2819. Minimum Relative Loss After Buying Chocolates

Hard

Leetcode Link

Problem Description

In this problem, you are given an array `prices` that represents the cost of chocolates and a 2D array `queries`, each containing two integers: `k_i` and `m_i`. Alice and Bob are purchasing chocolates together, and they have a special arrangement for sharing the cost. For any given chocolate:

- If the chocolate costs less than or equal to `k_i`, then Bob will pay for it entirely.
- If the chocolate costs more than `k_i`, Bob will pay `k_i`, and Alice will cover the remaining cost.

Bob can choose exactly `m_i` chocolates, and he wants to minimize his relative loss. The relative loss is defined as the difference between the total amount Bob paid and the total amount Alice paid, which he wants to minimize.

Your task is to determine the minimum relative loss Bob can achieve for each query `queries[i]` and return an array of integers representing these minimum losses.

Intuition

Given that Bob wants to minimize his relative loss (Bob's payment minus Alice's payment), we have to approach this problem by trying to minimize the expenses that go beyond Bob's contribution limit (`k_i`). To achieve this, we can choose more chocolates with prices within or equal to Bob's limit and try to minimize picking chocolates that require an additional payment from Alice.

To find the optimum distribution of chocolates that Bob should pay for completely and those that Alice contributes to, we can:

1. Sort the `prices` array in ascending order. This allows us to quickly identify which chocolates can be paid for by Bob and which will require a contribution from Alice.
2. We can use binary search to find the breakpoint where Bob should stop paying for chocolates entirely and start sharing the cost with Alice.

The function `f(k, m)` performs this binary search by finding the right number of chocolates Bob can fully pay for without incurring too much additional cost from Alice. Given that `m` chocolates in total need to be purchased, the function calculates the split (using a binary search approach) between the chocolates paid for entirely by Bob and those partially paid for by him (with the rest paid by Alice).

This split helps minimize Bob's relative loss, as it finds the best point where paying an equal or lesser price fully (for as many chocolates as possible) and sharing the cost of the more expensive chocolates leads to the smallest difference between Bob's payment and Alice's payment.

Finally, the `minimumRelativeLosses` method calculates the minimum loss for each query by considering the split point `l`, computing the total cost paid by Bob and Alice, and then determining the relative loss (Bob's payment minus Alice's payment). The result for each query is then appended to the `ans` list, which is returned as the solution to the problem.

Solution Approach

The solution uses a binary search algorithm combined with a prefix sum technique to efficiently compute the required values for minimizing Alice's contribution and thus Bob's relative loss.

Here's a step-by-step breakdown:

1. First, we sort the `prices` array. This is crucial as it allows us to apply binary search later.

2. We compute the prefix sums of the array using `accumulate(prices, initial=0)`. Prefix sums are efficient for calculating the sum of a subarray in constant time O(1). Having a prefix sum array `s` means `s[i]` represents the sum of the prices array from `prices[0]` to `prices[i-1]`.

3. The code then defines a function `f(k: int, m: int) -> int` that uses binary search to find the optimal split point `l`:

   - Initialize two pointers, `l` and `r`. `l` starts at 0, and `r` is the minimum of `m` and the index of the rightmost chocolate whose price does not exceed `k`. This index is found using `bisect_right(prices, k)` which returns the insertion point to the right of `k` in the sorted `prices` list.

   - While `l < r`, we find the mid-point and calculate how many chocolates to the right of `mid` Bob would need to share the cost with Alice (`right = m - mid`).

   - If the price of the chocolate at midpoint is less than `2 * k - prices[n - right]`, we move the `l` pointer to `mid + 1`. This condition checks if the price at the mid is preferred over the price at the symmetric position in the second half (towards the array's end).

   - If the condition is not met, we move the `r` pointer to `mid`. This helps us to narrow down the split point where Bob would switch from paying in full to sharing the cost.

4. For each `k, m` pair in `queries`, we get the split point `l` using `f(k, m)`. Calculate Bob's total payment as the sum of the chocolate prices up to `l` (`s[l]`) plus doubled `k` for each of the chocolates from `l` to `m` (`2 * k * r`). Calculate Alice's total payment as the sum of the chocolate prices not covered by Bob (`s[n] - s[n - r]`).

5. Compute the loss as `loss = s[l] + 2 * k * r - (s[n] - s[n - r])`, which represents Bob's total payment minus Alice's total payment.

6. Append each computed loss to the `ans` array, which stores Bob's minimum relative loss for each query.

By combining binary search to efficiently find the optimal split of chocolate prices Bob should pay fully, and prefix sums to quickly compute the sums for Bob's and Alice's payments, the entire algorithm remains efficient and effective at minimizing Bob's relative loss for all the queries.

Example Walkthrough

Let us consider a small example to illustrate the solution approach using the following `prices` array and `queries`.

Suppose the `prices` array is `[1, 4, 5, 7, 8]` representing the cost of chocolates in ascending order. And there is one query in the `queries` array: `[4, 3]` which means that Bob's payment limit `k_i` is 4 and he can choose `m_i` = 3 chocolates.

Here are the steps we would take to find the minimum relative loss for this query:

1. Since our `prices` array is already sorted, we don't need to sort it again.

2. We calculate the prefix sums of the `prices` array. The prefix sums array `s` would be `[0, 1, 5, 10, 17, 25]`.

3. Next, using the `f(k, m)` function, we perform a binary search to find the optimal number of chocolates that Bob will pay for entirely, without sharing the cost with Alice.

   Given `k` = 4 and `m` = 3, we initialize `l` = 0 and `r` = min(3, 2) since there are 2 chocolates at indices `0` and `1` that are less than or equal to `k`.

   So, `r` starts at 2. We examine the midpoint `mid` between `l` and `r`, which is 1. Calculating the number of chocolates to the right of `mid` that Bob has to share the cost for: `right = m - mid` = 3 - 1 = 2.

   The condition in our binary search checks if `prices[mid] < 2 * k - prices[n - right]`, which is `prices[1] < 2 * 4 - prices[5 - 2]`, meaning `4 < 8 - 5`. Since this is false, we don't adjust `l`; instead, we make `r` equal to `mid`.

   With `l` = 0 and `r` = 1, we find that `mid` = 0, and repeating the above condition, we get `prices[0] < 2 * 4 - prices[5 - 2]`, which means `1 < 8 - 5`. This condition is true, so we update `l` to `mid + 1`, which makes `l` = 1.

   As `l` cannot be less than `r`, our binary search concludes that `l` = 1 is the optimal split point.

4. For the query `[4, 3]`, we have `l` = 1 and `r` = `m` - `l` = 2. Bob's total payment is the sum of prefix sums up to `l` plus `k` times `r` or `s[1] + 2 * k * r`, which gives us `1 + 2 * 4 * 2 = 17`. Alice's total payment is `s[5] - s[5 - r] = 25 - 10 = 15`.

5. The relative loss for Bob is thus `17 - 15 = 2`. We append `2` to our answer array, which means for the query `(4, 3)`, the minimum relative loss Bob can achieve is `2`.

By going through the above steps, we conclude that by choosing the first one chocolate costing `1` (within his limit) and contributing to the costs of two more expensive chocolates (with Alice covering the remaining), Bob can minimize his relative loss to `2`.

Python Solution

```python
from bisect import bisect_right
from typing import List
from itertools import accumulate

class Solution:
    def minimumRelativeLosses(self, prices: List[int], queries: List[List[int]]) -> List[int]:
        # Helper function to find the split point where we choose which paintings to buy
        def find_split(k: int, m: int) -> int:
            left, right = 0, min(m, bisect_right(prices, k))
            # Binary search to find the optimal split between buying cheaper paintings
            # and the given value painting
            while left < right:
                mid = (left + right) >> 1
                remaining = m - mid
                # Compare the mid-th price with the price that makes the total loss minimum
                if prices[mid] < 2 * k - prices[len(prices) - remaining]:
                    left = mid + 1
                else:
                    right = mid
            return left

        prices.sort()  # Sort the list of prices
        accumulated_prices = list(accumulate(prices, initial=0))  # Get the accumulated price sum
        results = []  # This will store the results of our queries
        num_prices = len(prices)  # Get the number of available paintings

        # Process each query
        for value, quantity in queries:
            split_point = find_split(value, quantity)  # Find the split point based on value and quantity
            quantity_left = split_point
            quantity_right = quantity - split_point
            # Calculate the total loss
            loss = accumulated_prices[split_point] + 2 * value * quantity_right - (accumulated_prices[num_prices] - accumulated_price
            results.append(loss)  # Append the calculated loss to results

        return results

# Example of how to use the class
# solution = Solution()
# print(solution.minimumRelativeLosses([10, 15, 20], [[15, 2], [20, 3]]))
```

Java Solution

```java
class Solution {
    private int itemCount; // The number of items in the prices array
    private int[] itemPrices; // Array of item prices

    public long[] minimumRelativeLosses(int[] prices, int[][] queries) {
        itemCount = prices.length;
        Arrays.sort(prices); // Sort prices in ascending order
        this.itemPrices = prices; // Assign sorted prices to instance variable
        // Initialize the prefix sum array with an additional zero at the start
        long[] prefixSum = new long[itemCount + 1];
        for (int i = 0; i < itemCount; ++i) {
            prefixSum[i + 1] = prefixSum[i] + prices[i];
        }
        int q = queries.length; // The number of queries
        long[] answers = new long[q]; // Array to hold the answers to the queries
        for (int i = 0; i < q; ++i) {
            // Extract k and m from each query
            int k = queries[i][0], m = queries[i][1];
            // Find the optimal left index for dividing the array into two parts
            int left = optimalLeftIndex(k, m);
            int right = m - left; // Calculate the right part size
            // Calculate the minimum relative loss for the query
            answers[i] = prefixSum[left] + (2L * k * right) - (prefixSum[itemCount] - prefixSum[itemCount - right]);
        }
        return answers; // Return the array of answers
    }

    // Helper method to find the optimal left index
    private int optimalLeftIndex(int k, int m) {
        int left = 0, right = Arrays.binarySearch(itemPrices, k);
        if (right < 0) {
            right = -(right + 1);
        }
        right = Math.min(m, right);
        while (left < right) {
            int mid = (left + right) / 2;
            int remainingRight = m - mid;
            // Determine if we should move the partition to the right or left
            if (itemPrices[mid] < 2L * k - itemPrices[itemCount - remainingRight]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left; // Return the optimal left index
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::sort and std::upper_bound

class Solution {
public:
    // Function to calculate the minimum relative losses for each query.
    vector<long long> minimumRelativeLosses(vector<int>& prices, vector<vector<int>>& queries) {
        int n = prices.size();
        // Sort the prices in ascending order to apply binary search later.
        sort(prices.begin(), prices.end());

        // Build a prefix sum array to quickly calculate sums of segments.
        vector<long long> prefixSum(n + 1, 0);
        for (int i = 1; i <= n; ++i) {
            prefixSum[i] = prefixSum[i - 1] + prices[i - 1];
        }

        // Binary search function to find the optimal split point for the given constraints.
        auto calculateSplitPoint = [&](int target, int maxSize) {
            int left = 0, right = upper_bound(prices.begin(), prices.end(), target) - prices.begin();
            right = min(right, maxSize);
            while (left < right) {
                int mid = (left + right) >> 1;
                int numRight = maxSize - mid;
                if (prices[mid] < 2LL * target - prices[n - numRight]) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            }
            return left;
        };

        // Answer vector to hold the result for each query.
        vector<long long> ans;
        for (auto& query : queries) {
            int target = query[0], maxSize = query[1];
            // Find the split point where we get the minimum relative loss.
            int splitPoint = calculateSplitPoint(target, maxSize);
            int numRight = maxSize - splitPoint;
            // Calculate the loss based on the split point determined using binary search.
            long long loss = prefixSum[splitPoint] + 2LL * target * numRight - (prefixSum[n] - prefixSum[n - numRight]);
            ans.push_back(loss);
        }
        return ans;
    }
};
```

Typescript Solution

```typescript
function minimumRelativeLosses(prices: number[], queries: number[][]): number[] {
    // The length of the prices array
    const pricesLength: number = prices.length;
    // Sort the prices array in ascending order.
    prices.sort((a, b) => a - b);

    // Create a prefix sum array to store cumulative sum of sorted prices.
    const prefixSums: number[] = Array(pricesLength).fill(0);
    for (let i = 0; i < pricesLength; ++i) {
        prefixSums[i + 1] = prefixSums[i] + prices[i];
    }

    // Binary search to find the index of the first price greater than x.
    const searchIndex = (x: number): number => {
        let left = 0;
        let right = pricesLength;
        while (left < right) {
            const mid = Math.floor((left + right) / 2);
            if (prices[mid] > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Function to find number of elements to include from start of sorted prices
    // to minimize the loss when k items are priced at 'k' and remaining items are
    // at their respective prices from sorted prices.
    const findMinLossSplit = (k: number, m: number): number => {
        let left = 0;
        let right = Math.min(searchIndex(k), m);
        while (left < right) {
            const mid = Math.floor((left + right) / 2);
            const rightCount = m - mid;
            if (prices[mid] < 2 * k - prices[pricesLength - rightCount]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    };

    // Main logic where for each query, it calculates the minimum relative loss.
    const answers: number[] = [];
    for (const [outputPrice, outputCount] of queries) {
        const leftCount = findMinLossSplit(outputPrice, outputCount);
        const rightCount = outputCount - leftCount;
        answers.push(
            prefixSums[leftCount] +
            2 * outputPrice * rightCount -
            (prefixSums[pricesLength] - prefixSums[pricesLength - rightCount])
        );
    }

    // Return resulting minimum losses for all queries.
    return answers;
}
```

Time and Space Complexity

Time Complexity

The `minimumRelativeLosses` function consists of a main for-loop that processes each query, a sorted operation on the prices list, and a helper function (`f`) that uses binary search.

1. Sorting the `prices` list has a time complexity of `O(n log n)`, where `n` is the length of the prices.

2. The `accumulate` function, which calculates the prefix sum array `s`, has a time complexity of `O(n)` because it processes each element in the `prices` array exactly once.

3. The function `f` uses a binary search to find the optimal split point that minimizes relative losses. The binary search within the `f` function runs in `O(log min(m, n))`, where `m` is the number of elements in one query and `n` is the length of the prices. This is due to the binary search being potentially limited by `m`, the number of elements that the query wants to examine.

4. The main for-loop iterates over each query in `queries`. Assuming `q` is the number of queries, the for-loop runs `q` times. Inside this loop, the `f` function is called, which has the `O(log min(m, n))` complexity.

So the overall time complexity of the for-loop including the call to the `f` function is `O(q * log min(m, n))`.

Combining these complexities, we get `O(n log n) + O(n) + O(q * log min(m, n))`. Since `q * log min(m, n)` could potentially be larger than `n log n` and `n`, especially when `q` is large, we can consider `O(q * log min(m, n))` to be the dominating term in typical scenarios.

Final Time Complexity: `O(n log n) + O(n) + O(q * log min(m, n))`.

Space Complexity

1. The `prices` list is sorted in-place, so no additional space is required for that.

2. The `accumulate` function creates a prefix sum array `s` with `n + 1` elements, giving a space complexity of `O(n)`.

3. The `ans` list will hold `q` elements (one for each query), which gives a space complexity of `O(q)`.

The only additional space used is the `s` and `ans` lists, since binary search does not require additional space beyond the few variables for pointers and indices.

Final Space Complexity: `O(n) + O(q)` or simply `O(n + q)` if we assume `q` can be as significant as `n`.