1788. Maximize the Beauty of the Garden

Prefix Sum

Problem Description

Hard

Greedy Array

In this problem, we are working with a garden represented by a linear arrangement of flowers, each flower having a beauty value, which is a numerical value associated with it.

- A garden is considered *valid* if:
- The beauty value of the first flower is equal to the beauty value of the last flower in the garden. As the gardener, your job is to possibly remove any number of flowers from the garden to ensure that it becomes valid while

It contains at least two flowers.

maximizing the total beauty of the garden. The total beauty of the garden is defined as the sum of the beauty values of all the flowers that are left.

To summarize, the objective is to maximize the sum of the beauty values of a valid garden by selectively removing flowers. The solution to this challenge will require an efficient algorithm to calculate the maximum possible beauty without having to try all possible combinations of flower removal, which would be inefficient.

Intuition To find the maximum beauty of a valid garden, we analyze the following key points:

As the valid garden must start and end with flowers having the same beauty value, and we aim to maximize beauty, we must

Given point 1, removing any flowers with a negative beauty value from the middle of our garden will only increase the total

identify pairs of flowers with the same value that are as far from each other as possible, since all flowers between them can contribute to the total beauty.

beauty since they detract from it.

- With these points in mind, we can sketch an algorithm that: A. Keeps a running sum (s) of the beauty values (ignoring negative values as they are never beneficial to our sum), to quickly calculate the total beauty including any range of flowers.
- the flower array. As we iterate over the flowers array:

value twice (once for each flower at the ends) and subtracting the sum of values between them (as stored in s).

B. Utilizes a dictionary (d) to keep track of the last occurrence index for each beauty value we come across as we iterate through

If we encounter a flower with a beauty value we've seen before, we have a potential garden that starts at the previous

occurrence index and ends at the current index. We calculate the potential beauty for this garden by adding the total beauty

We update our maximum beauty (ans) if the current potential beauty is greater than what we have found so far. If we encounter a flower with a beauty value for the first time, we simply record its index in the dictionary.

The running sum (s) is updated at every step, adding the current flower's beauty value, but skipping negative values. By iterating through the entire array just once, we are able to calculate the maximum possible garden beauty by considering all pairs of same-beauty flowers and calculating the maximum sum efficiently.

The solution to this LeetCode problem involves finding subarrays with the same starting and ending beauty values and

maximizing the sum of the beauty values of the elements in between. The algorithm can be broken down into the following steps,

beauty values from flowers[0] up to flowers[i-1], excluding negative values, as they reduce the total beauty. The prefix

Solution Approach

The given solution code implements this approach and correctly calculates the answer.

Finding Maximum Beauty Garden: As we iterate over the flowers array:

prefix sum at the end index and adding the value of the end points twice.

Dictionary Tracking (d): We'll start with an empty dictionary.

1] + flowers [3] * 2 = 1 - 1 + 2 = 2. This is our first potential garden, so we set ans to 2.

incorporating efficient data structures and patterns: Prefix Sum Array: As an optimization for quick sum queries, a prefix sum array s is created. s[i] represents the sum of all

sum array allows for constant-time range sum queries, which is crucial for efficient computation. Dictionary Tracking: A dictionary d is used to keep track of the last index where each beauty value appears. This is used to find the most distant matching pair of the same beauty value, which potentially forms the start and end of a valid garden.

• If we encounter a beauty value that has appeared before, it means there's a potential valid garden ending with the current flower. The

beauty of this potential garden is calculated as the sum of all the values between the matching pair, including the value at the end points

themselves (which are same due to the valid garden condition). We calculate this by subtracting the prefix sum at the start index from the

o If it's the first time we encounter a certain beauty value, we record its index in the dictionary d, marking it as the start point for potential

- gardens with that beauty value. Maintaining Maximum Beauty Value: Throughout the iteration, we maintain a variable ans, which stores the maximum beauty value found so far. Whenever we find a potential garden with a higher beauty value than our current maximum, we update ans.
- * 2 (for the start and end flowers). Update the dictionary d for the current beauty value to the current index so that we can use the most recent index for calculating future gardens.

By following these steps, the provided Python code efficiently loops through the flower array once and finds the maximum

possible beauty of a valid garden. This approach leverages the usage of a prefix sum array for fast range sum calculations and a

o If flowers[i] has occurred before, compute the potential beauty by subtracting s[d[flowers[i]] + 1] from s[i] and adding flowers[i]

• For each flower flowers[i], the current flower's beauty is added to the prefix sum only if it is non-negative.

dictionary for O(1) look-up of indices, making the overall time complexity O(n), where n is the number of flowers.

We wish to identify a valid garden with the maximum total beauty. We will use the given solution approach to solve it.

Example Walkthrough Let's illustrate the solution approach with a small example. Suppose we have the following array of flowers with their respective beauty values:

Prefix Sum Array (s): We will construct this array on the fly as we iterate through the flowers array. Let's initialize the prefix

• For flowers [0] which has a beauty value of 1, d[1] does not exist yet, so we add it to d with a value of 0 and set the prefix sum s[0] to \circ flowers [1] with a beauty value of -2 is negative, so we do not add it to our prefix sum. Prefix sum remains s[0] = 1. • flowers [2] has a beauty of -1, again negative, so we continue. Prefix sum is still s[0] = 1. • When we reach flowers [3] with the beauty value of 1, d[1] exists and suggests a potential valid garden from index 0 to 3. We compute

the beauty by checking the prefix sum at the start index 0 and end index 3. The total beauty of this potential garden is s[3] - s[d[1] +

• Next, flowers [4] has a beauty value of 3. We add its index to d as d[3] = 4 and update our prefix sum to include this flower's positive

Maintaining Maximum Beauty Value (ans): Throughout the iteration, we have updated ans whenever we found a potential

By the end of this process, we determine that taking the entire array from index 0 to 5 gives us the maximum beauty of 5,

garden with a higher beauty value. First, ans was 2, but after considering the full array, we updated ans to 5.

• For each flower flowers[i], if the flower's beauty value is positive, it is added to the prefix sum. If not, it is ignored.

• When a flower's beauty value has been encountered before, calculate the potential beauty as described above.

The 'running sum' list holds the cumulative sum of the flowers' beauty values

// Initialize a prefix sum array that includes an extra initial element set to 0

// Create a map to keep track of the first index of each flower value encountered

// If the value has not been seen, map the current value to its first index

// Calculate the running total of positive values to form the prefix sums array

Update the dictionary d with the most recent index for the current beauty value.

resulting in a valid garden with [1, -2, -1, 1, 3, 1].

def maximumBeauty(self, flowers: List[int]) -> int:

for index, value in enumerate(flowers):

If we have seen the value before

Enumerate through the flowers list with index and value

Update the 'last seen' index for the current flower

• Lastly, flowers [5] has a beauty value of 1. Again, d[1] exists, so we check this potential garden which spans from index 0 to 5. The total beauty is s[5] - s[d[1] + 1] + flowers[5] * 2 = 4 - 1 + 2 = 5. This potential garden has a higher beauty than our current ans, so we

update ans to 5.

Update Rules:

Python

Java

class Solution {

class Solution:

Update Rules:

flowers = [1, -2, -1, 1, 3, 1]

sum array with 0.

Finding Maximum Beauty Garden:

• Iterating over the given flowers array:

beauty value, making s[4] = 4.

Solution Implementation

last_seen = {}

max_beauty = float('-inf')

if value in last seen:

last_seen[value] = index

public int maximumBeautv(int[] flowers) {

int maxBeauty = Integer.MIN_VALUE;

// Iterate through the flowers array

int currentValue = flowers[i]:

// Return the maximum beauty encountered

for (int i = 0; i < flowers.length; ++i) {</pre>

int[] prefixSums = new int[flowers.length + 1];

Map<Integer, Integer> firstIndexMap = new HashMap<>();

firstIndexMap.put(currentValue, i);

// Initialize the answer to the minimum possible integer value

// Check if the current flower value has been seen before

prefixSums[i + 1] = prefixSums[i] + Math.max(currentValue, 0);

// Array to track cumulative sum of positive beauty values up to each flower

const beautyValue = flowers[i].beauty; // Current flower's beauty value

let maxBeauty: number = Number.MIN_SAFE_INTEGER; // Variable to track the maximum beauty

// Map to track the last index at which a particular beauty value appeared

const lastAppearance: Map<number, number> = new Map<number, number>();

const cumulativeSum: number[] = new Array(numFlowers + 1).fill(0);

// Loop through all flowers to find the maximum possible beauty

// Record the first appearance of this beauty value

// Update the cumulative sum, ignoring negative beauty values

Update the 'last seen' index for the current flower

running_sum[index + 1] = running_sum[index] + max(value, 0)

cumulativeSum[i + 1] = cumulativeSum[i] + Math.max(beautyValue, 0);

The 'running sum' list holds the cumulative sum of the flowers' beauty values

Dictionary 'last_seen' keeps track of the last seen index for each flower value

Initialize 'ans' to negative infinity to handle the case when no valid scenario is found

Update the maximum beauty. It is the maximum of the current maximum beauty and

It has an extra initial element 0 for easier calculations of sums in slices

for (let i = 0; i < numFlowers; ++i) {

maxBeauty = Math.max(

maxBeauty,

);

last_seen = {}

return max_beauty

Time and Space Complexity

operations performed within this loop.

max_beauty = float('-inf')

last_seen[value] = index

Update the cumulative sum

Return the calculated maximum beauty

} else {

if (lastAppearance.has(beautyValue)) {

lastAppearance.set(beautyValue, i);

def maximumBeautv(self, flowers: List[int]) -> int:

running_sum = [0] * (len(flowers) + 1)

return maxBeauty; // Return the calculated maximum beauty

It has an extra initial element 0 for easier calculations of sums in slices running_sum = [0] * (len(flowers) + 1)# Dictionary 'last_seen' keeps track of the last seen index for each flower value

Initialize 'ans' to negative infinity to handle the case when no valid scenario is found

Update the maximum beauty. It is the maximum of the current maximum beauty and

Update the cumulative sum running_sum[index + 1] = running_sum[index] + max(value, 0) # Return the calculated maximum beauty return max_beauty

the sum of beauty values since the flower was last seen (not inclusive) plus the beauty value doubled

 $max_beauty = max(max_beauty, running_sum[index] - running_sum[last_seen[value] + 1] + value * 2)$

if (firstIndexMap.containsKev(currentValue)) { // Calculate the beauty using the current and first occurrence of currentValue // and update maxBeauty accordingly int firstIndex = firstIndexMap.get(currentValue); maxBeauty = Math.max(maxBeauty, prefixSums[i] - prefixSums[firstIndex + 1] + currentValue * 2); } else {

return maxBeauty;

#include <vector>

#include <algorithm>

#include <unordered map>

#include <climits> class Solution { public: // Calculates the maximum beauty of a garden based on the flowers' 'beauty' values int maximumBeautv(vector<int>& flowers) { int numFlowers = flowers.size(); // Number of flowers in the garden // Create a sum vector, where s[i] will hold the accumulated sum of all // flowers' beauty that can contribute positively up to index i-1 vector<int> cumulativeSum(numFlowers + 1, 0); // Dictionary to store the last index of a particular 'beauty' value flower unordered map<int, int> lastAppearance; int maxBeauty = INT_MIN; // The maximum beauty to be calculated // Iterate over all flowers to find the maximum beauty for (int i = 0; i < numFlowers; ++i) { int beautyValue = flowers[i]; // Current flower's beauty value // Check if this beauty value has appeared before if (lastAppearance.count(beautyValue)) { // Update the maxBeauty with the sum between first appearance of this value // and this appearance, plus double the beautyValue maxBeauty = max(maxBeauty, cumulativeSum[i] - cumulativeSum[lastAppearance[beautyValue] + 1] + beautyValue * 2);} else { // Keep track of the first appearance of this beauty value lastAppearance[beautyValue] = i; // Update the cumulative sum with the current beauty value if it's positive cumulativeSum[i + 1] = cumulativeSum[i] + max(beautyValue, 0); return maxBeauty; // Return the highest calculated beauty **}**; **TypeScript** // Structure representing a Flower, holds the beauty value interface Flower { beauty: number; // Function to calculate maximum beauty of a garden based on flowers' beauty values function maximumBeauty(flowers: Flower[]): number { const numFlowers: number = flowers.length; // Number of flowers in the garden

// Update maxBeauty using the sum of beauty values between two appearances, plus double the beautyValue

cumulativeSum[i] - cumulativeSum[lastAppearance.get(beautyValue)! + 1] + beautyValue * 2

Enumerate through the flowers list with index and value for index, value in enumerate(flowers): # If we have seen the value before if value in last seen:

class Solution:

the beauty of a sequence of flowers represented by an integer array. The method employs a dynamic programming approach and uses additional storage to keep track of sum results and indices. **Time Complexity:** The time complexity of the method maximumBeauty can be analyzed based on the single loop through the flowers array and the

The given Python code snippet defines a Solution class with a method maximumBeauty that calculates a specific value related to

the sum of beauty values since the flower was last seen (not inclusive) plus the beauty value doubled

 $max_beauty = max(max_beauty, running_sum[index] - running_sum[last_seen[value] + 1] + value * 2)$

• Within this loop, operations such as checking if a value is in a dictionary, updating the dictionary, computing a maximum, and updating the prefix sum array are all 0(1) operations. • Therefore, the dominating factor of time complexity is the loop itself, which gives us a total time complexity of O(n).

- **Space Complexity:**
- The space complexity of the method maximumBeauty is determined by the additional space used by the data structures s and d. • The prefix sum array s is of length len(flowers) + 1, which introduces an O(n) space complexity where n is the length of flowers.
- The dictionary d stores indices of distinct elements encountered in flowers. In the worst case, all elements of flowers are distinct, resulting in an O(n) space complexity, where n is the length of flowers.

• There is a loop that iterates through each element of the input list flowers, which gives us an O(n) where n is the length of flowers.

- The final space complexity is the sum of the complexities of the prefix sum array and the dictionary, both of which are O(n).
- Thus, the overall space complexity of the method maximumBeauty is O(n) as well.