2137. Pour Water Between Buckets to Make Water Levels Equal Medium Array **Binary Search Leetcode Link** 

You are given an array buckets where the i-th bucket contains buckets[i] gallons of water, and the objective is to equalize the

# **Problem Description**

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

water in all buckets. However, when you pour k gallons of water from one bucket to another, a certain percentage of the water (loss

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

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

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

more water to be transferred and hence, more would be lost due to spilling.

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.

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

## calculates total excess water (a) that can be moved from buckets with water levels above v and the total required water (b) to be

the buckets.

**Solution Approach** 

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

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

• Check Function: The core of the solution is the check function which determines if a certain water level v is achievable. It

level, x 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 (1) and upper (r) bounds is less than 1e-5. At this point, the value of 1 is close enough to the true answer that it can be returned.

there will be a loss percent spillage. This is calculated with the formula (v - x) \* 100 / (100 - loss) where v is the target

2. Perform the binary search: a. Calculate the midpoint (mid) between the current bounds 1 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 1 and r is less than 1e-5. The lower bound 1 will then give us the

1. Initialize the lower (1) and upper (r) bounds for the binary search. 1 starts at 0, and r starts at the maximum value in buckets.

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

By repeatedly narrowing the search range, the binary search algorithm efficiently zeroes in on the maximum level of water that can

Calculate midpoint mid between 1 and r. Initially, mid = (0 + 20) / 2 = 10.

maximum achievable level with the given loss percentage.

4. Return the lower bound 1 as the final answer.

be achieved in each bucket.

Step 2: Begin binary search:

Step 3: Apply the check function:

removing water, so a += 5.

removing water, so a += 10.

Total excess a is equal to 15 gallons of water.

 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

• Bucket 3 (buckets [2]): Has 20 gallons, also more than mid. We can move (20 - 10) = 10 gallons out. Again, no spillage when

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

• Use the check function to determine if we can equalize all buckets to level 10 after considering the 5% loss in transfers.

Then, update our bounds:

From Bucket 2 (buckets[1]), we have an excess of 0 gallons, since it's now equal to mid.

○ Total excess a is 5 gallons, but we need approximately 5.26 gallons for Bucket 1, so a < b.

∘ From Bucket 3 (buckets [2]), we can take away (20 - 15) = 5 gallons with no spillage.

• Now l = 10 and r = 20, calculate a new mid = (10 + 20) / 2 = 15.

def equalizeWater(self, buckets: List[int], loss: int) -> float:

water\_to\_add = 0 # Water to add to smaller buckets

water\_to\_remove += water\_in\_bucket - v

# Continue the loop until the precision of 1e-5 is reached

else: # If current bucket has less water than v

water\_to\_remove = 0 # Water to remove from larger buckets

# Calculate the water to remove without loss

# Calculate the water to add considering the loss

right = max(buckets) # The maximum possible volume is the largest bucket

# Check if it's possible to equalize water to this mid volume

right = mid # Otherwise, we reduce the target volume

// Function to equalize the water in the buckets considering the loss in pouring.

// The maximum amount of water can only be as much as the fullest bucket.

// Start with the lowest possible amount of water, which is 0.

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

// Return the maximum possible equal amount of water.

if water\_in\_bucket >= v: # If current bucket has more water than v

water\_to\_add +=  $(v - water_in_bucket) * 100 / (100 - loss)$ 

mid = (left + right) / 2 # Consider the mid volume between left and right

left = mid # If possible, we try to see if there's room for more water

# if it is possible to equalize the water to volume v

# Define an internal helper function that checks

def can\_equalize\_to\_volume(v):

for water\_in\_bucket in buckets:

# Initialize the binary search boundaries

if can\_equalize\_to\_volume(mid):

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

while right - left > 1e-5:

Running the check function with mid = 15, we find:

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

• Since we could equalize to level 10, let's move the lower bound up to mid. Set 1 to mid which is now 10.

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

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

• Set r to mid.

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

29

30

31

32

33

14

15

16

18

19

20

21

23

24

25

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

43

45

8

9

10

11

13

14

15

16

17

19

20

21

22

23

24

25

26

27

34

35

36

37

38

39

40

41

42 }

44 };

**}**;

left = 0

else:

double left = 0;

else {

return left;

left = mid;

right = mid;

// Otherwise, search lower.

has less than 10 gallons, b = 0.

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

with the spillage taken into account. For this example, let's say the binary search converges to a water level of approx. 12.82

By iterating over this process, the binary search algorithm eventually narrows down to the maximum water level that can be achieved

∘ From Bucket 1 (buckets [0]), we now need (15 - 10) \* 100 / (100 - 5) ≈ 5.26 gallons considering 5% loss.

from typing import List # Import List from typing module for type hinting class Solution:

34 # Final volume after equalization, rounded to 4 decimal places 35 return round(left, 4) 36

return water\_to\_remove >= water\_to\_add # We can equalize if we have enough water to redistribute

gallons. Finally, we return the value that 1 has converged to, which in our example would be roughly 12.82 gallons, as the maximum achievable water level in each bucket. Python Solution

### // Perform binary search within the precision of 1e-5. while (right - left > 1e-5) { double mid = (left + right) / 2; 11 // If it is possible to equalize to mid liters with the allowed loss, search higher. 12 13 if (canEqualize(buckets, loss, mid)) {

Java Solution

class Solution {

```
// Helper function to check if it is possible to equalize to 'targetVolume' liters.
       private boolean canEqualize(int[] buckets, int loss, double targetVolume) {
27
           double excessWater = 0; // Water to be poured out from fuller buckets.
28
           double requiredWater = 0; // Water needed to fill the less full buckets.
30
31
           // Iterate through each bucket.
32
           for (int waterInBucket : buckets) {
33
               // If the bucket has more water than the target, calculate the excess.
               if (waterInBucket > targetVolume) {
34
                   excessWater += waterInBucket - targetVolume;
35
36
37
               // If the bucket has less water than the target, calculate the required amount.
38
               else {
39
                   requiredWater += (targetVolume - waterInBucket) * 100 / (100 - loss);
40
41
42
43
           // We can equalize the water if we have enough excess to cover the required amount after considering the loss.
           return excessWater >= requiredWater;
44
45
46 }
47
C++ Solution
1 class Solution {
2 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());
           // Functor to check if a given water level is achievable
           auto isAchievable = [&](double targetLevel) {
11
12
               double excessWater = 0; // Water to be removed from fuller buckets
               double requiredWater = 0; // Water needed for emptier buckets, adjusted for loss
13
                for (int waterInBucket : buckets) {
14
                   if (waterInBucket > targetLevel) {
15
                       // Calculate excess water in buckets higher than target level
16
17
                       excessWater += waterInBucket - targetLevel;
                   } else {
18
                       // Calculate water needed for buckets lower than target level
19
                       // Adjust for loss percentage
20
                       requiredWater += (targetLevel - waterInBucket) * 100 / (100 - loss);
21
```

// It's achievable if there is enough excess water to account for losses

// If it's possible to achieve 'mid' level, try for a higher level

// Perform binary search to find the precise water level that is achievable

// If 'mid' level is not achievable, try for a lower level

// After binary search, left is the highest water level that can be achieved

// Helper function to check if it is possible to equalize at volume 'targetVolume'

// If the current bucket has more water than targetVolume,

// If the current bucket has less water than targetVolume,

// we calculate how much water (considering loss) is required to reach

totalRequiredWater += (targetVolume - bucketVolume) \* 100 / (100 - loss);

// Return true if there is enough excess water to make up for the required water after loss

// Return the lower bound as the result, which is the maximum water level possible after trimming

// the decimal places up to the fifth (which is the precision of our binary search)

while (right - left > 1e-5) { // We use a precision of 1e-5

// when filling the emptier buckets

double mid = (left + right) / 2;

1 function equalizeWater(buckets: number[], loss: number): number {

const canEqualizeToVolume = (targetVolume: number): boolean => {

// we add the excess water to totalExcessWater

totalExcessWater += bucketVolume - targetVolume;

// targetVolume and add it to totalRequiredWater

// Define lower and upper bounds for binary search

let upperBound = Math.max(...buckets);

// with the given loss percentage

let totalExcessWater = 0;

} else {

let totalRequiredWater = 0;

for (const bucketVolume of buckets) {

if (bucketVolume >= targetVolume) {

return totalExcessWater >= totalRequiredWater;

upperBound = middleVolume;

// Perform binary search to find the maximum water level possible

Range represents the initial range (max(buckets) - min(buckets)).

variables (a, b, l, r, mid, and the space to hold the check function).

As such, the space complexity is 0(1) - constant space complexity.

if (isAchievable(mid))

left = mid;

right = mid;

} else {

return left;

Typescript Solution

let lowerBound = 0;

return excessWater >= requiredWater;

```
while (upperBound - lowerBound > 1e-5) {
28
29
           const middleVolume = (lowerBound + upperBound) / 2;
30
31
           // Use the helper function to decide which half of the search range to keep
32
           if (canEqualizeToVolume(middleVolume)) {
33
                lowerBound = middleVolume;
```

} else {

return lowerBound;

Time and Space Complexity

**}**;

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 1 (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 0(log(Range/1e-5)), where

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

### • 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 0(n \* log(Range/1e-5)). Since 1e-5 is a constant, we can simplify this to 0(n \* log(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

- In summary: • Time Complexity: 0(n \* log(Range))
  - Space Complexity: 0(1)