2898. Maximum Linear Stock Score

Medium Array Hash Table [Leetcode Link]

Problem Description

In this problem, we have an array `prices` that represents the price of a certain stock on different consecutive days, indexed starting from 1. The goal here is to select some of the stock prices in such a way that they form a linear selection. A selection is considered linear if the difference between the stock prices and their respective indices is constant for every pair of consecutive prices in the selection.

Formally, we say that a selection of indices is linear if the equation `prices[indexes[j]] - prices[indexes[j - 1]] == indexes[j] - indexes[j - 1]` holds for every consecutive pair of indices in the selection.

Your task is to maximize the score of such a selection, where the score is simply the sum of all stock prices in the selection.

The key to solving this problem lies in identifying that a linear selection of stock prices essentially forms an arithmetic sequence when considering the relationship between the prices and the indices. The challenge is to figure out which selections can produce the maximum sum while still adhering to the linearity condition.

Intuition

The solution to this problem hinges on recognizing that an arithmetic sequence has a constant difference between consecutive terms. Translating this to our problem, we're looking for those stock prices that maintain a constant difference between the price and its index for every pair in our selection.

In simpler terms, we want to find a subset of `prices` where `(price - index)` is the same for all of them. If we find the constant, that means we found an arithmetic subsequence. We can quickly test if an element belongs to an arithmetic subsequence by just subtracting its index from its price and checking if we have seen that result before.

To implement this intuition, we can use a hash table (in Python, a `Counter` object from the collections library) where the keys are the `(price - index)` value and the values are the sum of prices that correspond to those `(price - index)` values. The reason we sum the prices and not just count them is because we are interested in the sum of the prices, which corresponds to the "score" mentioned in the problem.

For each stock price, we calculate the value `(prices[i] - i)` and add the stock price to the sum in our hash table at that key. After we go through all stock prices, the maximum sum stored in the hash table gives us the maximum score of a linear selection. This way, we ensure we only examine the elements of `prices` once, giving us an efficient solution.

Solution Approach

The solution uses a hash table to keep track of the sum of all prices that share the same `(price - index)` value. The data structure used here is a `Counter` from the Python collections module, which is a subclass of the dictionary specifically designed to count hashable objects.

Let's break down the steps in the algorithm as follows:

1. Create a `Counter` object, here denoted as `cnt`, which will act as our hash table.
2. Iterate over the `prices` array using the index and value (`i, x` respectively). For each element:
   - Calculate the difference between the price `x` and the index `i` which gives you the necessary constant to check if a price belongs to a linear sequence.
   - Use this difference `x - i` as a key in our hash table. Accumulate the value `x` in `cnt[x - i]`. This means that for all prices which share the same `(price - index)` difference, their values will be added together in the hash table.
3. After the iteration, we will have a hash table where each key represents a possible `(price - index)` difference, and the corresponding value is the sum of all prices that share that difference.
4. The final step is to find the maximum value in the hash table `cnt`. The `max(cnt.values())` operation will give us the highest sum of prices, which directly equates to the maximum possible score we can achieve with a linear selection.

The Reference Solution Approach succinctly states the transformation of the original equation in the problem to a simpler form that can be solved using a hash table. By keeping the sum of all prices that have the same `(price - index)` difference, we can easily retrieve the maximum score of a linear selection in `O(n)` time complexity, where `n` is the length of the `prices` array.

This approach significantly simplifies the problem as it avoids the need for nested loops or more complex data structures, effectively turning it into a single-pass solution with a final aggregate operation to find the maximum.

Example Walkthrough

Let's consider an example where the `prices` array is `[3, 8, 1, 7, 10, 15]`.

Now, we will walk through the solution approach step by step:

1. **Create a Counter object**: We start by creating a `Counter` called `cnt` that will hold the sums.

2. **Iterate over the `prices` array**:

   - **Index `i = 1`, Price `x = 3`**:
     - Calculate the difference: `x - i = 3 - 1 = 2`.
     - Update the hash table: `cnt[2] = 3`.
   - **Index `i = 2`, Price `x = 8`**:
     - Difference: `x - i = 8 - 2 = 6`.
     - Update: `cnt[6] = 8`.
   - **Index `i = 3`, Price `x = 1`**:
     - Difference: `x - i = 1 - 3 = -2`.
     - Update: `cnt[-2] = 1`.
   - **Index `i = 4`, Price `x = 7`**:
     - Difference: `x - i = 7 - 4 = 3`.
     - Update: `cnt[3] = 7`.
   - **Index `i = 5`, Price `x = 10`**:
     - Difference: `x - i = 10 - 5 = 5`.
     - Update: `cnt[5] = 10`.
   - **Index `i = 6`, Price `x = 15`**:
     - Difference: `x - i = 15 - 6 = 9`.
     - Update: `cnt[9] = 15`.

3. **Hash Table (`cnt`) After the Iteration**:

   - After iterating over all prices, our `cnt` will look like this:

   ```
   Counter({
     2: 3,
     6: 8,
     -2: 1,
     3: 7,
     5: 10,
     9: 15
   })
   ```

   Each key is a `(price - index)` difference, and each value is the sum of the prices that share that difference (in this case, just individual prices, as there are no repeat `(price - index)` differences).

4. **Find the Maximum Value in the Hash Table**:

   - To find the maximum score, we look for the maximum value in `cnt`: `max(cnt.values())`, which is `max([3, 8, 1, 7, 10, 15])`.
   - The maximum score here is `15`.

In this example, the maximum score corresponds to the price that has a unique `(price - index)` difference. In cases where there would be multiple prices sharing the same difference, their summed value would potentially be the maximum score.

The process demonstrates the power of the hashing technique to solve the problem efficiently. This way, we find the max score possible from a linear selection of stock prices in a single pass over the array.

Python Solution

```python
from collections import Counter

class Solution:
    def maxScore(self, prices: List[int]) -> int:
        # Initialize a counter to keep track of the score
        score_counter = Counter()

        # Iterate through the list of prices with their respective indices
        for index, price in enumerate(prices):
            # Calculate the 'key' for this price by subtracting the index
            key = price - index
            # Add the price to the counter for the calculated key
            score_counter[key] += price

        # Return the maximum score found in the counter
        # The values in the counter represent the cumulative score for each key
        return max(score_counter.values())

```

Java Solution

```java
class Solution {
    public long maxScore(int[] prices) {
        // A HashMap to keep track of the sums of price contributions
        Map<Integer, Long> contributionCounts = new HashMap<>();

        // Loop over the prices to calculate each contribution
        for (int i = 0; i < prices.length; ++i) {
            // We calculate each price's unique contribution key as the price minus the index
            int contributionKey = prices[i] - i;
            // We sum the actual price for each contribution key in the map
            contributionCounts.merge(contributionKey, (long) prices[i], Long::sum);
        }

        // Initialize the maximum score to zero
        long maxScore = 0;
        // Iterate over the values in the contributions map
        for (long contributionSum : contributionCounts.values()) {
            // Update maxScore to be the maximum of the current maxScore and the current contribution sum
            maxScore = Math.max(maxScore, contributionSum);
        }

        // Return the maximum score found
        return maxScore;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm> // for std::max

class Solution {
public:
    // Method to compute the maximum score based on the given problem statement
    long long maxScore(vector<int>& prices) {
        // Using a hash map to store the computed value difference and sum of prices with the same difference
        unordered_map<int, long long> countMap;

        // Iterate through the prices array
        for (int index = 0; index < prices.size(); ++index) {
            // The key here is the value difference between 'prices[index]' and 'index'
            // Simultaneously, aggregate the sum of prices that have the same value difference
            countMap[prices[index] - index] += prices[index];
        }

        // Variable to store the maximum score
        long long maxScore = 0;

        // Iterate through the hash map to find the maximum aggregated sum of prices
        for (auto& keyValue : countMap) {
            // The first element of the pair (keyValue.first) is not used here
            long long currentValue = keyValue.second;
            // Update the 'maxScore' with the maximum value found so far
            maxScore = std::max(maxScore, currentValue);
        }

        // Return the maximum score computed
        return maxScore;
    }
};

```

Typescript Solution

```typescript
// This function calculates the maximum score based on a specific scoring rule.
// The score for each number is determined by adding the number's value to the number of times
// it appears at an index equal to its value minus its index.
// For example, the number at index i contributes its value to the total score if it is equal to i.
function maxScore(prices: number[]): number {
    // Initialize the map to keep track of the score computed for each unique (price - index) pair
    const scoreCount: Map<number, number> = new Map();

    // Loop through each price in the array
    for (let i = 0; i < prices.length; ++i) {
        // Calculate the key for the map, which represents the unique price realization score
        const scoreKey: number = prices[i] - i;

        // Update the map with the new score, incrementing the existing score if the key already exists
        // If the key doesn't exist, it initializes the score with the current price value
        scoreCount.set(scoreKey, (scoreCount.get(scoreKey) || 0) + prices[i]);
    }

    // Find and return the largest score from the map
    return Math.max(...scoreCount.values());
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `prices` array. This is because the code iterates through each element of the `prices` array only once within a single for loop to build the counter, which in aggregate results in linear time complexity relative to the input size.

The space complexity of the code is also `O(n)` due to the use of a counter to store the frequency of each calculated value `(x - i)`. In the worst case, where all `(x - i)` values are unique, the counter could contain `n` key-value pairs corresponding to the number of elements in the `prices` array. Hence, the space complexity is linear as well.