1906. Minimum Absolute Difference Queries

Medium Array Hash Table

# **Problem Description**

index 1 to r inclusive. The minimum absolute difference for a subarray is determined by finding the smallest absolute value of the difference between any

In this LeetCode problem, we're required to find the minimum absolute difference of subarrays within a given integer array nums. A

subarray is specified through a range given in queries, where each query is in the form [I, r], meaning the subarray ranges from the

**Leetcode Link** 

two distinct elements within that subarray. It's important to note two key conditions:

- 1. We can only compare different elements, so the same elements should be disregarded when calculating the differences (e.g., | 5 - 5], which equals 0, should not be considered). 2. If all elements within the subarray are the same, the minimum absolute difference is -1 as there is no pair of different numbers to
- The task is to return an array of integers where each element is the result of computing the minimum absolute difference for the corresponding query.

Intuition To solve this problem, an efficient approach is to use prefix sums. Prefix sums allow us to quickly calculate the sum of a range of

numbers in an array, which is a useful strategy when you need to make multiple range queries.

particular query would be -1, indicating that all elements were the same in the subarray.

track of the count of each number up to a certain index.

two different numbers), and thus t is set to -1.

Let's walk through a small example to illustrate the solution approach:

for the subarrays specified by each pair of indices 1 and r in queries.

indexing). Since nums has 4 elements, pre\_sum will be a 5×101 matrix as shown below:

[0, 1, 0, 1, 0, 0, 0, 0, 0, ..., 0] // nums[1] (3) has appeared once up to index 1

[0, 1, 0, 1, 1, 0, 0, 0, 0, ..., 0] // nums[2] (4) has appeared once up to index 2

6 [0, 1, 0, 1, 1, 0, 0, 0, 1, ..., 0] // nums[3] (8) has appeared once up to index 3

For the query [0, 1], the relevant subarray is [1,3]. Looking at the pre\_sum matrix:

1 pre\_sum[1+1][1] - pre\_sum[0][1] = 1 - 0 = 1, so 1 appears in the subarray.

2 pre\_sum[1+1][3] - pre\_sum[0][3] = 1 - 0 = 1, so '3' appears in the subarray.

### Since we need to find the minimum absolute difference and not the sum, we modify the prefix sums approach to count the occurrences of each element in a range. This will help us to identify which elements are present in each subarray without having to

compare.

 $pre_sum[i][j]$  represents the number of times the number j has appeared in nums up to the i-th index. Once we have this prefix sum matrix built, we can then handle the queries. For each query [I, r], we look at the prefix sum matrix to

traverse the subarray repeatedly for every query. We can process the nums array to create a prefix sum matrix, where each entry

find which numbers are present in the subarray from I to r. We traverse the range of possible values, and when we find a number present in the subarray, we compare it with the last number found to calculate the difference and keep track of the smallest difference encountered. If at least two different numbers are found in the subarray, we would have a valid minimum difference. If not, the result for that

The implemented solution follows a few steps, using the concept of prefix sums and last seen elements: 1. Initialization: The first step involves initializing a prefix sum matrix pre\_sum of size (m+1) x 101, where m is the number of elements in nums, and 101 is used because the numbers are constrained between 1 to 100. The pre\_sum matrix is used to keep

### • The outer loop goes from 1 to m (inclusive), iterating over the input array nums. • The inner loop goes from 1 to 100, iterating over all possible number values within the constraints.

returned.

**Solution Approach** 

At each iteration, the code checks if the number j is the same as the i-th element of nums. If it matches (nums[i - 1] == j), the code increments the count for that number at the current prefix index. This populates pre\_sum such that pre\_sum[i][j]

result) and last to -1 as a placeholder for the previously encountered number in the subarray.

2. Building the Prefix Sum Matrix: The code populates the pre\_sum matrix using two nested loops:

- represents the number of times the number j has appeared from the start of nums up to index i-1. 3. Handling Queries: For each query defined by [1, r], the code translates it to a range in pre\_sum as [left, right+1] to use onebased indexing. It then initializes variables t to hold the maximum possible value (infinity at the start as a placeholder for no
- subarray. We then calculate the difference with the last seen number and update t with the minimum difference. last is updated to the current number j to be used in the next iterations.

5. Handling impossible cases: After processing the differences, if t remains as infinity (the code uses inf which is a constant

6. Storing the Result: The minimum difference t calculated for each query is then appended to the result list ans.

representing an infinite value), it means that no valid minimum absolute difference could be calculated (there were not at least

to calculate their difference. If pre\_sum[right][j] - pre\_sum[left][j] > 0, it means that the number j is present in the

4. Processing each Query: For each query, the code iterates from 1 to 100 to check which numbers are present in the subarray and

Through these steps, the algorithm efficiently computes the minimum absolute difference for multiple subarray queries in an array using prefix sums, resulting in reduced time complexity compared to a brute-force approach. **Example Walkthrough** 

7. Returning the Result: Finally, after processing all queries, the list ans containing all the minimum differences for each query is

1 pre\_sum = [[0]\*101 for i in range(5)] // 5 rows, 101 columns filled with 0

Step 1: Initialize the prefix sum matrix pre\_sum which has the size (n+1) x 101 (where n is the length of nums, to handle zero

Step 2: Build the prefix sum matrix by iterating over nums and updating the counts for each number between 1 and 100.

Consider the integer array nums = [1,3,4,8] and queries = [[0,1],[1,2],[2,3]]. We aim to find the minimum absolute difference

After building, let's say the matrix looks like this (trimmed for brevity): 2 [0, ..., 0] // Base row for ease of calculation, all zeroes [0, 1, 0, 0, 0, 0, 0, 0, 0, ..., 0] // nums[0] (1) has appeared once up to index 0

Since both 1 and 3 are present, we calculate the absolute difference: |1 - 3| = 2. So for the first query, the minimum absolute difference is 2.

4. So for the third query, the minimum absolute difference is 4.

# Get the lengths of the input list and the queries

 $prefix_sum = [[0] * 101 for _ in range(num_count + 1)]$ 

# Prepare a list to store the answers to each query

# Define the range for the current query

# Initialize the last seen number variable

if last\_seen != -1:

last\_seen = j

if min diff == inf:

 $min_diff = -1$ 

results.append(min\_diff)

left, right = queries[i][0], queries[i][1] + 1

# Initialize a prefix sum array with zeros

Step 3: Handle the queries [0, 1], [1, 2], and [2, 3].

For the query [1, 2], the relevant subarray is [3,4]. Using the similar approach as above, we find that 3 and 4 are present, and the minimum absolute difference is |3 - 4| = 1. So for the second query, the minimum absolute difference is 1. For the query [2, 3], the relevant subarray is [4,8]. Similarly, 4 and 8 are present, and the minimum absolute difference is |4 - 8| =

Step 4: After processing each query, we compile the results into the answer list ans = [2, 1, 4].

# This array will store the counts of each number from 1 to 100 up to index i

# Check if the current num is equal to the current value j

prefix\_sum[i][j] = prefix\_sum[i - 1][j] + count\_increment

# If this is not the first number we've seen, update the min\_diff

# If min\_diff is still infinity, it means no numbers are present, so set the result to -1

count\_increment = 1 if nums[i - 1] == j else 0

# Initialize the minimum difference to positive infinity

min\_diff = min(min\_diff, j - last\_seen)

# Update last\_seen with the current number

# Append the result of the current query to the results list

return answer; // Return the array of answers for all queries.

vector<int> minDifference(vector<int>& nums, vector<vector<int>>& queries) {

// Calculate the prefix sum for number j up to index i

int right = queries[i][1] + 1; // Add 1 to right to make it exclusive

// If the current number equals j, set t to 1; otherwise, set t to 0

// Get the size of the nums vector and the number of queries

int numsSize = nums.size(), queriesSize = queries.size();

// Initialize a 2D prefix sum array

// Populate the prefix sum array

vector<int> answer(queriesSize);

int left = queries[i][0];

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

// Process each query

int minDiff = 101;

int lastSeen = -1;

int prefixSum[numsSize + 1][101] = {0};

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

for (int j = 1;  $j \le 100$ ; ++j) {

int t = (nums[i - 1] == j) ? 1 : 0;

// Variable to store the minimum difference

// Variable to store the last seen element

// Loop over every possible integer

if (lastSeen != -1) {

for (int j = 1;  $j \le 100$ ; ++j) {

lastSeen = j;

prefixSum[i][j] = prefixSum[i - 1][j] + t;

// Initialize the answer vector to store the minimum differences

// Check if integer j is between left and right

if (prefixSum[right][j] - prefixSum[left][j] > 0) {

minDiff = min(minDiff, j - lastSeen);

// Update lastSeen with the current integer

// Update minDiff if there was a last seen integer

1 from math import inf class Solution: def minDifference(self, nums: List[int], queries: List[List[int]]) -> List[int]:

By utilizing the prefix sum approach, we avoid recalculating the presence of each number for every subarray query, which saves

computation time. The final answer gives the minimum absolute difference for each of the subarrays specified by queries.

12 13 # Calculate prefix sums 14 for i in range(1, num\_count + 1): 15 for j in range(1, 101): 16

#### 32 33 # Loop through the range of possible values 1 to 100 34 for j in range(1, 101): 35 # If the number j is present in the current segment (between left and right) if prefix\_sum[right][j] - prefix\_sum[left][j] > 0: 36

Step 5: Return ans.

**Python Solution** 

6

8

9

10

11

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

37

39

40

41

42

43

44

45

46

47

48

49

39

40

41

42

43

C++ Solution

#include <vector>

class Solution {

public:

9

10

11

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

using namespace std;

#include <climits> // For INT\_MAX

num\_count = len(nums)

results = []

# Process each query

min\_diff = inf

 $last_seen = -1$ 

for i in range(query\_count):

query\_count = len(queries)

```
# Return the list of results
 50
 51
             return results
 52
Java Solution
    class Solution {
         public int[] minDifference(int[] nums, int[][] queries) {
             int numsLength = nums.length, queriesLength = queries.length;
             // Initialize an array to store the prefix sums for each number from 1 to 100.
  5
             int[][] prefixSums = new int[numsLength + 1][101];
  6
             // Fill the prefixSums array with the counts of each number up to the current index.
  8
             for (int i = 1; i <= numsLength; ++i) {</pre>
                 for (int num = 1; num <= 100; ++num) {</pre>
  9
                     int presence = nums[i - 1] == num ? 1 : 0;
 10
 11
                     prefixSums[i][num] = prefixSums[i - 1][num] + presence;
 12
 13
 14
 15
             // Initialize an array to hold the answers to each query.
 16
             int[] answer = new int[queriesLength];
             for (int i = 0; i < queriesLength; ++i) {</pre>
 17
 18
                 int startRange = queries[i][0], endRange = queries[i][1] + 1;
 19
                 int minDifference = Integer.MAX_VALUE;
                 int lastNum = -1; // Store the last seen number to calculate the difference.
 20
 21
 22
                 // Iterate from 1 through 100 to find the minimum difference between successive numbers.
 23
                 for (int j = 1; j \le 100; ++j) {
 24
                     // Check if the number j is present in the subrange [startRange, endRange)
                     if (prefixSums[endRange][j] > prefixSums[startRange][j]) {
 25
 26
                         // Only update the minDifference if this is not the first number found.
                         if (lastNum != -1) {
 27
                             minDifference = Math.min(minDifference, j - lastNum);
 28
 29
 30
                         lastNum = j; // Update the lastNum to the current number.
 31
 32
 33
 34
                 // If no numbers were found, set the minimum difference as -1.
 35
                 if (minDifference == Integer.MAX_VALUE) {
 36
                     minDifference = -1;
 37
                 answer[i] = minDifference; // Set the answer for the current query.
 38
```

### 53 54 55 56

```
48
                 // If minDiff remains 101, it means no two different numbers were found so set minDiff to -1
 49
 50
                 if (minDiff == 101) {
 51
                     minDiff = -1;
 52
                 // Store the result in the answer vector
                 answer[i] = minDiff;
 57
 58
             // Return the calculated minimum differences for each query
 59
             return answer;
 60
 61 };
 62
Typescript Solution
    function minDifference(nums: number[], queries: number[][]): number[] {
         const numLength = nums.length; // Total number of elements in nums array
         const queryLength = queries.length; // Total number of queries
         const maxValue = 100;
                                            // Maximum value that nums can have
  5
  6
         // Prefix sum array to store the frequencies of each number up to index i
         const prefixSum: number[][] = [new Array(maxValue + 1).fill(0)];
  8
         // Populate the prefix sum array
  9
         for (let i = 0; i < numLength; ++i) {
 10
             let currentNumber = nums[i];
 11
 12
             // Copy the previous frequency array and update the current number frequency
 13
             prefixSum.push(prefixSum[i].slice());
             prefixSum[i + 1][currentNumber] += 1;
 14
 15
 16
 17
         const results = []; // Array to store the minimum differences for each query
 18
 19
         // Iterate through each query to calculate minimum differences
         for (let [left, right] of queries) {
 20
             let lastOccurrence = -1; // Last seen number (initialize to -1)
 21
 22
             let minDiff = Infinity; // Minimum difference found (initialize to infinity)
 23
 24
             // Iterate through all possible values in the range [1, maxValue]
             for (let value = 1; value <= maxValue; ++value) {</pre>
 25
                 // Check if the number 'value' is present between 'left' and 'right' indices
 26
                 if (prefixSum[left][value] < prefixSum[right + 1][value]) {</pre>
 27
                     // If 'lastOccurrence' is not -1, it means that we have already found a previous number
 28
 29
                     if (lastOccurrence !== -1) {
 30
                         // Update minDiff if the current difference is smaller than the previously found
                         minDiff = Math.min(minDiff, value - lastOccurrence);
 31
 32
 33
                     // Update the lastOccurrence to the current value
                     lastOccurrence = value;
 34
 35
 36
 37
 38
             // If no number is found between 'left' and 'right', return -1, else the minimum difference
 39
             results.push(minDiff === Infinity ? -1 : minDiff);
 40
 41
 42
         return results; // Return result array with minimum differences for each query
 43 }
 44
```

# and then processing each query to find the minimum difference between consecutive numbers present within the query range. 1. Calculating Prefix Sums: The code constructs a 2-D array pre\_sum of size (m+1) x 101, where each entry pre\_sum[i][j]

constant factor and can be omitted in Big O notation.

Time and Space Complexity

## elements in nums. 2. Processing Queries: For each of the n queries, the code looks for the smallest difference between consecutive numbers that are present in the specified range. Since the numbers range from 1 to 100, we iterate 100 times for each query to determine this.

**Space Complexity** 

**Time Complexity** 

Therefore, the complexity for processing all queries is 0(n \* 100). Combining both parts, the total time complexity is 0(m \* 100 + n \* 100), which simplifies to 0(m + n) considering that 100 is a

The given code involves two main parts: calculating the prefix sums for each number in the range [1, 100] for each index in nums,

represents the number of occurrences of the number j up to the i-th position in nums. Iterating through all values of nums and all

numbers [1, 100] to populate this array has a complexity of 0(m \* 100) since we iterate through 100 numbers for each of the m

- 1. Prefix Sum Array (pre\_sum): The size of the pre\_sum array is (m+1) x 101, so the space complexity contributed by this array is 0(m \* 101), which is 0(m) as 101 is a constant factor.
- O(n) to space complexity. Thus, the total space complexity of the code is 0(m + n) combining the prefix sum array and the output array.

2. Output Array (ans): The size of the output array is n, correlating directly with the number of queries. Therefore, it contributes

The space complexity is determined by the storage required for the pre\_sum array and the output array ans: