The solution space is monotonic - if a certain water level is achievable, all lower levels are also achievable .

Spillage Calculation: When adding water to a bucket, we need to account for loss by dividing the required amount by (100 - loss)/100 .

Median Not Applicable: Unlike similar problems, the optimal level isn't necessarily the median due to the spillage factor .

**Approach**

We'll use binary search to find the maximum achievable water level:

Search Space: From 0 to the maximum bucket value .

Feasibility Check: For each candidate level, calculate if available excess water can cover the deficit after accounting for spillage .

Precision Handling: Continue until the search space is smaller than 1e-5 .

**Solution Code**

class Solution {

public double equalizeWater(int[] buckets, int loss) {

double left = 0;

double right = Arrays.stream(buckets).max().getAsInt();

double precision = 1e-5;

while (right - left > precision) {

double mid = (left + right) / 2;

if (canEqualize(buckets, loss, mid)) {

left = mid;

} else {

right = mid;

}

}

return left;

}

private boolean canEqualize(int[] buckets, int loss, double target) {

double excess = 0;

double deficit = 0;

double efficiency = (100.0 - loss) / 100.0;

for (int water : buckets) {

if (water > target) {

excess += water - target;

} else {

deficit += (target - water) / efficiency;

}

}

return excess >= deficit;

}

}

**Explanation**

Binary Search Setup: We initialize the search range from 0 to the maximum bucket value .

Feasibility Check:

For buckets above target: accumulate excess water (no loss) .

For buckets below target: calculate required water including spillage .

Termination: The loop continues until the search space is sufficiently small (1e-5) .

Return Value: The left bound gives the maximum achievable water level .

**Complexity Analysis**

Time Complexity: O(n log M), where n is buckets length and M is the maximum value range .

Space Complexity: O(1), using constant extra space .

This approach efficiently finds the optimal water level while accounting for transfer losses through careful binary search and spillage calculations .

**How we come to the Binary Search idea for this problem ?**

The binary search approach for LeetCode 2137 emerges from several key observations about the problem's nature. Here's the detailed reasoning:

**1. Monotonic Property**

The fundamental insight is that the feasibility function is monotonic:

If a certain water level L is achievable, then all levels below L are also achievable (by pouring less water)

Conversely, if L is unachievable, all levels above L are also unachievable

This monotonicity makes binary search applicable, as it allows us to:

Test a candidate level (midpoint)

Narrow the search space based on the result (move left/right)

Converge to the optimal solution

**2. Problem Constraints Analysis**

Fixed Loss Percentage: Each transfer loses loss% of water, making simple averaging impossible

Asymmetrical Operations:

Taking water from fuller buckets has no loss

Adding water to emptier buckets incurs loss

Conservation of Water: Total available water = initial total - all losses during transfers

**3. Why Not Median/Greedy?**

Traditional methods fail because:

Median approach doesn't account for transfer losses

Greedy redistribution would require knowing the exact sequence of pours

Spillage makes the problem non-linear - each transfer's effectiveness depends on the loss percentage

**4. Formulating the Feasibility Check**

For any candidate level target, we calculate:

excess = sum(water - target) for all buckets > target

deficit = sum((target - water)/efficiency) for all buckets < target

Where efficiency = (100 - loss)/100

The level is achievable if: excess >= deficit

**5. Binary Search Parameters**

Initial bounds:

left = 0 (minimum possible level)

right = max(buckets) (maximum possible level)

Termination condition: When right - left < 1e-5 (sufficient precision)

**Visual Example**

Consider buckets = [1,2,7], loss = 50%:

Level 3 check:

- Excess: (7-3) = 4

- Deficit: (3-1)/0.5 + (3-2)/0.5 = 4 + 2 = 6

4 >= 6? → False → try lower level

Level 2 check:

- Excess: (7-2) = 5

- Deficit: (2-1)/0.5 = 2

5 >= 2? → True → try higher level

**Why This Works**

Upper Bound Guarantee: The maximum possible level cannot exceed the tallest bucket

Loss Compensation: The efficiency factor automatically accounts for spillage in deficit calculation

Progressive Refinement: Each iteration halves the search space while preserving the solution

This approach efficiently handles the non-linear aspects introduced by spillage while maintaining optimal O(n log M) complexity.

**The Binary Search solution able to guarantee "Return the maximum amount of water in each bucket after making the amount of water equal" is that because consider each pour will spill water so the "maximum amount of water remains" equal to "minimum pour times", and Binary Search solution guarantee "minimum pour times" ?**

**The binary search solution guarantees the maximum achievable equal water level not because it minimizes pour operations, but because it finds the highest possible level where the net available water after spillage can satisfy all deficits.** Here's why:

**Key Distinctions:**

Objective Difference:

**The problem asks for the maximum equal level, not minimum pours**

**Binary search finds the highest level where the physical constraints (water conservation with spillage) are satisfied**

Spillage Mechanics:

Each pour loses loss% of transferred water

To raise a bucket by Δ, you need to pour Δ/(1 - loss/100) water

This makes the total required water non-linear with respect to the target level

**Why Binary Search Finds the Maximum:**

Feasibility Function:

excess\_water >= total\_deficit\_with\_spillage

At the correct level, the excess water exactly covers the deficit (including spillage losses)

Higher levels would require more water than available after spillage

Search Behavior:

When a level L is feasible, we try higher levels (left = mid)

When infeasible, we try lower levels (right = mid)

This pushes the solution to the highest feasible level

**Mathematical Guarantee:**

The binary search solution works because:

Water Conservation Constraint:

Σ (buckets[i] - target) ≥ 0

(after accounting for spillage in deficit buckets)

Monotonicity:

If level L is feasible, any L' < L is also feasible

The highest feasible L is therefore the maximum possible equal level

**Solution Characteristics:**

|  |  |
| --- | --- |
| Property | Binary Search Solution |
| Objective | Maximizes equal level |
| Pour Operations | Not explicitly minimized |
| Spillage | Fully accounted for |
| Optimality | Guarantees maximum possible level under constraints |

This approach efficiently finds the physical limit of equalization given the spillage constraints, which is different from minimizing transfer operations.

**Refer to**

[L462.Minimum Moves to Equal Array Elements II (Ref.L453)](note://WEBf26377967abd88cf360cde7f55549d27)