1402. Reducing Dishes

<u>Array</u>

Dynamic Programming

Sorting

Problem Description

Hard

achieve the maximum total "like-time coefficient," which is the sum of individual dish satisfaction levels multiplied by the time taken to cook the dish, including the time taken to prepare all previous dishes. This means that if a dish is cooked second, its satisfaction level is multiplied by 2, if third, by 3, and so on. The chef can choose to cook the dishes in any order and can also decide not to cook some dishes at all if it leads to a higher total

The problem presents a scenario where a chef has various dishes to cook, each with its own satisfaction level. The chef aims to

like-time coefficient. The key challenge is to determine which dishes should be cooked and in what order to maximize the total like-time coefficient.

coefficient. However, dishes with negative satisfaction levels can decrease the overall score if they are cooked before dishes with positive satisfaction values.

Intuition

So, the initial approach is to sort the dishes in descending order of satisfaction levels—this way, we look to cook the more satisfying dishes earlier. Then, we iterate through the sorted dishes, accumulating their satisfaction levels. If at any point the running total of satisfaction (sum s) becomes zero or negative, we stop adding more dishes; since adding a dish with a nonpositive cumulative satisfaction won't help in increasing the like-time coefficient.

The intuition behind the solution is that cooking dishes with higher satisfaction levels earlier usually leads to a higher like-time

During the iteration, we keep accumulating the running total s to a separate answer variable ans. ans represents the sum of liketime coefficients. The moment the running total s would not increase if we added another dish, we break out of the loop, because

including any further dishes (with a lesser or negative satisfaction level) would not increase the maximal like-time coefficient. The key observations are: 1. Dishes with higher satisfaction levels should be prioritized.

2. If the accumulated satisfaction level falls to zero or below, further dishes (with lower satisfaction values due to sorting) will not contribute

Solution Approach

3. By keeping track of a running total of satisfaction levels, we can decide when to stop adding more dishes to our final answer.

Sort the array: First, we sort the satisfaction levels of the dishes in descending order using satisfaction.sort(reverse=True). This ensures that we consider the dishes with the highest satisfaction levels first.

breakdown:

positively to the like-time coefficient.

maximum sum of like-time coefficients.

following satisfaction levels: [4, 3, -1, -8, -2].

after sorting it in descending order.

 \circ For the first dish with satisfaction 4, s += 4 giving s = 4.

 \circ Next dish with satisfaction 3, s += 3 giving s = 7.

 \circ Next dish with satisfaction -1, s += (-1) giving s = 6.

Accumulate satisfaction levels:

correctly, the elements should be arranged as [4, 3, -1, -8, -2].

We do not consider the last two dishes because adding them would lower the total.

def max_satisfaction(self, satisfaction: List[int]) -> int:

Update the cumulative satisfaction sum.

Return the maximum total satisfaction possible.

max_total_satisfaction += cumulative_satisfaction

merge sort and insertion sort.

Example Walkthrough

Initialize variables: We initialize two variables, ans and s, to zero. ans will keep track of our maximum sum of like-time coefficients, while s will be used to keep track of the cumulative sum of the satisfaction levels as we iterate over the dishes.

Iterate over sorted dishes: We use a for-loop to iterate through each dish's satisfaction level in the sorted list.

The implementation of the solution takes a simple, yet effective approach to solving the given problem. Here's a step-by-step

- Accumulate satisfaction levels: In each iteration, we add the satisfaction level of the current dish to our cumulative sum s with s += x.
- Check for the break condition: After updating s, we check whether s is less than or equal to zero with if s <= 0:. This check is crucial because once s becomes non-positive, adding more dishes to the preparation (especially the ones with lower
- Update the maximum sum: Only if s is positive do we proceed to add it to our answer ans with ans += s. This step effectively applies the like-time coefficient, accumulating the product of the satisfaction level and cooking order.

satisfaction levels that come later in the sorted list) would not increase the like-time coefficient.

increase the like-time coefficient. The efficiency of this solution lies in the fact that it leverages the sorted order of dishes (based on satisfaction level) and stops early if the future choices would not contribute positively to the outcome.

In terms of data structures, the solution doesn't require any advanced structures; it simply operates on the list of integers given

as input. The sort operation on the list is an in-built Python feature that uses TimSort, a hybrid sorting algorithm derived from

Overall, this approach is efficient, with a time complexity of O(n log n) due to the sorting step, and a space complexity of O(1),

This solution uses a greedy algorithm pattern where at each step it greedily chooses to cook the dish that can potentially

Return the result: Once the loop completes (either by exhaustion of the list or by breaking early), we return ans as the

since no additional space other than a few variables is used.

Let's use a small example to illustrate the solution approach. Consider that the chef has five dishes to choose from with the

Initialize variables: We set ans = 0 and s = 0. Iterate over sorted dishes: We go through each dish's satisfaction level in the sorted list, which goes as [4, 3, -1, -8, -2]

Sort the array: We sort the satisfaction levels in descending order, giving us [-1, -2, -8, 3, 4]. To sort in reverse order

The next dish has satisfaction -8, but before we add it, we check if s has become zero or negative. It's still positive, so we continue. \circ s += (-8) would give s = -2, a negative number, so we do not include this dish as it would decrease our total like-time coefficient.

the like-time coefficient.

Python

class Solution:

Solution Implementation

- Check for break condition: Once s becomes non-positive, we stop adding more dishes since future additions will not increase the like-time coefficient anymore.
- \circ After the second dish, ans = 4 + 7 = 11. \circ After the third dish, ans = 11 + 6 = 17.

 \circ After the first dish, ans = 0 + 4 = 4.

cumulative_satisfaction = 0

Iterate over each satisfaction value.

cumulative_satisfaction += value

max_total_satisfaction = 0

for value in satisfaction:

return max_total_satisfaction

Return the result: The total maximum sum of like-time coefficients based on the dishes chosen is $\frac{17}{2}$. By following this approach, the chef can easily figure out the best possible combination of dishes to cook in order to maximize

Update the maximum sum: After each addition to s that is positive, we also add that running total to ans:

Sort the satisfaction values in descending order to process the larger # values first, which is beneficial when cumulatively multiplying by the index. satisfaction.sort(reverse=True)

Initialize the running total of satisfaction (`cumulative_satisfaction`) and the answer.

Increment the answer by the current cumulative satisfaction value.

* @param satisfaction Array of satisfaction levels from different dishes.

If the cumulative satisfaction becomes non-positive, # no point in considering further values (since they are sorted in descending order). if cumulative_satisfaction <= 0:</pre> break

```
* This method finds the maximum total satisfaction by choosing a sub-sequence to cook in a sequence each
* element having a satisfaction level represented by the array satisfaction. More formally, it maximizes
* the sum of the satisfaction[i] * (i+1), where i is the order of cooking in the sequence and satisfaction[i]
* is the satisfaction from the i-th dish.
```

#include <algorithm>

// Function to calculate the maximum satisfaction

int maxSatisfaction(vector<int>& satisfaction) {

for (int satValue : satisfaction) {

maxSatisfaction += runningSum;

cumulative_satisfaction += value

if cumulative_satisfaction <= 0:</pre>

break

return max_total_satisfaction

Time and Space Complexity

If the cumulative satisfaction becomes non-positive,

max_total_satisfaction += cumulative_satisfaction

Return the maximum total satisfaction possible.

Increment the answer by the current cumulative satisfaction value.

if (runningSum <= 0) {</pre>

break;

return maxSatisfaction;

// Sorting the satisfaction values in non-increasing order

int maxSatisfaction = 0; // This will store the maximum total satisfaction

// If the running sum becomes non-positive, we should stop

int runningSum = 0; // This keeps the running sum of satisfaction values

runningSum += satValue; // Update the running sum with the current value

// because adding more negative values will not increase the total satisfaction

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

// Iterate through the sorted satisfaction values

// Update the maximum satisfaction value

// Return the computed maximum satisfaction value

class Solution {

public:

class Solution {

Java

```
* @return The maximum total satisfaction that can be achieved.
    public int maxSatisfaction(int[] satisfaction) {
       // First, sort the satisfaction levels to potentially cook less satisfying dishes earlier.
       Arrays.sort(satisfaction);
        int totalSatisfaction = 0; // Stores the maximum total satisfaction that can be achieved.
        int cumulativeSatisfaction = 0; // Used in calculating the total satisfaction iteratively.
       // Start with the most satisfying dish and move towards the least satisfying.
        for (int i = satisfaction.length - 1; i >= 0; --i) {
            cumulativeSatisfaction += satisfaction[i];
            // If the cumulative satisfaction becomes non-positive then no need to consider previous dishes
           // as it will not contribute to increasing the total satisfaction anymore.
            if (cumulativeSatisfaction <= 0) {</pre>
                break;
            // Update the total satisfaction.
            totalSatisfaction += cumulativeSatisfaction;
       // Return the maximum total satisfaction.
       return totalSatisfaction;
#include <vector>
```

```
TypeScript
  // This function calculates the maximum total satisfaction
  // by sorting the array and choosing dishes to maximize
  // the sum of (satisfaction of the i-th dish * (i+1)).
  function maxSatisfaction(satisfaction: number[]): number {
      // Sort the satisfaction array in descending order to consider the most satisfying dishes first.
      satisfaction.sort((a, b) => b - a);
      // Initialize variables for the maximum answer (ans) and a running sum (runningSum).
      let maxTotalSatisfaction = 0;
      let runningSum = 0;
      // Loop through each satisfaction value in the sorted array.
      for (const satValue of satisfaction) {
          // Add the current satisfaction value to the running sum.
          runningSum += satValue;
          // If the running sum becomes non-positive, it's not beneficial to serve more dishes.
          if (runningSum <= 0) {</pre>
              break;
          // Update the maximum total satisfaction so far by adding the current running sum.
          maxTotalSatisfaction += runningSum;
      // Return the maximum total satisfaction value.
      return maxTotalSatisfaction;
class Solution:
   def max_satisfaction(self, satisfaction: List[int]) -> int:
       # Sort the satisfaction values in descending order to process the larger
       # values first, which is beneficial when cumulatively multiplying by the index.
        satisfaction.sort(reverse=True)
       # Initialize the running total of satisfaction (`cumulative_satisfaction`) and the answer.
        cumulative_satisfaction = 0
        max_total_satisfaction = 0
       # Iterate over each satisfaction value.
        for value in satisfaction:
            # Update the cumulative satisfaction sum.
```

Sorting the satisfaction list has a time complexity of O(N log N), where N is the length of the satisfaction list. This is because the sort function in Python uses the Timsort algorithm, which is a combination of merge sort and insertion sort.

Time Complexity

thus the time complexity of the for-loop is O(N). Since these operations are performed sequentially, the overall time complexity of the algorithm is the sum of the two

The time complexity of the provided code primarily comes from the sorting operation and the for-loop.

no point in considering further values (since they are sorted in descending order).

- complexities however, the dominating factor here is the sorting operation. Therefore, the overall time complexity of the code is O(N log N).
- **Space Complexity** The space complexity of the code comes from the extra space used to sort the satisfaction array and the variables ans and s

used to compute the result. Sorting the array in Python is done in-place, which means that no extra space proportional to the

After sorting, the code iterates over the sorted satisfaction list, performing a constant amount of work for each element,

input size is used, so the space complexity would be 0(1) for the sorting operation. However, if the implementation of the sort function creates a copy of the array, then the space complexity would be O(N).

As the only extra variables used are ans and s, both of which use a constant amount of space, the space complexity due to these variables is 0(1).

Therefore, the overall space complexity of the code is 0(1) if the sort is done in-place. If not, it would be 0(N) due to the space required for a copy of the array during sorting.