1561. Maximum Number of Coins You Can Get

Medium Greedy Array Math Game Theory Sorting

Problem Description

In this problem, we have `3n` piles of coins of different sizes, which means the total number of piles is divisible by 3. You are playing a game with two other friends, Alice and Bob. You will take turns choosing any 3 piles of coins. During each turn:

1. Alice will always take the pile with the largest number of coins.
2. You will take the next largest pile.
3. Bob will take the pile with the smallest number of coins.

This process repeats until there are no more piles left. Your objective is to maximize the number of coins you get by the end of the game. The question provides us with an array of integers `piles`, where `piles[i]` represents the number of coins in the `i-th` pile, and we need to return the maximum coins you can get under the game's rules.

Intuition

The intuition behind the solution is to use a greedy approach. A greedy algorithm is one that always makes the choice that looks best at the moment without considering the future outcomes. Applying this to our scenario, since Alice will always take the largest pile and Bob the smallest, the best strategy for you is to ensure that each time you choose 3 piles, the second largest pile is as large as possible, without ever being the largest.

To ensure that, the best approach is to sort the piles in ascending order. By doing so, we can then pair the largest available pile for Alice with the next largest pile for ourselves, which leaves the smallest pile in this subset for Bob. By choosing from the larger end of the sorted piles, we can maximize the coins we get.

After sorting the piles, we can iterate through the sorted array taking the second largest element from the end in each step until we have picked `n` piles ourselves. To do so:

1. We start from the second element from the end, as this would have been the largest pile if we picked any three piles.
2. We move two steps towards the front of the array each time to ensure we are always picking the next pile with the maximum number of coins available for us.
3. We stop picking after we have selected `n` piles for ourselves, which happens after `n` iterations as there are `3n` piles and we are picking one out of every three piles.

Therefore, the sum of the selected piles would give us the maximum coins we can have.

Solution Approach

The solution to this problem leverages the `sort` function to order the array of piles and then accumulate the coins that you would receive according to the rules established by the game.

Here is a step-by-step breakdown of the implementation:

1. **Sorting the Piles**: First, we use Python's built-in sort function `piles.sort()` which sorts the array of piles in ascending order. Sorting is a common pre-processing step in greedy algorithms. No additional data structure is used here; the array is sorted in place.
2. **Slicing the Sorted Piles**: After sorting, the largest piles are at the end of the array. By employing Python's slicing, we can get every other pile starting from the second-to-last element using the slice `piles[-2 : len(piles) // 3 - 1 : -2]`.
   - `piles[-2]` is the starting point, which represents the second largest pile because the largest pile would be taken by Alice.
   - `len(piles) // 3 - 1` is the stopping point, which ensures that only the piles meant for you are included, leaving out the smallest third of the piles, which are meant for Bob.
   - `-2` is the step, which allows us to skip every alternate pile to simulate the take turns in which you would always end up with the second largest pile out of every chosen set of three piles.
3. **Summation**: Lastly, we apply the `sum()` function on the sliced array to get the total number of coins accumulated across all chosen piles. This sum represents the maximum number of coins you can obtain by following this strategy.

The algorithm exhibits a greedy property since in every turn, it picks the local optimum (the second biggest pile of every chosen set of three piles) without considering the rest of the array. This local optimum choice leads to a global optimum, which is the maximum number of coins you can achieve.

The complexity of this algorithm is primarily determined by the sorting step. Python's sort function has a time complexity of O(n log n). The slicing step is O(n), where `n` is the total number of piles divided by 3, since we're summing every second pile out of the two-thirds of the piles. So, the overall time complexity is dominated by the sorting step, making it O(n log n).

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Assume we have an array of piles `piles = [2, 4, 1, 2, 7, 8]`. There are `3n = 6` piles, which implies that `n = 2`. According to the rules, Alice will choose the largest pile, you will choose the second-largest pile, and Bob will choose the smallest pile each turn.

Here is how you can follow the steps outlined in the solution approach:

1. **Sorting the Piles**: First, sort `piles` to get `[1, 2, 2, 4, 7, 8]`.
2. **Slicing the Sorted Piles**: To simulate taking turns:
   - Start from the second-to-last element to avoid the largest pile, which Alice takes. This gives us a starting point of `piles[-2]`, which is `7`.
   - End at `len(piles) // 3 - 1` which is `1` here, as you only need to collect `n` piles and the last third are the smallest ones taken by Bob.
   - Use a step of `-2` to simulate the turns, meaning you take every other pile from the end of the sorted `piles`.

   Following this slicing, we would pick the piles at indices `-2` and `-4` which correspond to the values `[7, 4]` in the original sorted `piles`.
3. **Summation**: Summing the slice `[7, 4]` gives us `11`. This is the maximum number of coins you can gather following this strategy.

Therefore, by applying the greedy algorithm, we were able to maximize your share of the coins to `11`. This example validates the approach described and shows how sorting, slicing, and summing in a strategic manner results in an optimal solution for you in this coin pile game.

Solution Implementation

Python

```python
class Solution:
    def maxCoins(self, piles):
        # Sort the piles in ascending order.
        piles.sort()

        # Initialize the total coins collected.
        total_coins = 0

        # We start taking coins from the second largest pile (hence -2 index)
        # and continue with every second pile from the end until we reach
        # the point which is just beyond one third of the piles count from the end.
        # This allows us to simulate the process of taking piles from the end,
        # skipping the smallest one each time and ignoring the smallest third.
        for i in range(len(piles) // 3, len(piles), 2):
            total_coins += piles[i]

        # Return the total coins collected.
        return total_coins
```

Java

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

class Solution {

    /**
     * Calculate the maximum number of coins you can get.
     * @param piles Array of piles where each pile has a certain number of coins.
     * @return The maximum coins that can be taken.
     */
    public int maxCoins(int[] piles) {
        // Sort the array to arrange the piles in ascending order
        Arrays.sort(piles);

        // Initialize answer to 0, this will hold the sum of coins picked
        int maxCoins = 0;

        // Start from the second largest pile and move backwards in steps of 2
        // This approach ensures that we take the second largest piles leaving the smallest ones
        for (int i = piles.length - 2; i >= piles.length / 3; i -= 2) {
            // Add the number of coins from the current selected pile to the answer
            maxCoins += piles[i];
        }

        // Return the calculated maximum number of coins
        return maxCoins;
    }
}
```

C++

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

class Solution {
public:
    // Function to maximize the number of coins you can get.
    int maxCoins(vector<int>& piles) {
        // Sort the piles in ascending order.
        std::sort(piles.begin(), piles.end());

        int totalCoins = 0; // This will store the total number of coins you can get.

        // Start from the second-largest pile and move two steps backward
        // each time to get to the next pile you can take.
        for (int i = piles.size() - 2; i >= piles.size() / 3; i -= 2) {
            totalCoins += piles[i]; // Add the coins from the current pile to your total.
        }

        // Return the total number of coins.
        return totalCoins;
    }
};
```

TypeScript

```typescript
function maxCoins(piles: number[]): number {
    // Sort the array in ascending order.
    piles.sort((a, b) => a - b);

    // Get the total number of piles.
    const totalPiles = piles.length;

    // Initialize the maximum number of coins.
    let maxCoins = 0;

    // Calculate the number of times we can pick piles according to the rules.
    const timesToPick = Math.floor(totalPiles / 3);

    // Iterate to pick the second-largest pile from each triplet from the end.
    for (let i = 1; i <= timesToPick; i++) {
        // The index for the second-largest pile in each triplet
        const index = totalPiles - 2 * i;

        // Accumulate the number of coins from the chosen piles.
        maxCoins += piles[index];
    }

    // Return the maximum number of coins we can collect.
    return maxCoins;
}
```

```python
class Solution:
    def maxCoins(self, piles):
        # Sort the piles in ascending order.
        piles.sort()

        # Initialize the total coins collected.
        total_coins = 0

        # We start taking coins from the second largest pile (hence -2 index)
        # and continue with every second pile from the end until we reach
        # the point which is just beyond one third of the piles count from the end.
        # This allows us to simulate the process of taking piles from the end,
        # skipping the smallest one each time and ignoring the smallest third.
        for i in range(len(piles) // 3, len(piles), 2):
            total_coins += piles[i]

        # Return the total coins collected.
        return total_coins
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined primarily by the sort operation and the slicing to sum up the required coins.

1. Sorting the `piles` list is the most computationally heavy operation. The sort function in Python typically uses Timsort, which has a time complexity of O(n log n), where `n` is the number of elements in the list to sort.
2. The slicing and summing operations occur after sorting. Slicing in Python is O(k), where `k` is the number of elements in the slice. However, in this case, the code is slicing from the second-to-last element down to a third of the list's length, creating a subsequence. The sum function operates over this subsequence. Its complexity is O(m), where `m` is the length of the subsequence.

The value of `m` is approximately `(2/3)n/2` elements because we start at the second to last element and take every second element till we reach the first third of the array. Thus, the slice and sum operation would have a complexity of O(n).

Combining both the sorting and the summation complexities, the time complexity of the code is O(n log n) + O(n), which simplifies to O(n log n) because the sorting complexity dominates the overall time complexity.

Space Complexity

The space complexity of the code consists of the additional space required for the sorted array and space for the variables used in summing operation:

1. The `.sort()` method sorts the list in place and therefore does not require additional space, so this contributes O(1) space complexity.
2. Slicing does not create a new list; it creates a view on the existing list, which also is O(1) space complexity since it's not duplicating the entire list.
3. Variable space for the sum operation is negligible and constant, so adds O(1).

Considering all the extra space required, the overall space complexity of the function is O(1), since no additional space proportional to the input size is created.