1838. Frequency of the Most Frequent Element Greedy Array Binary Search Prefix Sum Sorting Sliding Window Medium

Problem Description

you're allowed to increment any element by 1 up to k times. The frequency of an element is how many times that element appears in the array. The objective is to find the highest possible frequency of any number in the array after making at most k increments in total.

You are given an integer array nums and an integer k. The task is to find the maximum frequency of any element in nums after

Intuition

The intuition behind the solution is to first sort the array. By doing this, we can then use a sliding window to check each subarray and calculate how many operations we would need to make all elements in the subarray equal to the rightmost element.

Starting with the smallest elements on the left, we work towards the right. For each position r (right end of the window), we calculate the number of operations needed to raise all elements from 1 (left end of the window) to r to the value of the element

at position r. This is done by multiplying the difference between nums[r] and nums[r - 1] by r - 1 (size of the current window) minus one). If the total operations required exceed k, we shrink the window from the left by increasing 1, subtracting the necessary operations as we go to keep the window valid (total operations less than or equal to k). Meanwhile, we're always keeping track of

the maximum window size we're able to achieve without exceeding k operations, as this directly corresponds to the maximum

frequency. The solution uses a two-pointer technique to maintain the window, expanding and shrinking it as needed while traversing the sorted array only once, which gives an efficient time complexity. Solution Approach

the left and right ends of the window, respectively. The fundamental data structure used here is the list (nums), which is sorted to facilitate the sliding window logic. Here is a step-by-step breakdown:

The algorithm implemented in the given solution follows a <u>sliding window</u> approach using two pointers, 1 and r, which represent

Sort the Array: The array nums is sorted to allow comparing adjacent elements and to use the sliding window technique

nums.sort()

l, r, window = 0, 1, 0

the value of nums[r].

the limit again.

while window > k:

l += 1

return ans

Example Walkthrough

•

•

•

•

Python

as the window expands or shrinks.

ans = $\max(1, 1 - 0) = 1$.

Increment r and it becomes 2.

window += (nums[r] - nums[r - 1]) * (r - 1)

window -= nums[r] - nums[l]

Initialize Pointers and Variables: Three variables are initialized:

ans = 1 # A single element has at least a frequency of 1.

maximum frequency that can be achieved, and we return it.

Let's consider a small example to illustrate the solution approach in action:

window which holds the total number of operations performed in the current window.

effectively.

• 1 (left pointer) which starts at 0. r (right pointer) which starts at 1 since the window should at least contain one element to make comparisons. ans which keeps track of the maximum frequency found.

Expand the Window: The algorithm enters a loop that continues until the right pointer r reaches the end of the array. while r < len(nums):</pre>

Calculate Operations: In each iteration, calculate the operations needed to increment all numbers in the current window to

Shrink the Window: If window exceeds k, meaning we cannot increment the elements of the current window to match nums[r] with the allowed number of operations, move 1 to the right to reduce the size of the window until window is within

frequency $\frac{1}{2}$ if the current window size (r - 1) is greater than the previous maximum.

Enter the loop. nums[r] is 2, and nums[l] is 1. window becomes (2 - 1) * (1 - 0) = 1.

Again, window $\ll k$, so we do not shrink the window. Update ans $= \max(1, 2 - 0) = 2$.

Increment r and it becomes 3. Since r is now equal to len(nums), the loop ends.

Next iteration, nums [2] is 3. We calculate operations needed: window += (3 - 2) * (2 - 0) = 1 + 2 = 3.

```
r += 1
ans = max(ans, r - 1)
 Return the Result: Once we have finished iterating through the array with the right pointer r, the ans variable holds the
```

The sliding window technique coupled with the sorted nature of the array allows for an efficient solution. This technique avoids

recalculating the increment operations for each window from scratch; instead, it updates the number of operations incrementally

Update Maximum Frequency: After adjusting the window to ensure we do not exceed k operations, update the maximum

Suppose we have an array nums = [1, 2, 3] and k = 3. First, we sort nums, which in this case is already sorted: [1, 2, 3]. Initialize the pointers and variables: l = 0, r = 1, window = 0, and ans = 1.

Since window $\leq k$, we do not need to shrink the window. We then update ans with the new window size, which is r - 1.

Putting it all together, we found that after incrementing the elements optimally, we can have at least two elements with the same

value in the array [1, 2, 3] by using at most 3 increments (for example: incrementing 1 and 2 to make the array [3, 3, 3]).

Return ans. So the maximum frequency after k increments is 2.

Solution Implementation

while right < n:</pre>

return max_freq

Arrays.sort(nums);

// Start with two pointers:

sol = Solution()

class Solution {

from typing import List

class Solution:

Thus, the maximum frequency is 2.

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

max_freq, cumulative_diff = 1, 0

while cumulative diff > k:

public int maxFrequency(int[] nums, int k) {

// 'left' at 0 for the start of the window,

while (operationsSum > k) {

int maxFrequency(std::vector<int>& nums, int k) {

// Initialize variables: n is the size of the vector,

// 'window' represents the total increments needed to make

// Using two pointers technique: l -> left, r -> right.

// element nums[right] to the value of nums[right - 1].

for (int left = 0, right = 1; right < n; ++right) {</pre>

window -= (nums[right] - nums[left]);

// all elements in the current window equal to the current maximum element.

// Update the 'window' with the cost of bringing the newly included

// If the total increments exceed k, shrink the window from the left.

// Calculate max frequency by finding the maximum window size till now.

Initialize two pointers for sliding window and the variables needed.

cumulative_diff += (nums[right] - nums[right - 1]) * (right - left)

If the cumulative differences exceed k, shrink the window from the left.

Increase the total difference by the difference between the current right element

and its predecessor multiplied by the number of elements in the current window.

Iterate with right pointer over the list to expand the window.

cumulative_diff -= nums[right] - nums[left]

The class and method could be tested with a simple test case as follows:

Move the right pointer to the next element in the list.

Update the maximum frequency using the current window size.

Return the maximum frequency achievable with total operations $\leq k$.

left, right, n = 0, 1, len(nums)

max_freq, cumulative_diff = 1, 0

while cumulative diff > k:

max_freq = max(max_freq, right - left)

print(sol.maxFrequency([1,2,4], 5)) # Expected output: 3

left += 1

right += 1

return max_freq

Time and Space Complexity

since n log n dominates n for large n.

sol = Solution()

Time Complexity

while right < n:</pre>

window += 1LL * (nums[right] - nums[right - 1]) * (right - left);

// Beginning with the second element as there's nothing to the left of the first.

std::sort(nums.begin(), nums.end());

// ans will hold the maximum frequency.

// Sort the given vector.

int n = nums.size();

int maxFrequency = 1;

long long window = 0;

while (window > k) {

left++;

max_freq = max(max_freq, right - left)

Sort the input list in non-decreasing order. nums.sort() # Initialize two pointers for sliding window and the variables needed. left, right, n = 0, 1, len(nums)

Increase the total difference by the difference between the current right element

and its predecessor multiplied by the number of elements in the current window.

cumulative_diff -= nums[right] - nums[left] left += 1 # Move the right pointer to the next element in the list. right += 1

Update the maximum frequency using the current window size.

Return the maximum frequency achievable with total operations $\leq k$.

// Function to find the maximum frequency of an element after performing operations

int operationsSum = 0; // This will hold the sum of operations used at any point

// Calculate the max frequency based on the window size and update it if necessary

// Calculate the total operations done to make all elements from 'left' to 'right' equal to nums[right]

int n = nums.length; // Store the length of the array for iteration

// 'right' at 1. since we'll start calculating from the second element

operationsSum += (nums[right] - nums[right - 1]) * (right - left);

// If the total operations exceed k, shrink the window from the left

Iterate with right pointer over the list to expand the window.

cumulative_diff += (nums[right] - nums[right - 1]) * (right - left)

If the cumulative differences exceed k, shrink the window from the left.

print(sol.maxFrequency([1,2,4], 5)) # Expected output: 3 Java

// Sort the array to group similar elements together

for (int left = 0, right = 1; right < n; ++right) {</pre>

The class and method could be tested with a simple test case as follows:

// Return the maximum frequency return maxFreq;

operationsSum -= (nums[right] - nums[left++]);

maxFreq = Math.max(maxFreq, right - left + 1);

C++

public:

#include <vector>

class Solution {

#include <algorithm>

```
maxFrequency = std::max(maxFrequency, right - left + 1);
        // Return the maximum frequency found.
        return maxFrequency;
};
TypeScript
function maxFrequency(nums: number[], k: number): number {
    // Sort the array
    nums.sort((a, b) => a - b);
    // Initialize the answer with a single element frequency
    let maxFrequency = 1;
    // Initialize a variable to keep track of the total sum that needs to be added to make all elements equal during a window slide
    let windowSum = 0;
    // Get the count of elements in the array
    let n = nums.length;
    // Initialize two pointers for our sliding window technique
    for (let left = 0, right = 1; right < n; right++) {</pre>
        // Update the window sum by adding the cost of making the right-most element equal to the current right-most element
        windowSum += (nums[right] - nums[right - 1]) * (right - left);
        // Shrink the window from the left if the cost exceeds the value of k
        while (windowSum > k) {
            windowSum -= nums[right] - nums[left];
            left++;
        // Update the maximum frequency with the current window size if it's greater
        maxFrequency = Math.max(maxFrequency, right - left + 1);
    // Return the maximum frequency achieved
    return maxFrequency;
from typing import List
class Solution:
    def maxFrequency(self, nums: List[int], k: int) -> int:
        # Sort the input list in non-decreasing order.
        nums.sort()
```

The provided code first sorts the nums list, and then uses a two-pointer approach to iterate over the list, adjusting the window of elements that can be made equal by applying up to k operations. • Sorting the list: Sorting takes O(n log n) time, where n is the number of elements in nums.

Space Complexity

one direction and can move at most n times, so it also contributes to O(n) operations. Combining the sort and the two-pointer iteration results in a time complexity of 0(n log n + n), which simplifies to 0(n log n)

• Two-pointer window iteration: The while loop iterates over the sorted list, adjusting the window of elements by incrementing the right pointer

and potentially the left pointer. The right pointer makes only one pass over the list, resulting in O(n) operations. The left pointer moves only in

Therefore, the overall time complexity of the code is $0(n \log n)$.

The space complexity of the code depends on the space used to sort the list and the additional variables used for the algorithm. • Sorting the list: The sort operation on the list is in-place; however, depending on the sorting algorithm implemented in Python (Timsort), the sorting can take up to O(n) space in the worst case.

• Additional variables: The code uses a few additional variables (1, r, n, ans, window). These require constant space (0(1)). Therefore, considering both the sorting and the additional variables, the space complexity of the code is 0(n) in the worst case

due to the sorting operation's space requirement.