2819. Minimum Relative Loss After Buying Chocolates

Hard

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.

In this problem, you are given an array prices that represents the cost of chocolates and a 2D array queries, each containing two

Leetcode Link

- 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

with Alice.

will require a contribution from Alice.

Problem Description

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

2. We can use binary search to find the breakpoint where Bob should stop paying for chocolates entirely and start sharing the cost

prices within or equal to Bob's limit and try to minimize picking chocolates that require an additional payment from 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 1, 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.

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.

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

3. The code then defines a function f(k: int, m: int) -> int that uses binary search to find the optimal split point 1:

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

o 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

Solution Approach

Here's a step-by-step breakdown:

prices[0] to prices[i-1].

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 1 < 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

loss for all the queries.

switch from paying in full to sharing the cost.

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

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 1 (s[1]) plus doubled k for each of the chocolates from 1 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.

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

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.

mid that Bob has to share the cost for: right = m - mid = 3 - 1 = 2.

entirely, without sharing the cost with Alice.

Example Walkthrough

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

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

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

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

So, r starts at 2. We examine the midpoint mid between 1 and r, which is 1. Calculating the number of chocolates to the right of

prices [5 - 2], meaning 4 < 8 - 5. Since this is false, we don't adjust 1; instead, we make r equal to mid.

Python Solution

class Solution:

8

13

14

15

16

17

18

19

20

21

22

23

30

31

32

33

34

35

36

37

1 from bisect import bisect_right

from itertools import accumulate

def find_split(k: int, m: int) -> int:

mid = (left + right) >> 1

remaining = m - mid

right = mid

quantity_left = split_point

Calculate the total loss

quantity_right = quantity - split_point

40 # print(solution.minimumRelativeLosses([10, 15, 20], [[15, 2], [20, 3]]))

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

else:

return left

return results

38 # Example of how to use the class

39 # solution = Solution()

left = mid + 1

prices.sort() # Sort the list of prices

from typing import List

relative loss Bob can achieve is 2.

equal to k.

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

+ 2 * k * r, which gives us 1 + 2 * 4 * 2 = 17. Alice's total payment is s[5] - s[5 - r] = 25 - 10 = 15.

def minimumRelativeLosses(self, prices: List[int], queries: List[List[int]]) -> List[int]:

Compare the mid-th price with the price that makes the total loss minimum

accumulated_prices = list(accumulate(prices, initial=0)) # Get the accumulated price sum

Helper function to find the split point where we choose which paintings to buy

if prices[mid] < 2 * k - prices[len(prices) - remaining]:</pre>

results.append(loss) # Append the calculated loss to results

means 1 < 8 - 5. This condition is true, so we update 1 to mid + 1, which makes 1 = 1. As 1 cannot be less than r, our binary search concludes that 1 = 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 1 plus k times r or s[1]

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

The condition in our binary search checks if prices [mid] < 2 * k - prices [n - right], which is prices [1] < 2 * 4 - prices [n - right]

- 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.
- left, right = 0, min(m, bisect_right(prices, k)) 9 # Binary search to find the optimal split between buying cheaper paintings 10 # and the given value painting 11 while left < right:</pre> 12

24 results = [] # This will store the results of our queries 25 num_prices = len(prices) # Get the number of available paintings 26 27 # Process each query 28 for value, quantity in queries:

loss = accumulated_prices[split_point] + $2 * value * quantity_right - (accumulated_prices[num_prices] - accumulated_price$

split_point = find_split(value, quantity) # Find the split point based on value and quantity

```
private int itemCount; // The number of items in the prices array
      private int[] itemPrices; // Array of item prices
3
      public long[] minimumRelativeLosses(int[] prices, int[][] queries) {
          itemCount = prices.length;
6
```

C++ Solution

public:

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

48

4

5

6

54

55

56

57

58

59

61

60 }

scenarios.

);

Time Complexity

return answers;

47 };

#include <vector>

class Solution {

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

sort(prices.begin(), prices.end());

for (int i = 1; $i \le n$; ++i) {

while (left < right) {</pre>

} else {

return left;

vector<long long> ans;

for (auto& query : queries) {

ans.push_back(loss);

// The length of the prices array

prices.sort((a, b) => a - b);

const pricesLength: number = prices.length;

// Sort the prices array in ascending order.

return ans;

};

vector<long long> prefixSum(n + 1, 0);

right = min(right, maxSize);

left = mid + 1;

right = mid;

int mid = (left + right) >> 1;

int numRight = maxSize - mid;

// Answer vector to hold the result for each query.

int target = query[0], maxSize = query[1];

int numRight = maxSize - splitPoint;

int n = prices.size();

// Function to calculate the minimum relative losses for each query.

prefixSum[i] = prefixSum[i - 1] + prices[i - 1];

auto calculateSplitPoint = [&](int target, int maxSize) {

// Sort the prices in ascending order to apply binary search later.

// Build a prefix sum array to quickly calculate sums of segments.

if (prices[mid] < 2LL * target - prices[n - numRight]) {</pre>

// Find the split point where we get the minimum relative loss.

// 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]);

int splitPoint = calculateSplitPoint(target, maxSize);

// Create a prefix sum array to store cumulative sum of sorted prices.

vector<long long> minimumRelativeLosses(vector<int>& prices, vector<vector<int>>& queries) {

// Binary search function to find the optimal split point for the given constraints.

int left = 0, right = upper_bound(prices.begin(), prices.end(), target) - prices.begin();

8

9

Java Solution

1 class Solution {

```
long[] prefixSum = new long[itemCount + 1];
10
            for (int i = 0; i < itemCount; ++i) {</pre>
11
12
                prefixSum[i + 1] = prefixSum[i] + prices[i];
13
14
            int q = queries.length; // The number of queries
15
            long[] answers = new long[q]; // Array to hold the answers to the queries
16
            for (int i = 0; i < q; ++i) {
                // Extract k and m from each query
17
18
                int k = queries[i][0], m = queries[i][1];
19
                // Find the optimal left index for dividing the array into two parts
                int left = optimalLeftIndex(k, m);
20
21
                int right = m - left; // Calculate the right part size
22
                // Calculate the minimum relative loss for the query
                answers[i] = prefixSum[left] + (2L * k * right) - (prefixSum[itemCount] - prefixSum[itemCount - right]);
24
25
            return answers; // Return the array of answers
26
27
28
        // Helper method to find the optimal left index
        private int optimalLeftIndex(int k, int m) {
29
            int left = 0, right = Arrays.binarySearch(itemPrices, k);
30
31
            if (right < 0) {
32
                right = -(right + 1);
33
34
            right = Math.min(m, right);
            while (left < right) {</pre>
35
                int mid = (left + right) / 2;
36
37
                int remainingRight = m - mid;
38
                // Determine if we should move the partition to the right or left
39
                if (itemPrices[mid] < 2L * k - itemPrices[itemCount - remainingRight]) {</pre>
                    left = mid + 1;
40
                } else {
41
                    right = mid;
42
43
44
45
            return left; // Return the optimal left index
46
47
48
```

Typescript Solution function minimumRelativeLosses(prices: number[], queries: number[][]): number[] {

```
const prefixSums: number[] = Array(pricesLength).fill(0);
 8
        for (let i = 0; i < pricesLength; ++i) {</pre>
 9
10
            prefixSums[i + 1] = prefixSums[i] + prices[i];
11
12
       // Binary search to find the index of the first price greater than x.
13
14
        const searchIndex = (x: number): number => {
15
            let left = 0;
16
            let right = pricesLength;
            while (left < right) {</pre>
17
18
                const mid = Math.floor((left + right) / 2);
                if (prices[mid] > x) {
19
                    right = mid;
20
21
                } else {
22
                    left = mid + 1;
23
24
25
            return left;
26
        };
27
28
       // 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
29
30
       // at their respective prices from sorted prices.
31
        const findMinLossSplit = (k: number, m: number): number => {
32
            let left = 0;
            let right = Math.min(searchIndex(k), m);
33
34
            while (left < right) {</pre>
35
                const mid = Math.floor((left + right) / 2);
                const rightCount = m - mid;
37
                if (prices[mid] < 2 * k - prices[pricesLength - rightCount]) {</pre>
                    left = mid + 1;
38
                } else {
39
40
                    right = mid;
41
42
43
            return left;
        };
44
45
46
       // Main logic where for each query, it calculates the minimum relative loss.
47
        const answers: number[] = [];
        for (const [outputPrice, outputCount] of queries) {
48
49
            const leftCount = findMinLossSplit(outputPrice, outputCount);
            const rightCount = outputCount - leftCount;
50
            answers.push(
51
52
                prefixSums[leftCount] +
53
                2 * outputPrice * rightCount -
```

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

Time and Space Complexity

a helper function (f) that uses binary search.

element in the prices array exactly once.

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.

(prefixSums[pricesLength] - prefixSums[pricesLength - rightCount])

// Return resulting minimum losses for all queries.

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 $0(n \log n) + 0(n) + 0(q * \log \min(m, n))$. 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

Final Time Complexity: $0(n \log n) + 0(n) + 0(q * \log min(m, n))$.

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

The minimumRelativeLosses function consists of a main for-loop that processes each query, a sorted operation on the prices list, and

3. The function f uses a binary search to find the optimal split point that minimizes relative losses. The binary search within the f

to the binary search being potentially limited by m, the number of elements that the query wants to examine.

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

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.

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

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