1518. Water Bottles

Easy Math Simulation

Problem Description

The LeetCode problem presents a scenario where you have a specific number of full water bottles (`numBottles`) and a market that allows you to exchange a certain number of empty water bottles (`numExchange`) for one full bottle. Every time you drink a water bottle, it becomes empty, and you have the potential to use these empty bottles to get more water from the market. The challenge is to figure out the maximum number of water bottles you can drink given the initial number of bottles you have and the exchange rate at the market.

Intuition

The intuition behind the solution is to start drinking the full water bottles you have initially, keeping track of the bottles as they become empty. Every time you drink a bottle, your total count of bottles consumed increases by one. When you have enough empty bottles to exchange for a new one, you essentially 'recycle' those empty bottles, which reduces the total count of empty bottles by the exchange rate minus one (since you get one full bottle back). You then continue the process until you no longer have enough empty bottles to make an exchange.

This approach revolves around a loop where in each iteration:

- You simulate the drinking process by adding the number of initial bottles to the total drunk count.
- You check if you have enough empty bottles to make an exchange.
- If you do, you exchange the empty ones, with the process reducing the number of empty bottles by the exchange rate minus one, adding one new full bottle to the total count, and then proceeding to drink again.
- This continues until you don't have enough empty bottles to exchange for a new one.

The solution is a greedy approach whereby at each step, you consume as many bottles as possible and exchange as soon as possible, thereby guaranteeing the maximum drinking count.

Solution Approach

The implementation of the solution uses a while loop to simulate the drinking and exchanging process until it's not possible to exchange empty bottles for a full one.

Initially, we set `ans` to `numBottles` since we can drink all those bottles at the very least. We then enter a loop that continues as long as `numBottles` is greater than or equal to `numExchange`. This condition checks whether we have enough empty bottles to make an exchange for a full one.

During each iteration of the loop:

- We simulate exchanging empty bottles for a full one by reducing `numBottles` by `numExchange - 1`. We subtract one less than `numExchange` because we are effectively giving away `numExchange` bottles and receiving one full bottle in return, resulting in a net loss of `numExchange - 1` empty bottles.
- We increment the answer (`ans`) by 1 to account for the new full bottle we obtained from the exchange, which we assume we drink immediately.

When it's no longer possible to exchange (i.e., when `numBottles` is less than `numExchange`), we exit the loop and the total `ans` represents the maximum number of bottles we could have drunk.

The algorithm uses constant space, as we only need a few integer variables to keep track of the bottles consumed and the bottles remaining, and the time complexity is O(n), where n is the initial number of bottles because we decrease `numBottles` by at least one in each iteration of the loop.

Overall, the solution counts the total number of bottles drunk by incrementally exchanging empty bottles for full ones until we can make no more exchanges, adding up the initial full bottles and the extra ones gained through exchange.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Say we have `numBottles = 9` full water bottles and the market allows us to exchange `numExchange = 3` empty bottles for 1 full bottle.

Initially, we can drink all 9 bottles, so `ans` initially becomes 9. Each time we drink a bottle, it becomes empty. After drinking these, we have 9 empty bottles.

Now, we can exchange these 9 empty bottles for 9/3 = 3 full bottles. After drinking these 3 bottles, our total `ans` is now 9 (initially drunk) + 3 (newly drunk) = 12.

Again, we have 3 empty bottles from the ones we just consumed, so we can exchange these 3 empty bottles for 1 full bottle. After drinking this one as well, our `ans` is 12 + 1 = 13.

Now we have only 1 empty bottle left, and because we have fewer empty bottles than `numExchange`, we can't exchange them for more full bottles.

Therefore, the maximum number of water bottles we can drink is 13.

In this example, the while loop runs each time we have enough empty bottles to exchange for a new full bottle. Each time, we add to our total count of drunk bottles and reduce the number of empty bottles according to the exchange rate. This process of drinking and exchanging continues until we can no longer exchange the empties for a full one, and we obtain our `ans`.

Solution Implementation

Python

```python
class Solution:
    def numWaterBottles(self, num_bottles: int, num_exchange: int) -> int:
        # Total amount of drinkable bottles is initially the same as the number of bottles.
        total_drinkable_bottles = num_bottles

        # Continue the loop as long as we have enough bottles to exchange for a new one.
        while num_bottles >= num_exchange:
            # Calculate the remaining bottles after performing an exchange.
            # The number of empty bottles to exchange is reduced by 'num_exchange',
            # but we get one new bottle in return, hence the '-1'.
            num_bottles = num_bottles - num_exchange + 1

            # Update the total number of drinkable bottles after the exchange.
            total_drinkable_bottles += 1

        # Return the total number of drinkable bottles we can have.
        return total_drinkable_bottles
```

Java

```java
class Solution {
    public int numWaterBottles(int numBottles, int numExchange) {
        // Initialize totalDrunkBottles with the number of bottles we start with
        int totalDrunkBottles = numBottles;

        // Continue the process as long as we have enough empty bottles to exchange
        while (numBottles >= numExchange) {
            // Calculate the number of new bottles we can get by exchanging
            int newBottles = numBottles / numExchange;

            // Update the count of total drunk bottles with the new bottles
            totalDrunkBottles += newBottles;

            // Update the current number of bottles we have,
            // Including the new bottles and the bottles left after the exchange
            numBottles = newBottles + (numBottles % numExchange);
        }

        // Return the total number of drunk bottles
        return totalDrunkBottles;
    }
}
```

C++

```cpp
class Solution {
public:
    // The function calculates the total number of water bottles one can drink,
    // given the initial number of bottles and the number of empty bottles required
    // to exchange for a new full water bottle.
    int numWaterBottles(int numBottles, int numExchange) {
        int totalDrunk = numBottles; // Total bottles drunk includes initial bottles
        while (numBottles >= numExchange) {
            // Each time the loop runs, we can exchanae 'numExchange' empty bottles
            // for one new full bottle. In the process, we effectively lose
            // 'numExchange - 1' empty bottles since we get one full bottle back.
            numBottles -= (numExchange - 1);

            // Increment the total drunk since we gained one bottle by exchanging
            totalDrunk++;

            // No change in amount of code here, but using loop condition for clarity
        }
        return totalDrunk;
    }
};
```

TypeScript

```typescript
// This function calculates the total number of water bottles one can drink
// given the number of initial bottles and the number of empty bottles
// required for exchanging them for a new one.
function numWaterBottles(initialBottles: number, exchangeRate: number): number {
    // The total number of water bottles one can drink.
    let totalDrinks = initialBottles;

    // Keep exchanging the bottles until there are not enough to exchange for a new one.
    while (initialBottles >= exchangeRate) {
        // Calculate the number of new bottles earned through exchange.
        const newBottles = Math.floor(initialBottles / exchangeRate);

        // Update total drinks with the new bottles obtained.
        totalDrinks += newBottles;

        // Update the count of bottles: bottles left after exchange + new bottles.
        initialBottles = newBottles + (initialBottles % exchangeRate);
    }

    return totalDrinks;
}
```

```python
class Solution:
    def numWaterBottles(self, num_bottles: int, num_exchange: int) -> int:
        # Total amount of drinkable bottles is initially the same as the number of bottles.
        total_drinkable_bottles = num_bottles

        # Continue the loop as long as we have enough bottles to exchange for a new one.
        while num_bottles >= num_exchange:
            # Calculate the remaining bottles after performing an exchange.
            # The number of empty bottles to exchange is reduced by 'num_exchange',
            # but we get one new bottle in return, hence the '-1'.
            num_bottles = num_bottles - num_exchange + 1

            # Update the total number of drinkable bottles after the exchange.
            total_drinkable_bottles += 1

        # Return the total number of drinkable bottles we can have.
        return total_drinkable_bottles
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be represented as `O(n)`, where `n` represents the initial number of bottles. This is because with each iteration we effectively simulate drinking a bottle of water and then exchanging the empty bottles for a full one. The loop continues until the number of bottles is less than `numExchange`, which happens after `n / (numExchange - 1)` iterations, where integer division is implied.

Space Complexity

The space complexity of the code is `O(1)` as there are only a few integer variables allocated (`ans`, `numBottles`, `numExchange`), and no additional data structures that grow with the size of the input are used. The amount of memory used is constant regardless of the input size.