1058. Minimize Rounding Error to Meet Target String Medium Greedy Array Math

Leetcode Link

Problem Description

defined as the sum of the absolute differences between the original prices and their rounded values. To clarify with an example, if we have prices prices = [0.70, 2.80, 4.90] and a target of 8, our goal is to round these prices to

whole numbers such as 1, 3, and 4, respectively, so that their sum is 8. The rounding error in this case is |1-0.70| + |3-2.80| + |4-4.90| = 0.30 + 0.20 + 0.10 = 0.60.

Yet, there might be cases where it is impossible to round the prices to reach the exact target sum. For instance, if we had the target 10 in the previous example, there's no way to round 0.70, 2.80, 4.90 to sum up to 10.

Intuition

To arrive at the solution, a good starting point is to consider the minimum and maximum sums we can get by rounding all prices

(minimum sum <= target <= maximum sum), we return "-1" as it is impossible to achieve the target by rounding.

down (using floor) or up (using ceil), respectively. We calculate the total sum of the floored values and count the items that have a fraction (i.e., numbers that are not already integers). If the target is outside the range of the minimum sum and the maximum sum

Given that the target is within reach, we want to minimize the error by carefully choosing which numbers to round up and which to round down. Since this rounding must add up to the target, we need to round up a specific number of prices, corresponding to the difference between the target and the minimum sum we calculated earlier. To minimize the error, we should prioritize rounding up numbers with the smallest fractional parts, as doing so will lead to the smallest possible error.

The solution approach is as follows: 1. Calculate the total sum of floor values of all the prices. 2. Calculate how much we need to add to the floored sum to reach the target. 3. Sort the fractional parts of each number in descending order. By doing this, we can ensure that when rounding up, we always

4. Compute the error contributed by the numbers we need to round up (first d numbers after sorting) and the error contributed by

the numbers we leave as floor values. 5. Return the sum of these errors as a formatted string with three decimal places.

add the smallest necessary amount to our total sum.

- Solution Approach
- The solution uses simple data manipulation techniques and sorting algorithm to achieve the desired minimum error value. Following is the detailed explanation of how the Python code implements the solution:
- 1. Initial Summation and Difference Collection The first step in the code iterates through the given prices and calculates the sum of their floor values. This is done by iterating each price p in the prices array, converting it to a float, and then adding its floor

value to mi. Simultaneously, the code checks if the given price has a non-zero fractional part (i.e., p is not an integer) and, if so,

appends the fractional part to the arr list by using d := p - int(p). This approach employs basic looping and arithmetic operations, and utilizes a list to collect all fractional parts for later

code determines it's not feasible to reach the target and immediately returns "-1".

O(n log n) time complexity for Python's Timsort algorithm, where n is the number of prices given.

parts first when we round up, which minimizes the error.

f'{ans:.3f}', which completes our rounding error calculation.

aggregation functions like sum() makes the code concise and efficient.

 \circ For 2.50, its floor value is 2 and the fractional part is 2.50 - 2 = 0.50.

 \circ We have one value to round up, which is 0.40, so error from it is 1 - 0.40 = 0.60.

processing.

2. Checking the Feasibility After initial summation and difference collection, we verify whether it's possible to hit the target by rounding. This is checked by the condition if not mi <= target <= mi + len(arr): return "-1". Here, mi denotes the sum of

the floor values, and mi + len(arr) denotes the sum if we were to round all prices up. If the target is not within this range, the

3. Sorting and Error Calculation If the target is achievable, then the arr list, which contains all the differences between each price

and its floor value, is sorted in descending order using arr.sort(reverse=True). This allows us to select the smallest fractional

4. Computing the Total Rounding Error The total rounding error is computed by taking into account the numbers that we have to round up to reach the target. The difference between the target and initial sum mi gives us d, the count of prices we must round up. The sum of rounding errors for prices rounded up is sum(arr[:d]), and for those rounded down is sum(arr[d:]). We subtract the first from d and add the second to find the total error. 5. Formatting the Result Finally, the code computes the answer and formats it to 3 decimal places using the formatted string

This solution is efficient as it makes use of the properties of numbers and their rounding behaviors, along with sorting, which runs in

The primary data structure utilized here is the list, for collecting the fractional parts and later sorting them. The use of list slicing and

The given solution elegantly combines simple mathematical observation with a well-known sorting algorithm to solve what initially appears to be a complex problem. This implementation underlines the effectiveness of combining fundamental programming constructs and algorithms to solve practical computational problems.

Considering the prices array: \circ For 1.50, its floor value is 1 and the fractional part is 1.50 - 1 = 0.50.

The minimum sum mi is 6 and if we were to round all prices up, the maximum sum would be 6 + 3 = 9. The target 7 is within this

We sort arr in descending order, resulting in [0.50, 0.50, 0.40]. It remains intact as all elements are in the correct order.

We need to increase our sum from 6 to 7, hence d = 7 - 6 = 1. This means we need to round up one price. To minimize error,

 \circ For 3.40, its floor value is 3 and the fractional part is 3.40 - 3 = 0.40. The sum of floor values, mi, is 1 + 2 + 3 = 6, and the fractional parts arr are [0.50, 0.50, 0.40].

Let's step through the solution approach using the example where prices = [1.50, 2.50, 3.40] and the target sum is 7.

we round up the price with the smallest fractional part, which is 0.40. The rounding error for rounding up:

Example Walkthrough

2. Checking the Feasibility

3. Sorting and Error Calculation

4. Computing the Total Rounding Error

1. Initial Summation and Difference Collection

range (6 <= 7 <= 9), so proceeding is possible.

The rounding error for numbers we don't round up:

Therefore, the total error is 0.60 + 1.00 = 1.60.

to meet our target sum with the minimal rounding difference.

Initialize an array to hold the fractional parts of the prices

if not floor_sum <= target <= floor_sum + len(fractional_parts):</pre>

and then subtracting this from the total sum of fractional parts

return "-1" # If it is impossible, return "-1"

public String minimizeError(String[] prices, int target) {

// Initialize an array list to store the fractional parts.

// Add the floor value of the price to the floor sum.

// If there is a fractional part, add it to the fractions list.

if (target < floorSum || target > floorSum + fractions.size()) {

// Initialize the floor sum of prices.

List<Double> fractions = new ArrayList<>();

// Convert the string to a double value.

double fraction = price - (int) price;

fractions.add(fraction);

// Check if the target is unreachable.

for (const auto& priceStr : prices) {

fractions.push_back(fraction);

// If the target is less than the sum of floor values or

// Calculating the number of elements to take the ceiling

for (int i = ceilCount; i < fractions.size(); ++i) {</pre>

// Truncate the string to have only 3 decimal places

// return "-1" indicating it's not possible to meet the target

// Initialize the error sum with the count of ceilings needed,

if (target < floorSum || target > floorSum + fractions.size()) {

if (fraction > 0) {

return "-1";

int ceilCount = target - floorSum;

for (int i = 0; i < ceilCount; ++i) {</pre>

string errorStr = to_string(errorSum);

size t decimalPos = errorStr.find('.');

double errorSum = ceilCount;

sort(fractions.rbegin(), fractions.rend());

double price = Double.parseDouble(priceStr);

// Calculate the fractional part of the price.

and calculate the floor sum and store the fractional parts

fractional_parts.append(fraction)

remaining_sum = target - floor_sum

fractional_parts.sort(reverse=True)

Iterate over the string representation of prices, convert them to floats,

plus the possible additions of 1 for each fractional part to form integers

Calculate the additional sum required to reach the target from the floor sum

5. Formatting the Result We format the total error to three decimal places, resulting in the answer: "1.600".

This example illustrates that by sorting the fractional parts and rounding up only the required number of smallest fractional parts, we

optimize the rounding error while achieving our target sum. Using this strategy allows us to determine the best numbers to round up

○ The two values we keep as floor values are the two 0.50s, and their errors remain 0.50 + 0.50 = 1.00.

Python Solution

fractional_parts = []

for price_str in prices: price = float(price_str) floor_sum += int(price) # Adding the integer part (floor) of the price # If the price is not an integer, store the fractional part if fraction := price - int(price):

Check if it is impossible to reach the target by summing the integer parts of prices

Sort the fractional parts in reverse order to prepare for minimizing the rounding error

Calculate the minimum error by adding the smallest fractional parts up to the remaining sum

class Solution: def minimizeError(self, prices: List[str], target: int) -> str: # Initialize the floor sum which represents the sum of all floor values of the prices floor_sum = 0

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

9

10

11

13

14

15

16

17

18

19

20

21

Java Solution

class Solution {

int floorSum = 0;

// Loop over each price.

for (String priceStr : prices) {

floorSum += (int) price;

if (fraction > 0) {

from typing import List

```
32
           # then, adding the count of flooring operations (since we've taken their fractions)
33
           min_error = sum(fractional_parts[remaining_sum:]) \
34
                        - sum(fractional_parts[:remaining_sum]) \
35
                        + remaining_sum
36
37
           # Format the result to 3 decimal places and return as a string
           return f'{min_error:.3f}'
38
39
```

```
38
39
40
41
42
```

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

47

48

49

50

51

57

58

59

60

};

```
22
                return "-1";
23
24
           // Calculate how many fractions we need to round up to reach the target.
           int roundUpCount = target - floorSum;
25
26
           // Sort the fractions in non-ascending order.
27
           fractions.sort(Collections.reverseOrder());
28
29
           // Start calculating the error by adding the number of fractions required to round up.
           double error = roundUpCount;
30
31
           // Subtract the largest fractions that we're rounding up.
32
           for (int i = 0; i < roundUpCount; ++i) {</pre>
33
               error -= fractions.get(i);
34
35
           // Add the remaining fractions that we're rounding down.
36
           for (int i = roundUpCount; i < fractions.size(); ++i) {</pre>
37
               error += fractions.get(i);
           // Format the error to 3 decimal places.
           DecimalFormat decimalFormat = new DecimalFormat("#0.000");
           return decimalFormat.format(error);
43 }
44
C++ Solution
  1 #include <vector>
  2 #include <string>
     #include <algorithm>
    class Solution {
    public:
         // Function to minimize the rounding error to meet a specific target sum
         string minimizeError(vector<string>& prices, int target) {
  8
             int floorSum = 0; // to store the sum of floor values of the prices
 10
             vector<double> fractions; // to store the fractional parts of the prices
 11
```

// Loop through the price strings, convert them to double and calculate floorSum and fractions

double price = stod(priceStr); // convert string to double

// If there's a fractional part, add it to the fractions list

// more than what can be achieved by taking the ceiling of all fractions

// Sort the fractions in order to maximize the lower error when rounding

// then adjust by subtracting fractions for ceilings and adding fractions for floors

errorSum -= fractions[i]; // subtract elements which are to be ceiled

errorSum += fractions[i]; // add elements which are to be floored

// Convert the double error sum to string with precision handling

floorSum += static_cast<int>(price); // accumulate the floor part

double fraction = price - floor(price); // calculate the fraction part

```
52
            if (decimalPos != string::npos) {
53
                errorStr = errorStr.substr(0, decimalPos + 4);
54
55
56
           // Return the error sum as a string
```

return errorStr;

```
Typescript Solution
    function minimizeError(prices: string[], target: number): string {
         let floorSum: number = 0; // To store the sum of floor values of the prices
         let fractions: number[] = []; // To store the fractional parts of the prices
  5
         // Loop through the price strings, convert them to numbers, and calculate floorSum and fractions
         prices.forEach(priceStr => {
             let price: number = parseFloat(priceStr); // Convert string to number
             floorSum += Math.floor(price); // Accumulate the floor part
             let fraction: number = price - Math.floor(price); // Calculate the fraction part
  9
 10
 11
             // If there's a fractional part, add it to the fractions array
 12
            if (fraction > 0) {
 13
                 fractions.push(fraction);
 14
         });
 15
 16
 17
        // If the target is less than the sum of floor values or more than the max possible by ceilings
         if (target < floorSum || target > floorSum + fractions.length) {
 18
             return "-1"; // Return "-1" indicating it's not possible to meet the target
 19
 20
 21
 22
        // Calculate the number of elements where we take the ceiling
 23
         let ceilCount: number = target - floorSum;
 24
 25
         // Sort the fractions to maximize the lower error when rounding
 26
         fractions.sort((a, b) => b - a);
 27
 28
         // Initialize the error sum with the count of ceilings needed,
         // then adjust by subtracting fractions for ceilings and adding fractions for floors
 29
 30
         let errorSum: number = ceilCount;
         for (let i = 0; i < ceilCount; i++) {</pre>
 31
 32
             errorSum -= fractions[i]; // Subtract elements that will be ceiled
 33
 34
         for (let i = ceilCount; i < fractions.length; i++) {</pre>
             errorSum += fractions[i]; // Add elements that will be floored
 35
 36
 37
 38
         // Convert the error sum to a string with controlled precision
         let errorStr: string = errorSum.toFixed(3);
 39
 40
 41
         // Return the error sum as a string
 42
         return errorStr;
 43 }
 44
```

1. Conversion of strings to floats: The loop converts each string in the prices list to a float. This process takes O(n) where n is the number of strings since each string is processed once.

Regarding the space complexity:

Therefore, the total space complexity of the function is O(n).

Time Complexity

Time and Space Complexity

3. Sorting fractional parts: The fractional parts are sorted in reverse order which has a time complexity of O(n log n).

The function minimizeError has several components, each contributing to the total time complexity:

integer using int(p) and collects the fractional part if any. This is also 0(n) time.

4. Summing specific portions of the array: The algorithm sums parts of the array, which is linear with respect to the number of elements summed. In the worst case, this is O(n). Therefore, the overall time complexity of minimizeError is dominated by the sorting step, which makes the time complexity O(n log

2. Rounding down floats and collecting fractional parts: Within the same loop, the code rounds down the float to the nearest

- n). **Space Complexity**
 - 1. Storing fractional parts: The array arr stores at most n fractional parts. Hence, its space complexity is O(n). 2. Return value formatting: String formatting for the return value does not use additional space that scales with the input size.

In this problem, we are given an array of prices and a target value. We are asked to round each price either up (ceil) or down (floor) to an integer such that the sum of the rounded prices equals the target. Our objective is to minimize the rounding error, which is