2944. Minimum Number of Coins for Fruits

Medium Topics Array Dynamic Programming Monotonic Queue Heap (Priority Queue)

Problem Description

The given problem presents a scenario where you're at a fruit market considering the purchase of exotic fruits. You have an array `prices` representing the cost of each fruit, indexed starting from 1. The special offer at the market is that if you buy the `i`th fruit, you can get the next `i` fruits for free. However, one can still choose to purchase a fruit that could be gotten for free, in order to activate the offer from that fruit and get additional fruits for free.

The goal is to find the minimum number of coins required to buy all the fruits. It's important to note that the strategy might involve sometimes purchasing a fruit, even when it could be taken for free, to benefit from the subsequent offers.

Intuition

Solution Approach

To find the minimum coins needed, we can use a recursive strategy together with memoization to avoid recalculating subproblems. The main idea is to approach the problem from the start and, for each fruit we consider purchasing, explore all the possibilities of the subsequent offers it might unlock. Then, we choose the path that leads to the minimum cost.

The function `dfs(i)` defines the cost of purchasing fruits starting from the `i`th fruit onward. If choosing to buy fruit `i` at `prices[i - 1]` coins (since the array is 1-indexed), you can then skip `i` fruits (because they are free) and consider the next `i` fruits, thus we recursively calculate `dfs(j)` for `j` ranging from `i+1` to `i*2+1`. We are looking for the minimal cost among all these possible `j` paths.

When `i * 2` is greater or equal to the length of the `prices` array, it means buying the `i`th fruit would give us all remaining fruits for free, so in that case, we just return the price of the `i`th fruit. By recording our results with `@cache`, we save the results of subproblems, speeding up the recursive process since we won't recompute the cost for any starting index more than once.

This solution leverages dynamic programming to break the problem down into overlapping subproblems and ensures that we only calculate each subproblem once, thus efficiently arriving at the overall minimum cost.

Solution Approach

The solution implements a Depth-First Search (DFS) strategy with dynamic programming and memoization for optimization. Let's go through each piece of the solution to understand how it's implemented:

Depth-First Search (DFS)

The core of the solution is a recursive DFS function that explores all possible options to minimize the cost. The function `dfs(i)` represents the cost of acquiring all fruits starting from the `i`th index. The `dfs` function is defined to operate on the assumption that, if we purchase fruit `i`, we instantly receive the next `i` fruits for free as per the given market offer. We then call `dfs` recursively to accumulate the minimum cost needed for the remaining fruits.

Memoization

To ensure that we don't recompute the minimum cost for the same starting index more than once, the solution uses the `@cache` decorator from Python. This decorator automatically saves the results of the `dfs` function calls into a cache. Subsequent calls with the same arguments will return the stored result instead of recomputing it. Thus, it converts our DFS into a dynamic programming approach that reuses previously computed values.

Implementation Details

The algorithm begins by defining the `dfs` function with the following logic:

- It checks if `i` is such that buying the `i`th fruit would result in acquiring all remaining fruits for free (when `i * 2 >= len(prices)`). In this case, it returns the price of the `i`th fruit since no further purchases are necessary.
- Otherwise, it calculates the sum of the price of the `i`th fruit and the minimum cost among all possible next steps. The next steps are given by the range `i + 1` to `i * 2 + 1`. It does this by calling `dfs(j)` for each `j` in this range and choosing the minimum result via the `min` function.

The solution starts by calling `dfs(1)` which indicates that we begin by considering the first fruit and recursively determine the minimum cost to acquire all fruits from there.

Algorithm Complexity

The time complexity of this algorithm is better than the naive recursive approach due to memoization, which reduces the number of unique recursive calls. The exact time complexity can be hard to determine due to the varying nature of recursive calls depending on the input; however, it is significantly optimized by avoiding redundant calculations.

To summarize, the solution uses a combination of DFS for traversing different combinations, memoization for optimization, and dynamic programming concepts to efficiently solve the problem of acquiring all the fruits for a minimum cost.

Example Walkthrough

Let's consider a small example where the array `prices` is `[3, 5, 1, 4, 2]`. This array represents the cost of each fruit, with the index starting from 1.

We want to find the minimum number of coins required to purchase all the fruits, following the market offer that allows us to get `i` next fruits for free when we purchase the `i`th fruit.

We use the DFS approach along with memoization to solve this. We start by calling the function `dfs(1)`.

Here's how it will work step by step:

1. **Calling `dfs(1)`:** We are considering buying the first fruit, which costs 3 coins.
   - If we purchase it, we get to pick the next fruit for free, which means we can skip considering the second fruit.
   - Then, we consider the minimum cost of buying fruits starting from indices 2 through 3 (`dfs(2)` and `dfs(3)`). We need to calculate the minimum cost from acquiring fruits from these starting points.
2. **Calling `dfs(2)`:** The cost of the second fruit is 5 coins.
   - This will offer us the next 2 fruits for free (3rd and 4th fruits get skipped).
   - Now, we calculate the cost of starting from the 5th fruit (`dfs(5)`), which will just be the cost of the 5th fruit itself.
3. **Calling `dfs(3)`:** The cost of the third fruit is 1 coin.
   - Purchasing this fruit, we get the next 3 fruits for free, and since it only leaves the 5th fruit, the cost here is just 1 coin.
4. **Calling `dfs(5)`:** Directly returns the cost of the 5th fruit (2 coins), since there are no more fruits after this.

Now let's calculate the total costs for each path:

- If we take the path of `dfs(1) -> dfs(2) -> dfs(5)`:
  - The cost will be `3 + 5 + 2 = 10` coins.
- If we take the path of `dfs(1) -> dfs(3)`:
  - The cost will be `3 + 1 = 4` coins (because buying the 3rd fruit yields all remaining fruits for free).

So, the minimum number of coins required is 4, which is the result of `dfs(1) -> dfs(3)`.

With memoization, the results of `dfs(2)` and `dfs(5)` are stored in the cache. If there were other recursive calls that needed these results, the algorithm would not recompute them but retrieve them from the cache, thereby saving time and making the algorithm more efficient. This is how the DFS and memoization work together to find the minimum cost in this scenario.

Solution Implementation

Python

```python
# Import the 'functools' library to use the 'lru_cache' decorator for memoization
from functools import lru_cache

class Solution:
    def minimumCoins(self, prices: List[int]) -> int:
        # Decorator to provide memoization to the dfs function
        # This will store results of subproblems to avoid recomputation
        @lru_cache(maxsize=None)
        def dfs(index: int) -> int:
            # If the current index * 2 is greater than or equal to the length of 'prices',
            # then we're at a leaf node and should return the price at this index.
            if index * 2 >= len(prices):
                return prices[index - 1]  # -1 because the prices are 1-indexed

            # Otherwise, recursively calculate the minimum cost to reach each child,
            # and add the price at the current index to it.
            # Find the minimal total cost among all possible next steps.
            return prices[index - 1] + min(
                dfs(child_index)
                for child_index in range(index + 1, index * 2 + 2)
            )

        # Start the recursive function from the first index (1-indexed)
        return dfs(1)
```

Java

```java
class Solution {
    // Member variables with more conventional naming
    private int[] prices;
    private int[] memoizationArray;
    private int numOfPrices;

    // Method to calculate the minimum number of coins needed, starting with a given price array
    public int minimumCoins(int[] prices) {
        numOfPrices = prices.length;
        memoizationArray = new int[numOfPrices + 1];
        this.prices = prices;
        // Start the recursive depth-first search from the first index (1-based index)
        return dfs(1);
    }

    // Helper method for depth-first search
    // Each call calculates the minimum price from the given index 'i'
    private int dfs(int i) {
        // If the current index is in the last half of the array,
        // a single coin of the corresponding value is used
        if (i * 2 >= numOfPrices) {
            return prices[i - 1];  // Return the face value as we have reached the end
        }

        // If this state has not been calculated before, calculate it
        if (memoizationArray[i] == 0) {
            // Initialize to a very high value to ensure proper min comparison
            memoizationArray[i] = Integer.MAX_VALUE;

            // Explore all possible next states
            for (int j = i + 1; j <= i * 2 + 1; ++j) {
                // Minimize the coins by making the best choice at each step
                memoizationArray[i] = Math.min(memoizationArray[i], prices[i - 1] + dfs(j));
            }
        }

        // Return the minimum coins found for this state
        return memoizationArray[i];
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to compute the minimum number of coins needed
    // It takes a vector of prices as an argument
    int minimumCoins(vector<int>& prices) {
        int n = prices.size(); // Get the number of elements in prices
        vector<int> minCoins(n + 1, INT_MAX); // Initialize the minCoins DP array with a high value (INT_MAX)

        // Recursive function to find the minimum coins required.
        // It uses memoization to store the results of subproblems.
        function<int(int)> computeMinCoins = [&](int currentIndex) {
            // Base case: if the current index is more than half the size
            // we are at a leaf node and return its price
            if (currentIndex * 2 >= n) {
                return prices[currentIndex - 1];
            }
            // If the current index has not been previously computed
            if (minCoins[currentIndex] == INT_MAX) {
                // Iterate over the range of possible next indices within the problem constraints
                for (int nextIndex = currentIndex + 1; nextIndex <= currentIndex * 2 + 1; ++nextIndex) {
                    // Update the minimum coins for the current index by comparing the previously stored value
                    // with the sum of the current index's price and the computed minimum coins required
                    // for the next index (computed recursively)
                    minCoins[currentIndex] = min(minCoins[currentIndex], prices[currentIndex - 1] + computeMinCoins(nextIndex));
                }
            }
            // Return the minimum coins required for the current index
            return minCoins[currentIndex];
        };

        // Call the recursive function with the starting index of 1
        return computeMinCoins(1);
    }
};
```

TypeScript

```typescript
// Function to determine the minimum total cost to buy all the products
function minimumCoins(prices: number[]): number {
    // The number of prices given in the input array
    const numOfPrices = prices.length;

    // Cache array to store minimum cost for calculated states to avoid repeated calculations
    const memo: number[] = Array(numOfPrices + 1).fill(0);

    // Helper function for Depth-First Search (DFS) to find minimum total cost
    const dfs = (currentIndex: number): number => {
        // Base case: when the double of the current index is out of array bounds
        if (currentIndex * 2 >= numOfPrices) {
            return prices[currentIndex - 1]; // Return the item's price at the current index
        }

        // Check whether the current state already has computed minimum cost
        if (memo[currentIndex] === 0) {
            // Initialize memo[currentIndex] with a large number
            memo[currentIndex] = Number.MAX_SAFE_INTEGER;

            // For every next index within the allowed range (not more than double the current index)
            for (let nextIndex = currentIndex + 1; nextIndex <= currentIndex * 2 + 1; ++nextIndex) {
                // Recursive call to find the minimum cost and update the memo array
                memo[currentIndex] = Math.min(memo[currentIndex], prices[currentIndex - 1] + dfs(nextIndex));
            }
        }

        // Return the minimum cost for the current index
        return memo[currentIndex];
    };

    // Call DFS starting from the first element
    return dfs(1);
}

# Import the 'functools' library to use the 'lru_cache' decorator for memoization
from functools import lru_cache

class Solution:
    def minimumCoins(self, prices: List[int]) -> int:
        # Decorator to provide memoization to the dfs function
        # This will store results of subproblems to avoid recomputation
        @lru_cache(maxsize=None)
        def dfs(index: int) -> int:
            # If the current index * 2 is greater than or equal to the length of 'prices',
            # then we're at a leaf node and should return the price at this index.
            if index * 2 >= len(prices):
                return prices[index - 1]  # -1 because the prices are 1-indexed

            # Otherwise, recursively calculate the minimum cost to reach each child,
            # and add the price at the current index to it.
            # Find the minimal total cost among all possible next steps.
            return prices[index - 1] + min(
                dfs(child_index)
                for child_index in range(index + 1, index * 2 + 2)
            )

        # Start the recursive function from the first index (1-indexed)
        return dfs(1)
```

Time and Space Complexity

The given Python code takes advantage of recursion with memoization to find the minimum number of coins needed from a list of prices. The function `dfs` is decorated with `@cache`, which stores the results of previous computations to avoid redundant calculations.

Time Complexity:

- For each call to `dfs(i)`, the function iterates from `i + 1` to `i * 2 + 1`, which results in a maximum of `i` recursive calls in the worst case.
- Since memoization is used, each state `dfs(i)` for `i = 1` to `len(prices) // 2` is computed only once.
- The total number of states is bound by $O(n)$, where n is the length of the `prices` list. This is because the depth of recursion is at most $\log(n)$ due to the condition `i * 2 >= len(prices)` that halts deeper recursive calls.
- Therefore, the time complexity is $O(n * \log(n))$, as each state can result in a linear number of calls based on the current index, and the depth is logarithmic relative to n.

Space Complexity:

- Memoization will require additional space for caching the results. At most, there will be $O(n)$ cache entries because each entry corresponds to a unique starting index `i`.
- The maximum depth of the recursive call stack will also be $O(\log(n))$ due to the nature of the problem, as the recursion depth is determined by how many times the index can be doubled before reaching `len(prices)`.
- Combining the space needed for memoization and the maximum recursion depth, we have $O(n)$ space complexity for caching and $O(\log(n))$ space complexity for the call stack, leading to a total space complexity of $O(n)$.

In conclusion, the time complexity of the code is $O(n * \log(n))$, and the space complexity is $O(n)$.