1695. Maximum Erasure Value Medium Hash Table **Sliding Window** <u>Array</u>

Problem Description

subarray consists of consecutive elements from the given array, and a score is calculated by taking the sum of the unique elements within that subarray. We are looking for the subarray with the highest sum where all elements are unique. The challenge is to devise an efficient algorithm to perform this action rather than using brute-force, which could be very slow for large arrays. Intuition

The task is to find the maximum score you can obtain by erasing exactly one subarray from an array of positive integers nums. A

The intuition behind the solution is to utilize a sliding window approach along with a hash table (or dictionary in Python) to keep track of the most recent index of each number in the array. The sliding window allows us to consider a subarray of unique elements which we can dynamically resize as we traverse the array.

2. Keep a cumulative sum array which will help in getting the sum of any subarray in constant time. 3. Set two pointers, j starting at the beginning of the array and i which we will move forward through the array.

Here's an outline of our approach:

4. For each element v at index i: Update j to be the maximum of it own current value or the index stored in the hash table for the current element v. This is required to

1. Initialize a dictionary or hash table to store the most recent index where each element has appeared as we iterate through the array.

- ensure our window does not include duplicate values.
- Calculate the sum of elements in the current window (subarray without duplicates) using the cumulative sum array and update the answer if this sum is greater than the previously recorded answer.
 - 5. Return the maximum sum recorded during this process. The use of the cumulative sum and a hash table allows us to efficiently calculate the sum of each potential unique subarray and track the presence of duplicates respectively. The sliding window dynamically adjusts to skip over elements that would lead to
- duplication, thus ensuring that at all times, the subarray being considered is unique. The max operations ensure that the window only expands when needed and shrinks appropriately to exclude duplicates.

Update the hash table with the current index for the element v.

Solution Approach The implementation of the solution uses a sliding window approach to consider every possible subarray that contains unique elements. Here is the detailed walk-through:

appeared.

Data Structures used:

The dictionary d starts off empty.

greater than ans, update ans.

a unique subarray is found.

 An array s, which is a cumulative sum array, such that s[i] is the sum of the first i numbers in nums. Initialization:

A dictionary d with integer keys and integer values, which serves as a hash table to keep track of the last index at which each element has

- **Sliding Window:** Define two pointers: i represents the current position in nums, and j is the start index of the window which ensures all elements to its
 - right, up to i, are unique. An ans variable is initialized to 0 for storing the maximum sum of subarrays encountered.

For each element v, check for duplicates by looking up v in the hash table d. If v is found, update j to be the maximum of its current

While keeping j as the start of the unique window, compute the sum of the current window by subtracting s[j] from s[i]. If this sum is

• The cumulative sum array s is initialized with 0 as the first element and the cumulative sums of nums after that.

Iterating through nums: The outer loop goes through each element v at index i (1-indexed for simplicity, due to the initial 0 in s).

value or the one stored in d[v], the index after the last occurrence of v.

Update the hash table d[v] with the new index i to mark the most recent occurrence of v.

- **Maintaining the Maximum Score:** As the loop progresses, the ans variable keeps track of the maximum score seen so far, and it is updated accordingly when a larger sum of
- **Return the Result:** After the loop has finished iterating over all elements in nums, ans holds the maximum score attainable by erasing exactly one subarray of unique elements. It is returned as the final answer.

Through the use of cumulative sums, we are able to tell the sum of the subarray nums[j+1...i] in constant time by a simple

subtraction: s[i] - s[j]. Using the hash map d, we can dynamically adjust our unique window's start position j whenever a

duplicate discovers that it has appeared previously in the window. The algorithm efficiently computes the maximum sum subarray with no duplicates with a time complexity of O(N), where N is the size of the array nums, since each element is processed exactly

Using Data Structures:

once.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the array nums as [3, 2, 1, 2, 3, 3]. We want to find the maximum sum of a subarray with unique elements after erasing exactly one subarray.

 We initialize an empty dictionary d. ○ We also create a cumulative sum array s=[0, 3, 5, 6, 8, 11, 14], representing the cumulative sum up to each index. **Initializing Variables:** The dictionary d is {}. The cumulative sum array s is [0, 3, 5, 6, 8, 11, 14]. Starting with a Sliding Window:

3. \circ At i=2, nums[i] is 2. No 2 in d, so j stays. Add 2 to d with value 2. ans becomes s[2] - s[j] = 5 - 0 = 5.

Iteration:

nums.

efficiently.

class Solution:

Python

Java

class Solution {

class Solution {

public:

Solution Implementation

from collections import defaultdict

index dict = defaultdict(int)

max_sum = start_index = 0

from itertools import accumulate

 \circ At i=3, nums[i] is 1. No 1 in d, so j stays. Add 1 to d with value 3. ans becomes s[3] - s[j] = 6 - 0 = 6. ∘ At i=4, nums[i] is 2 again, but d does contain 2, so we update j to max(j, d[2] + 1) = max(0, 3) = 3. Update 2 in d with 4. ans is still 6 because s[4] - s[j] = 8 - 6 = 2.

By employing this method, we have considered all subarrays with unique elements by dynamically adjusting the beginning of the

 \circ At i=5, nums[i] is 3. As 3 is in d, update j to max(j, d[3] + 1) = max(3, 2) = 3. Update 3 in d with 5. ans remains 6.

 \circ At i=6, nums[i] is 3. As 3 is in d, we update j to max(j, d[3] + 1) = max(3, 6) = 6. Update 3 in d with 6. ans remains 6.

∘ At i=1, nums[i] is 3. d does not contain 3, so no change to j. We add 3 to d with the value 1. ans becomes s[1] - s[j] = 3 - 0 =

- **Maintaining Maximum Score:** • Each iteration considers whether the current unique window sum is greater than ans. The variable ans is kept up to date at all times.
 - **Returning Result:** • After completing the loop, we have ans = 6, which is the maximum score obtainable by erasing one subarray of unique elements from

def maximumUniqueSubarray(self, nums: List[int]) -> int:

prefix sums = list(accumulate(nums, initial=0))

index_dict[value] = current_index

public int maximumUniqueSubarray(int[] nums) {

// Iterate over the nums array

for (int i = 1; i <= length; ++i) {</pre>

int maximumUniqueSubarrav(vector<int>& nums) {

// Store the size of the nums vector

let frequency: number[] = new Array(10001).fill(0);

// Store the size of the nums array

// Populate the prefixSum array

let maxSum: number = 0;

let startIndex: number = 0;

for (let i = 0; i < numsSize; ++i) {</pre>

for (let i = 1: i <= numsSize: ++i) {</pre>

let currentValue = nums[i - 1];

const numsSize: number = nums.length;

function maximumUniqueSubarray(nums: number[]): number {

// Create an array to store the prefix sum of nums

let prefixSum: number[] = new Array(numsSize + 1);

prefixSum[i + 1] = prefixSum[i] + nums[i];

// Initialize the maximum sum of unique elements

// Initialize the start index of the current subarray

// Return the maximum sum of a unique element subarray

def maximumUniqueSubarray(self, nums: List[int]) -> int:

prefix sums = list(accumulate(nums, initial=0))

for current index, value in enumerate(nums, 1):

initial 0). Thus, it also consumes 0(n) space.

Combining the above, the total space complexity remains O(n).

// Function to calculate the maximum sum of unique-elements subarray

int frequency[10001]{};

int value = nums[i - 1]; // Current value

int length = nums.length; // Length of nums

Return the maximum subarray sum containing unique elements

int[] prefixSum = new int[length + 1]; // Array to store prefix sum

int maxSum = 0; // Initialize the maximum sum of a unique subarray

int windowStart = 0; // Initialize the start of the current window

return maxSum; // Return the maximum sum of a unique subarray

maxSum = Math.max(maxSum, prefixSum[i] - prefixSum[windowStart]);

lastIndex[value] = i; // Update the last index of the current value

// Create a frequency array initialized to zero for the possible range of values in nums

// Define the array of frequencies with a default value of zero for a range of possible values in nums array

prefixSum[0] = 0; // Initialize the first element as 0 for correct prefix sum calculation

// Iterate through the nums array to find the max sum of a unique-element subarray

// Update the startIndex to be the maximum of the current startIndex or the

// Calculate the maxSum by considering the current unique subarray sum

Initialize a dictionary to store the most recent index of each number

Create a prefix sum array with an initial value of 0 for convenience

Iterate over the numbers alongside their indices (starting from 1)

Update the maximum sum if the current subarray sum is greater

after the previous occurrence of the current value

start index = max(start index, index dict[value])

Initialize the maximum subarray sum and the start index for the subarray

Update the start index to the maximum of the current start_index and the next index

startIndex = Math.max(startIndex, frequency[currentValue]);

// last index where currentValue was found (to maintain uniqueness in subarray)

Initialize a dictionary to store the most recent index of each number

Create a prefix sum array with an initial value of 0 for convenience

Initialize the maximum subarray sum and the start index for the subarray

int[] lastIndex = new int[10001]; // Array to store the last index of each number

Two pointers are set: i starts at 1 and j starts at 0.

A variable ans is initialized to hold the maximum sum, starting at 0.

subarray using the j pointer. Instead of calculating the sum each time, we utilized the cumulative sum array for quick calculations. The dictionary d helped maintain indices of previous occurrences. Thus, we arrived at the maximum score

Iterate over the numbers alongside their indices (starting from 1) for current index, value in enumerate(nums, 1): # Update the start index to the maximum of the current start_index and the next index # after the previous occurrence of the current value start index = max(start index, index dict[value]) # Update the maximum sum if the current subarray sum is greater current subarray sum = prefix sums[current index] - prefix_sums[start_index] max sum = max(max sum, current subarray sum) # Update the dictionary with the current index of the value

windowStart = Math.max(windowStart, lastIndex[value]); // Update the start of the window to be after the last occurrence

// Calculate the current sum of unique subarray and compare it with the maximum sum found so far

// Calculate prefix sum array for (int i = 0; i < length; ++i) {</pre> prefixSum[i + 1] = prefixSum[i] + nums[i];

return max_sum

```
int numsSize = nums.size();
        // Create an array to store the prefix sum of nums
        int prefixSum[numsSize + 1];
        prefixSum[0] = 0; // Initialize the first element as 0 for correct prefix sum calculation
        // Populate the prefixSum array
        for (int i = 0; i < numsSize; ++i) {</pre>
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        // Initialize the maximum sum of unique elements
        int maxSum = 0;
        // Initialize the start index of the current subarray
        int startIndex = 0;
        // Iterate through the nums array to find the max sum of a unique-subarray
        for (int i = 1; i <= numsSize; ++i) {</pre>
            int currentValue = nums[i - 1];
            // Update the startIndex to be the maximum of the current startIndex or the
            // last index where currentValue was found (to ensure uniqueness in subarray)
            startIndex = max(startIndex, frequency[currentValue]);
            // Calculate the maxSum by considering the current unique subarray sum
            maxSum = max(maxSum, prefixSum[i] - prefixSum[startIndex]);
            // Update the index in frequency array to the current position for currentValue
            frequency[currentValue] = i;
        // Return the maximum sum of a unique element subarray
        return maxSum;
};
TypeScript
```

maxSum = Math.max(maxSum, prefixSum[i] - prefixSum[startIndex]); // Update the index in frequency array to the current position for currentValue frequency[currentValue] = i;

return maxSum;

class Solution:

from collections import defaultdict

index dict = defaultdict(int)

max_sum = start_index = 0

from itertools import accumulate

current subarray sum = prefix sums[current index] - prefix_sums[start_index] max sum = max(max sum, current subarray sum) # Update the dictionary with the current index of the value index_dict[value] = current_index # Return the maximum subarray sum containing unique elements return max_sum Time and Space Complexity

The time complexity of the code is O(n), where n is the length of the nums array. This is because the code involves a single loop

that iterates over nums, and within this loop, the operations performed (accessing the dictionary d, updating the sliding window's

sum using the prefix sum s, and calculating the maximum ans) are all constant time operations. **Space Complexity**

Time Complexity

The space complexity of the code is also O(n) due to the following reasons: The d dictionary stores each unique value encountered in the nums array with its latest index position. In the worst case, if all

- elements are unique, the dictionary could hold up to n key-value pairs. The s list is a prefix sum array which contains a cumulative sum of the nums array, and it has n + 1 elements (including the