2335. Minimum Amount of Time to Fill Cups

Easy · Greedy · Array · Sorting · Heap (Priority Queue) · Leetcode Link

Problem Description

You are tasked with using a water dispenser that can dispense cold, warm, and hot water to fill cups. Each type of water (cold, warm, hot) corresponds to an array position (`amount[0]`, `amount[1]`, and `amount[2]` respectively) indicating how many cups of each type you need to fill. The machine has two modes of operation:

- In one second, it can fill two cups of different types of water.
- In one second, it can fill one cup of any type of water.

The goal is to find the minimum number of seconds required to fill all the cups as per the `amount` array.

Intuition

The intuition behind the solution comes down to maximizing the efficiency of the water dispenser. Because we can fill two cups of different water types in one go, it is in our best interest to fill two cups as often as possible. One key insight is identifying that if the sum of cups needed for the two lesser needed water types (`amount[0]` and `amount[1]`) is less than or equal to the number of cups needed for the most-needed type (`amount[2]`), then the minimum time will be equal to the maximum amount among the three. This is because while we fill up cups of type 2, we can always pair them with cups from types 0 or 1 without ever having to wait.

However, in cases where the amount needed for types 0 and 1 in aggregate is more than type 2, simply matching two cups at a time is not sufficient. In such a scenario, we would divide the total sum of cups by 2, since that's the maximum throughput per second we could achieve. If there is an odd number of cups, it takes an extra second to fill the last one, hence the addition of 1 before dividing by 2. Sorting the `amount` array helps us easily identify the scenarios, ensuring that `amount[2]` is always the largest value that can potentially limit the speed at which all cups are filled.

The solution correctly computes the minimum number of seconds by considering both possibilities: either by being constrained by the largest single type (when the largest amount is greater than the sum of the other two), or by optimally using the dispenser's ability to fill two different cups in a second, evenly distributing the effort.

Solution Approach

The solution follows a simple yet effective approach to solve the problem, making use of the following steps:

1. **Sorting the Array:** Firstly, the `amount` array is sorted. By doing this, we ensure that `amount[2]` will have the largest value among the three, which is crucial for the logic that follows.

2. **Identifying Constraints:** The solution checks if the sum of the two smaller amounts (`amount[0]` + `amount[1]`) is less than or equal to the largest amount (`amount[2]`). This is an important condition because if it's true, it implies that we can always pair any cup we fill from the largest amount with another type, hence the total time will be constrained by the largest amount. In other words, while we fill the `amount[2]` cups, we'll also be able to fill the cups of types `amount[0]` and `amount[1]` in parallel without any additional time.

3. **Computing the Time:** If the previous condition is true, the function returns the value of `amount[2]`, as we're constrained by this number. If not, the solution computes the total sum of all the amounts and divides it by 2 (since two cups can be filled per second at most). If there's an odd number of cups, we'll need an extra second to fill the last one, hence the sum is incremented by 1 before dividing by 2.

4. **Returning the Result:** The result from the computation gives us the minimum number of seconds required to fill all cups, fulfilling the problem's requirement.

The solution does not require complex data structures or algorithms; it is a direct application of math and logic based on the parameters of the problem. The key to optimizing the filling process is the ability to identify when we are limited by the largest amount and when we can proceed by simply dividing the total work by the optimal throughput (two cups per second).

The overall time complexity of the algorithm is `O(1)`, since the input size is constant (always an array of 3 elements) and sorting such a small array is a constant time operation. The space complexity is also `O(1)`, as no additional space is needed regardless of the input.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Given Amounts:

Suppose the `amount` array is given as `[5, 3, 1]`, which corresponds to the number of cups that need to be filled with cold, warm, and hot water, respectively.

Step 1: Sorting the Array

First, we sort the `amount` array. After sorting, the array changes to `[1, 3, 5]`. Now, `amount[2]` is the largest value, which is 5.

Step 2: Identifying Constraints

We check if the sum of the two smaller amounts (`amount[0]` + `amount[1]`) is less than or equal to the largest amount (`amount[2]`). In our example, `1 + 3` is equal to 4, which is less than `amount[2]` which is 5.

Step 3: Computing the Time

Since the sum of the two smaller amounts (4) is less than the largest amount (5), we can conclude that the dispenser will always be actively filling a cup of the largest amount (cold water in this case) while also filling cups of the other two types. Thus, the time taken is constrained by the largest amount, `amount[2]`.

Therefore, the minimum number of seconds required to fill all the cups is equal to `amount[2]`, which is 5 seconds.

Returning the Result

The solution returns 5 as the minimum number of seconds required to fill all cups. This satisfies the goal since we utilize every second to fill two different types of cups until we run out of the type with the lesser amount, after which we are constrained by the largest amount.

Note that if the `amount` array was given as `[2, 3, 5]` instead, after sorting and summing the two smaller amounts (`2+3=5`), we see that the sum equals the largest amount (`amount[2]=5`). In this case, the solution would still return 5 seconds as the minimum time required to fill all cups. However, if the `amount` array was `[2, 6, 7]`, then the sum of the two lesser amounts is 8, which is larger than 7 (`amount[2]`). Then the total sum would be `2+6+7=15`, and we would need `15 + 1` (to account for the odd number of cups) divided by 2, which gives us 8 seconds as the minimum time to fill all the cups.

Python Solution

```python
from typing import List

class Solution:
    def fillCups(self, amounts: List[int]) -> int:
        # Sort the amount list in non-decreasing order
        amounts.sort()

        # Check if the sum of the smallest two is less than or equal to the largest
        # This means we can pair the smallest amounts with the largest to minimize time
        if amounts[0] + amounts[1] <= amounts[2]:
            # In such a case, the time required is determined by the largest amount,
            # since we will be filling that while simultaneously filling the others
            return amounts[2]

        # If not, we can pair cups to be filled in a way that minimizes the total time
        # The time required is half the sum of all amounts, rounded up
        # This is because we can always fill two cups at a time
        return (sum(amounts) + 1) // 2

# The implementation can be used as follows:
# solution = Solution()
# result = solution.fillCups([1, 3, 5])  # Example input
# print(result)  # Output will be the minimum time required to fill all cups

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting operation

class Solution {
    // Method to calculate minimum time needed to fill the cups
    public int fillCups(int[] amounts) {
        Arrays.sort(amounts); // Sort the amounts array in ascending order

        // If the sum of the two smallest amounts is equal to or less than the largest amount
        if (amounts[0] + amounts[1] <= amounts[2]) {
            // Only the largest amount matters, because while filling the largest cup,
            // the other two can be filled in parallel.
            return amounts[2];
        }

        // If the condition is not met, then return half the sum of all amounts
        // The plus one before dividing by two ensures that we round up, which is necessary
        // because we are counting discrete units of time and cannot have a partial time unit.
        return (amounts[0] + amounts[1] + amounts[2] + 1) / 2;
    }
}

```

C++ Solution

```cpp
#include <algorithm> // Include algorithm for std::sort
#include <vector>    // Include vector for using std::vector

class Solution {
public:
    int fillCups(std::vector<int>& amounts) {
        // First, sort the amounts in non-decreasing order
        std::sort(amounts.begin(), amounts.end());

        // Check if the sum of the two smallest amounts is less than or equal to the largest amount
        if (amounts[0] + amounts[1] <= amounts[2]) {
            // The largest amount determines the minimum time needed
            // because we can fill the smaller cups simultaneously while filling the largest cup.
            return amounts[2];
        } else {
            // If the sum of the two smallest amounts is greater than the largest amount,
            // we can simultaneously fill two cups at a time, and it will take more than
            // just the time for the largest cup.
            // The total time needed is the sum of all the amounts plus one (for the final pour)
            // divided by 2 because we fill two cups at a time.
            return (amounts[0] + amounts[1] + amounts[2] + 1) / 2;
        }
    }
};

```

Typescript Solution

```typescript
/**
 * Fills cups with given amounts as quickly as possible, one unit per cup per time step.
 * At each time step, you should choose two different cups to fill up by one unit each.
 * If there's only one cup with a remaining amount, you fill that cup by one unit.
 * The function returns the minimum number of time steps needed to fill all cups.
 *
 * @param {number[]} amounts - An array of three integers representing the initial amount in each of the three cups.
 * @returns {number} The minimum number of time steps necessary to fill all cups.
 */
function fillCups(amounts: number[]): number {
    // Sort the array in ascending order so that amounts[0] <= amounts[1] <= amounts[2].
    amounts.sort((first, second) => first - second);

    // Destructure the sorted array for better readability.
    let [smallCup, mediumCup, largeCup] = amounts;

    // Compute the difference between the sum of the smaller cups and the largest cup.
    let difference = smallCup + mediumCup - largeCup;

    // If the difference is non-positive, we can fill up both smaller cups and then finish with the largest one.
    if (difference <= 0) {
        return largeCup;
    } else {
        // Otherwise, calculate the number of time steps needed to equalize all cups.
        // The Math.ceil function is used to account for fractional units from the division that
        // represents time steps where only one unit can be filled.
        return Math.ceil(difference / 2) + largeCup;
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is determined by the sorting operation and the subsequent constant-time calculations.

- The `sort()` method in Python is based on the Timsort algorithm, which has a time complexity of `O(n log n)`, where `n` is the length of the list to be sorted. In this case, since the list `amount` is always of length 3 (representing the three cups), the sorting time complexity is effectively constant, often denoted as `O(1)` as it does not grow with input size.

- After sorting, the method performs a few arithmetic operations and comparisons, which are all done in constant time, `O(1)`.

Therefore, the overall time complexity of the method is `O(1)` since all operations are constant time for a fixed-size list.

Space Complexity

The space complexity is determined by the amount of additional memory used by the program as a function of the input size.

- The given code does not utilize any additional space that grows with the size of the input, as the list's size is constant and only a few additional variables are used for the computation.

- Sorting is done in place, which does not require extra space proportional to the input size.

Thus, the overall space complexity of the method is `O(1)`, representing constant space usage.