1475. Final Prices With a Special Discount in a Shop

Easy Stack Array Monotonic Stack Leetcode Link

Problem Description

You are provided with an array called `prices` where each element `prices[i]` indicates the cost of the `i-th` item in a store. This store offers a special discount on each item. When you buy the `i-th` item, you get a discount equal to the value of `prices[j]`, where `j` is the smallest index greater than `i` such that `prices[j]` is less than or equal to `prices[i]`. If there's no such item that fulfils the condition for the discount, then no discount is applied to the `i-th` item. The problem requires you to determine the actual price you would pay for each item after applying the special discount if applicable and return this as a new array of prices.

Intuition

The intuition behind the solution involves a stack to keep track of the items (by their indices) that we have not yet found a smaller price for. This is an optimization technique to make finding the next smaller price more efficient.

As we iterate over the array, we compare the current price with the price at the top of the stack:

- If the current price is less than or equal to the price of the item at the top of the stack, it means we found a smaller price. Therefore, the item at the top of the stack is eligible for a discount equal to the current price. We then update the answer for the item at the top of the stack by subtracting the current price from it and pop this index from the stack.
- If the current price is higher, it means this item could potentially offer a discount to the following items. So, we add the current index to the stack to signify that we have not yet found a smaller price for the item at this index.
- We continue this process until we've looked at all the prices and the stack has ensured all items received their due discounts where applicable.

This approach leverages the property of monotonically decreasing stacks to optimize the process of finding the next smaller element, which makes it possible to solve the problem in linear time.

Solution Approach

The solution utilizes a data structure called a stack to efficiently track indices of prices that are awaiting a discount. The stack property of "last-in, first-out" is particularly helpful here because it allows us to always consider the most recent item when looking for the next lower price.

Here is how the algorithm works:

- We initialize an empty stack `stk` and a list `ans` that is a copy of the original `prices` list.
- We iterate over each item in `prices` using its index `i` and value `v`. For each item, we check whether a discount can be applied based on the items in the stack.
- Inside the loop, there's an inner `while` loop that checks if the stack is not empty and whether the price at the top of the stack (the last item that was put into the stack) is greater than or equal to the current price `v`.
  - If this is true, it implies that the item corresponding to the index at the top of the stack is eligible for a discount of value `v` (the current item's price).
  - We then pop the top index from the stack and subtract the discount `v` from the original price in the `ans` list at that index.
  - This process continues until the stack is empty or the current price `v` is no longer less than or equal to the prices of items indexed in the stack.
- After the inner `while` loop ends (meaning no more discounts can be applied or the stack is empty), the current index `i` is appended to the stack.
- This implies that this item is now waiting to potentially give a discount to a future item, or it will not receive a discount itself if no cheaper item comes after it.
- Once the iteration over all prices completes, the `ans` list will have been modified to contain the final prices after discount for each item.

By using the stack this way, each item is pushed and popped at most once, leading to a time complexity of O(n) where n is the number of items in `prices`, making the algorithm efficient and suitable for larger datasets.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Imagine we have the following prices: `[4, 2, 3, 7, 5]`. We want to calculate the final price for each item after applying the special discount according to our problem description and solution approach.

We initialize an empty stack `stk` and a list `ans` that is a copy of the original `prices` list, thus `ans` would initially be `[4, 2, 3, 7, 5]`.

1. Start at index `0` with price `4`. Stack is currently empty, so push `0`. Stack: `[0]`
2. Move to index `1` with price `2`. The top of the stack refers to price `4`. Since `2` is smaller than `4`, apply discount to `ans[0]`: `ans[0] = 4 - 2`. Pop `0` from the stack and push `1` on the stack. Stack: `[1]`, `ans` becomes `[2, 2, 3, 7, 5]`.
3. Proceed to index `2` with price `3`. The top index `1` has the price `2`, which is not greater than `3`, so no discount is applied. Push `2` onto the stack. Stack: `[1, 2]`
4. Next is index `3` with price `7`. `2` (from `ans[2]`) is smaller than `7`, so no discount is applied to the price at index `3`. Push `3` onto the stack. Stack: `[1, 2, 3]`
5. Lastly, index `4` with price `5`. Compare with top of the stack index `3` which is `7`. Since `5` is lower, apply discount to `ans[3]`: `ans[3] = 7 - 5`. Pop `3` from the stack. Next, compare with top of the stack index `2`, which has the price `3`. Price `3` is not higher than `5`, so we stop and push `4` onto the stack. Stack: `[1, 2, 4]`, `ans` becomes `[2, 2, 3, 2, 5]`.

Finally, there are no more items to consider, and the `ans` array holds the final discounted prices. The function would return the `ans` array, which is `[2, 2, 3, 2, 5]`. This is how much you would pay for each item after applying the discounts.

Python Solution

```python
from typing import List

class Solution:
    def finalPrices(self, prices: List[int]) -> List[int]:
        # Initialise a stack to keep track of the indices of prices
        stack = []

        # Make a copy of the prices list to store the final prices after discounts
        final_prices = prices[:]

        # Iterate over the prices with their corresponding indices
        for index, value in enumerate(prices):
            # Check for prices in the stack that are greater or equal to the current price
            while stack and prices[stack[-1]] >= value:
                # Apply the discount to the price at the top of the stack by subtracting
                # the current value from it, then pop it from the stack
                final_prices[stack.pop()] -= value

            # Push the current index onto the stack
            stack.append(index)

        # Return the modified prices list with applied discounts
        return final_prices

```

Java Solution

```java
class Solution {
    public int[] finalPrices(int[] prices) {
        // Create a stack to keep track of the indices of prices that haven't found a discount yet
        Deque<Integer> stack = new ArrayDeque<>();
        int n = prices.length;
        // Initialize an array to hold the final prices after applying the discounts
        int[] finalPrices = new int[n];

        // Iterate over the array of prices
        for (int i = 0; i < n; ++i) {
            // Store the original price as the final price for now
            finalPrices[i] = prices[i];

            // Check if the current price can be a discount for the price at the top of the stack
            while (!stack.isEmpty() && prices[stack.peek()] >= prices[i]) {
                // Apply discount to the top price and update it in the finalPrices array
                finalPrices[stack.pop()] -= prices[i];
            }

            // Push the current index onto the stack
            stack.push(i);
        }

        // Return the array of final prices after applying the discounts where applicable
        return finalPrices;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> finalPrices(vector<int>& prices) {
        // Create a stack to keep track of indices of prices
        stack<int> indexStack;

        // Initialize the answer vector with the original prices
        vector<int> discountedPrices = prices;

        // Loop through each price
        for (int i = 0; i < prices.size(); ++i) {
            // While stack is not empty and the current price is less than or equal to
            // the price at the top of the stack (indicates a discount is available)
            while (!indexStack.empty() && prices[indexStack.top()] >= prices[i]) {
                // Apply the discount to the price at the top index
                discountedPrices[indexStack.top()] -= prices[i];
                // Pop the index from the stack as the discount has been applied
                indexStack.pop();
            }
            // Push the current index onto the stack
            indexStack.push(i);
        }

        // Return the vector containing final prices after discounts
        return discountedPrices;
    }
};

```

Typescript Solution

```typescript
function finalPrices(prices: number[]): number[] {
    // Initialize the length of the prices array to avoid recomputation.
    const lengthOfPrices = prices.length;

    // Initialize the result array to store the final discounted prices.
    const discountedPrices = new Array(lengthOfPrices);

    // Initialize a stack to keep track of prices from the end to start.
    const priceStack: number[] = [];

    // Iterate over the prices array from the end to the beginning.
    for (let i = lengthOfPrices - 1; i >= 0; i--) {
        // Store the current price for readability.
        const currentPrice = prices[i];

        // Check prices in the stack; if any price is greater than the current price,
        // it cannot be a discount for the current price, so remove it.
        while (priceStack.length !== 0 && priceStack[priceStack.length - 1] > currentPrice) {
            priceStack.pop();
        }

        // Calculate the discounted price for the current item.
        // If the stack is empty, no discount is applied (hence the nullish coalescing operator ?? is used).
        discountedPrices[i] = currentPrice - (priceStack[priceStack.length - 1] ?? 0);

        // Add the current price to the stack for possible future discounts.
        priceStack.push(currentPrice);
    }

    // Return the array of discounted prices.
    return discountedPrices;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the `prices` list. This is because each element is pushed onto the stack at most once, and each element is popped from the stack at most once. The while loop will only iterate for each element if there is a matching discount, so even though there's a nested loop, it does not result in a quadratic time complexity.

The space complexity of the code is also `O(n)`. This is due to the stack `stk` that, in the worst case, can hold all the elements from the `prices` list if no discounts apply. Additionally, the `ans` list is a copy of the `prices` list and thus also takes `O(n)` space.