2952. Minimum Number of Coins to be Added

Medium Greedy Array Sorting

Problem Description

In this problem, you are given an array of integers `coins` which represents different coin denominators and an integer `target`. Your goal is to determine the smallest number of additional coins of any denomination that you must add to the `coins` array so that it becomes possible to form every integer value in the inclusive range from 1 to `target`.

This means that if you take any integer `x` within the range `[1, target]`, you should be able to choose a subsequence of coins from the updated `coins` array that adds up exactly to `x`. The term "subsequence" here refers to a sequence that can be derived from the `coins` array by removing some or none of the coins, without changing the order of the remaining coins.

Intuition

The intuition behind the solution is using a greedy approach. The key idea is to always ensure that we can form all amounts leading up to the current target amount `s`. Initially, we can form the amount `0` with no coins.

As we increment `s`, we want to ensure that we can make all amounts up to the current `s`. We sort the `coins` to process them in ascending order. When we consider a new coin value, two scenarios can occur:

1. The coin value is less than or equal to `s`: in this case, by adding the coin to our collection, we can now form amounts in the range of `[s, s+coin value - 1]`. This happens because we can already form all amounts up to `s-1`, and with the new coin value, we can extend the range to that coin value.
2. The coin value is greater than `s`: If we encounter a coin with a value larger than `s`, we cannot increment `s` using it, as there is a gap in the amounts we can form. To bridge this gap, we pretend to add a new coin of value exactly `s` to our collection, which then allows us to form all amounts up to `2*s - 1`. In practical terms, this means we increment `s` to `2*s` (doubling it) and increase the count of additional coins needed (`ans`).

We continue this process until `s` exceeds `target`. The variable `ans` keeps track of the number of additional coins we are injecting into the collection to maintain the required sequence.

Solution Approach

The solution provided is a direct implementation of the greedy approach described earlier. Here's a step-by-step explanation of how the algorithm is applied:

1. **Sorting the Coins**: First, we sort the `coins` array in non-decreasing order. This is important because it allows us to process the coins from the smallest to the largest, which is crucial for the greedy approach.

```
coins.sort()
```

2. **Initialization**: We initialize two variables: `s` starting at `1` representing the current target amount we can form, and `ans` starting at `0` indicating the number of additional coins needed. We also have an index `i` starting at `0` for iterating over the `coins` array.

```
s = 1
ans = i = 0
```

3. **Iterating Through the Coins**:

```
while s <= target:
```

We loop until `s` exceeds or equals `target`. This is our stopping condition, ensuring we can create all amounts from 1 to `target`.

4. **Using existing coins if they are less than or equal to `s`**: Inside the loop, we check whether we are still within bounds of the `coins` array and whether the current coin is less than or equal to the current amount `s` that we want to construct.

```
if i < len(coins) and coins[i] <= s:
    s += coins[i]
    i += 1
```

If so, we add the value of that coin to `s`, effectively merging it into our running total, and move to the next coin.

5. **Injecting Additional Coins**: When we encounter a coin value greater than `s` (or run out of provided coins), it means we have a gap in the amounts we can create. We simulate adding an additional coin of value `s` to our collection.

```
else:
    s <<= 1
    ans += 1
```

The operation `s <<= 1` doubles `s`, and we increase the `ans` counter to account for the additional coin we've theoretically added.

By the time we exit the loop, `ans` will hold the minimum number of additional coins required for making every amount up to `target` obtainable. The final step is simply to return the `ans`.

```
return ans
```

Throughout the implementation, we are using basic list operations for sorting and indexing, with no need for complex data structures. The pattern used in this algorithm is inherently greedy as it always "fills the gap" with the smallest possible coin needed at each step while using existing coins whenever possible.

Example Walkthrough

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

Suppose we are given the array `coins = [1, 2, 5]` and the `target` value is `11`. We want to find the minimum number of additional coins that will enable us to make every integer value from `1` to `11`.

Following the solution approach:

1. **Sorting the Coins**: First, we sort the array `coins`, which in this example is already sorted as `[1, 2, 5]`.
2. **Initialization**: We set `s = 1`, `ans = 0`, and `i = 0`. `s` is the current target amount we can form, `ans` is the additional coins needed, and `i` is the index for the `coins` array.
3. **Iterating Through the Coins**: We start iterating through the array with the condition `while s <= target`:
   - `s = 1` and `coins[i] = 1` (since `i = 0`): We add `coins[i]` to `s`, getting `s = 2`, and increment `i` to `1`.
4. **Using existing coins if they are less than or equal to `s`**:
   - `s = 2` and `coins[i] = 2` (since `i = 1`): We add `coins[i]` to `s`, getting `s = 4`, and increment `i` to `2`.
   - `s = 4` and `coins[i] = 5` (since `i = 2`): Here, `coins[i]` is greater than `s`, which means we cannot use this coin directly to form `s`. We proceed to the next step.
5. **Injecting Additional Coins**:
   - Since `coins[i]` is greater than `s`, we add an additional coin of value `s` to the collection which in this step `s = 4`, now `s` becomes `8` (`s <<= 1`), and we increment `ans` to `1`.
   - The loop continues, `s = 8` <= `target`, but `i` has exceeded the length of coins, so we consider it as encountering a coin greater than `s` again, thus we double `s` to `16` and increment `ans` again to `2`.
   - Now `s = 16` which is greater than `target`, so the loop terminates.

The final returned value of `ans` is `2`, indicating that we need to add two additional coins of denominations that we need to decide (in this example, additional coins of denominations `4` and `8` would work) to be able to make every amount from `1` to `11` with the given `coins` array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minimumAddedCoins(self, coins: List[int], target: int) -> int:
        # Sort the given list of coins in ascending order
        coins.sort()

        # Initialize the current sum of coins to 1, as the smallest coin we can add
        current_sum = 1

        # Initialize the answer (the number of coins to add) and index counter for the coins list
        additional_coins_count = 0
        coin_index = 0

        # Continue until the current sum reaches the target value
        while current_sum <= target:
            # If there are more coins to consider and the next coin is less than or
            # equal to the current sum, it means we can include this coin without missing any value
            if coin_index < len(coins) and coins[coin_index] <= current_sum:
                current_sum += coins[coin_index]  # Add the value of the current coin to the sum
                coin_index += 1  # Move to the next coin
            # If no appropriate coin is found, we have to add a new coin
            else:
                # We double the current sum because by adding a coin with a value equal to the current sum,
                # we ensure that we can reach all the sums up to twice the current sum.
                current_sum <<= 1

                # Increment the answer as we decide to add a new coin
                additional_coins_count += 1

        # Return the total number of additional coins needed to make the target value
        return additional_coins_count
```

Java

```java
class Solution {
    public int minimumAddedCoins(int[] coins, int target) {
        // Sort the array of coins to start with the smallest denomination.
        Arrays.sort(coins);

        // Initialize variables:
        // - 'ans' will hold the count of additional coins required.
        // - 'currentSum' represents the current sum we can create with the existing coins.
        // - 'i' is used to iterate through the coins array.
        int ans = 0;
        int currentSum = 1; // We start from 1 because we want to be able to create sums starting from 1 up to 'target'.

        // Iterate while the current sum is less than or equal to the target sum.
        while (currentSum <= target) {
            // If there are still coins left, and the current coin's value is less than or equal to the current sum,
            // we can use this coin to increase the current sum.
            if (i < coins.length && coins[i] <= currentSum) {
                currentSum += coins[i]; // Add the coin's value to the current sum.
                i++; // Move on to the next coin.
            } else {
                // If there are no coins left that can be used, or the next coin's value is too large,
                // double the current sum. This corresponds to adding a coin with value equal to the current sum.
                currentSum <<= 1; // This is equivalent to 'currentSum = currentSum * 2'.
                ans++; // Increment the count of additional coins needed.
            }
        }

        // Return the total number of additional coins needed to create all sums from 1 up to the target.
        return ans;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum number of additional coins needed to make up to the target amount.
    int minimumAddedCoins(vector<int>& coins, int target) {
        // Sort the coins in increasing order.
        sort(coins.begin(), coins.end());

        // Initialize answer (additional coins needed) to 0.
        int additionalCoins = 0;

        // Initialize current amount to 1 (the minimum amount we can make with additional coins).
        int currentSum = 1;

        // Initialize the index for iterating through the sorted coins vector.
        int coinIndex = 0;

        // Continue until the current amount is less than or equal to the target amount.
        while (currentSum <= target) {
            // If we have not used all coins and the current coin is less than or equal to current amount,
            // we can use this coin to make the current amount.
            if (coinIndex < coins.size() && coins[coinIndex] <= currentSum) {
                currentSum += coins[coinIndex++]; // Use the coin and increment the coinIndex.
            } else {
                // Otherwise, we need to add a coin that is double the current amount
                // because it's the smallest denomination not yet used.
                currentSum <<= 1; // Doubling the current amount.
                additionalCoins++; // Increment the count of additional coins needed.
            }
        }

        // Return the total number of additional coins needed.
        return additionalCoins;
    }
};
```

TypeScript

```typescript
function minimumAddedCoins(coins: number[], target: number): number {
    // Sort the coins array in ascending order to process them from smallest to largest
    coins.sort((a, b) => a - b);

    let additionalCoins = 0; // Keep track of the number of additional coins needed
    let currentSum = 1; // Initialize the current sum to 1, as 0 wouldn't be helpful for addition

    // Iterate through the sorted coins array
    // and simultaneously track the coverage of sums until we reach the target
    for (let i = 0; currentSum <= target; ) {
        // If the current coin is less than or equal to the current sum,
        // add the coin's value to the sum and move to the next coin
        if (i < coins.length && coins[i] <= currentSum) {
            currentSum += coins[i++];
        } else {
            // If the current coin is bigger than the current sum,
            // double the current sum (this is equivalent to adding a power-of-two coin)
            // and increment the additionalCoins to indicate a new coin was added
            currentSum <<= 1;
            additionalCoins++;
        }
    }

    // Return the final count of additional coins needed to reach the target sum
    return additionalCoins;
}
```

```python
from typing import List

class Solution:
    def minimumAddedCoins(self, coins: List[int], target: int) -> int:
        # Sort the given list of coins in ascending order
        coins.sort()

        # Initialize the current sum of coins to 1, as the smallest coin we can add
        current_sum = 1

        # Initialize the answer (the number of coins to add) and index counter for the coins list
        additional_coins_count = 0
        coin_index = 0

        # Continue until the current sum reaches the target value
        while current_sum <= target:
            # If there are more coins to consider and the next coin is less than or
            # equal to the current sum, it means we can include this coin without missing any value
            if coin_index < len(coins) and coins[coin_index] <= current_sum:
                current_sum += coins[coin_index]  # Add the value of the current coin to the sum
                coin_index += 1  # Move to the next coin
            # If no appropriate coin is found, we have to add a new coin
            else:
                # We double the current sum because by adding a coin with a value equal to the current sum,
                # we ensure that we can reach all the sums up to twice the current sum.
                current_sum <<= 1

                # Increment the answer as we decide to add a new coin
                additional_coins_count += 1

        # Return the total number of additional coins needed to make the target value
        return additional_coins_count
```

Time and Space Complexity

The time complexity of the code is primarily determined by the sorting operation and the while loop. The sort function has a complexity of $O(n \log n)$, where n is the number of coins. After sorting, the while loop runs in $O(n)$ since it iterates over the sorted list of coins once. However, the `s <<= 1` operation is logarithmic with respect to the `target`, causing a $O(\log target)$ complexity. Since there could be multiple doublings until `s` exceeds `target`, and considering the combined operations, the overall time complexity would be $O(n \log n + \log target)$.

In terms of space complexity, the sort operation can be done in-place, which means the space used is constant with respect to the input. However, the source of additional space complexity is the internal implementation of the sort function which typically uses $O(\log n)$ space on the call stack. Hence, the space complexity of the code is $O(\log n)$.

Please note that the reference answer mentions space complexity as $O(\log n)$ which aligns with the typical space required for the sort's call stack in its internal implementation. However, if the sort were to be implemented in a way that is strictly in-place and without extra space usage (such as with an in-place variation of heapsort), the space complexity would be constant, $O(1)$.