2155. All Divisions With the Highest Score of a Binary Array

Medium <u>Array</u>

Problem Description You are given a binary array, nums, that contains only 0s and 1s. The task is to find all indices at which you can divide this array

into two subarrays, nums_left and nums_right, such that the division score is maximized. The division score is defined as the count of 0s in nums_left plus the count of 1s in nums_right.

The array can be divided at any index i, which creates nums_left with elements from the start of the array up to index i=1, and nums_right.

The array can be divided at any index i, which creates nums_left with elements from the start of the array up to index i-1, and nums_right with elements from index i to the end. Note that if i is 0, then nums_left is empty and nums_right contains all the elements, while if i is the length of the array (n), then nums_left contains all the elements and nums_right is empty.

Intuition

The goal is to return indices that yield the highest division score. The indices can be returned in any order.

beginning of the array towards the end, and right starts off as the total count of 1s in the entire array.

To solve this problem, we need to understand that at each potential division index, the score is affected by the number of zeros to the left and the number of ones in the array

every time.

Here's the approach to solve the problem:

1. Initialize two counters, left and right. left will hold the number of 0s that we've encountered so far as we iterate from the

beforehand, we can easily calculate the division score for each index without having to count the number of ones to the right

2. Initialize a variable mx that will hold the maximum division score that we've found so far. Initially, this will be equal to right

achieved. We return this list.

because before any division, all 1s would be in nums_right and this is the highest possible score at the start.

3. Start iterating through the array. With each element, update left and right accordingly. If you encounter a 0, increment left.

If you encounter a 1, decrement right. This is because a zero would now be contributing to the division score on the left, and

- a one would no longer contribute to the score on the right after passing the division point.

 4. At each index, calculate the current division score, t, as the sum of left and right. If it's equal to mx, append the current
- update mx with t and reset ans to a new list containing just the current index + 1. This keeps track of all indices where we have encountered the maximum score.

 5. After we have finished iterating through the array, we will have a list of all indices at which the maximum division score can be

index + 1 to the list of indices with maximum score, ans. If the current score is greater than the max score found so far (mx),

The implementation follows a simple but efficient algorithm that ensures we only need a single pass through the input array, which is a key characteristic of an O(n) time complexity solution. The approach heavily relies on the usage of two variables to

represent the current number of zeros and ones relevant to the potential division at any index in the array. Here's the detailed

2. Initialize a counter right to the sum of the elements in the nums array (since the array contains only 0s and 1s, this sum equals the number of 1s

achieved.

implementation approach:

previous indices are no longer valid.

1. Initialize left to 0 and right to 3 since there are three 1s in the array.

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

current_score = score_left + score_right

best_indices.append(index + 1)

// Helper method to calculate the sum of ones in the array

private int sum(int[] nums) {

for (int num : nums) {

sum += num;

// Sum all elements in the array

vector<int> maxScoreIndices(vector<int>& nums) {

for (int i = 0; i < nums.size(); ++i) {</pre>

if (scoreAtEachIndex[i] === maxScore) {

// Return the array of indices with maximum score

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

Set the maximum score as the initial score of right partition

If the number is 0, it contributes to the left partition's score

If the number is 1, it contributes to the right partition's score

Current total score is the sum of the scores from both partitions

Found a new max score, update max score and reset best indices

Initialize the score for the left partition (initially 0) and the right partition

maxIndices.push(i);

return maxIndices;

score_left = 0

score_right = sum(nums)

max_score = score_right

best_indices = [0]

if num == 0:

elif num == 1:

The initial best index is 0

Iterate through the list of numbers

for index, num in enumerate(nums):

score_left += 1

score_right -= 1

elif max_score < current_score:</pre>

max_score = current_score

best_indices = [index + 1]

Return the list of indices that yield the maximum score

from typing import List

class Solution:

if (nums[i] == 0) {

} else {

int leftZeroes = 0; // Counter for the number of zeroes on the left

resultIndices.push_back(0); // 0 is a potential max score index

int maxScore = rightOnes; // Maximum score initialized with the sum of ones

// Iterate through the nums array to find where the maximum score occurs

vector<int> resultIndices; // Vector to store the indices where max score occurs

++leftZeroes; // Increment leftZeroes for each zero encountered

--rightOnes; // Decrement rightOnes for each one encountered

int currentScore = leftZeroes + rightOnes; // Calculate current score

int rightOnes = accumulate(nums.begin(), nums.end(), 0); // Counter for the number of ones on the right

int sum = 0;

return sum;

Set the maximum score as the initial score of right partition

If the number is 0, it contributes to the left partition's score

Current total score is the sum of the scores from both partitions

This index yields a score equal to the current max score

Check if the current score equals the max score found so far

Solution Approach

in the array).
3. Set the maximum score mx initially equal to right because, at the start (when the division point is at index 0 and nums_left is empty), all 1s are in nums_right.
4. Create an array ans with a single element 0 because initially, the highest score can be achieved when the division point is at index 0.
5. Iterate over the elements in the nums array using a loop. For each element at index i, do the following:

If the current element num is 0, increment left because we've found another 0 that contributes to the score when it's in nums_left.

If the current element is 1, decrement right as this 1 no longer contributes to the score in nums_right after moving past index i.
 Calculate the temporary score t as the sum of left and right, which represents the current division score at index i.
 Compare the temporary score t with the maximum score mx:

1. Initialize a counter left to 0. This will track the number of 0s to the left of the current index as we iterate through the array.

If t is equal to mx, append i + 1 to ans. The +1 is necessary because the division occurs after the current index, so the next index is the potential division point.
 If t is greater than mx, update mx to t and set ans to a new list containing only i + 1, since we have found a new highest score and the

6. Continue this process until the end of the array. After the loop ends, the ans array contains all indices where the maximum division score is

7. Finally, return the ans array.

The use of a single loop and constant-time update operations ensures that the overall time complexity remains linear, which is

and an output array, avoiding any additional data structures that could increase the space usage.

2. Set mx to 3 because at the index 0, nums_right would contain all the 1s, giving the maximum possible score initially.

crucial for handling large arrays efficiently. The space complexity is also kept low since we are only using a handful of variables

- Example Walkthrough

 Let's walk through an example to illustrate the solution approach with a binary array nums.
- Suppose nums = [0, 1, 0, 1, 1]. We want to find all indices where we can divide this array to maximize the division score.

 Following the solution approach:

Let's iterate through each element and apply the steps: • At index 0, nums [0] is 0. Increment left to 1 (now left = 1, right = 3), the score here is 1 + 3 = 4. Since 4 is greater than mx, we update mx to 4

score is less.

Python

3. Create an array ans with a single element 0.

and set ans to [1] (next index after division).

4. Start iterating over the array nums.

• At index 1, nums[1] is 1. Decrement right to 2 (now left = 1, right = 2), the score now is 1 + 2 = 3. The score 3 is not greater than mx (4), so we continue without updating mx or ans.

dividing the array at indices 1 or 3 gives us both subarrays that maximize the division score.

Initialize the score for the left partition (initially 0) and the right partition

(becoming [1, 3]).
At index 3, nums [3] is 1. Decrement right to 1 (now left = 2, right = 1), and the score here is 2 + 1 = 3. No change to mx or ans since the

The iteration is complete, and the ans array [1, 3] contains all the indices that give the maximum division score of 4. Thus,

• At index 4, nums [4] is 1. Decrement right to 0 (now left = 2, right = 0), and the score is 2 + 0 = 2, which is again less than mx.

• At index 2, nums [2] is 0. Increment left to 2 (now left = 2, right = 2), the new score is 2 + 2 = 4. This equals the mx, thus we append 3 to ans

- Solution Implementation
 - from typing import List
 class Solution:

if num == 0: score_left += 1 # If the number is 1, it contributes to the right partition's score elif num == 1:

best_indices = [0]

score_left = 0

score_right = sum(nums)

max_score = score_right

The initial best index is 0

Iterate through the list of numbers

for index, num in enumerate(nums):

score_right -= 1

if max_score == current_score:

elif max_score < current_score:</pre>

```
# Found a new max score, update max score and reset best indices
                max_score = current_score
                best_indices = [index + 1]
       # Return the list of indices that yield the maximum score
        return best_indices
Java
class Solution {
   // Method to find all indices where we can split the input array nums such that the sum of zeros to the left and ones to the
   public List<Integer> maxScoreIndices(int[] nums) {
       // Initialize count of zeros to the left and ones to the right of the index
       int zerosCount = 0, onesCount = sum(nums);
       // Initialize max score with the score if we split before the first index (all ones to the right)
       int maxScore = onesCount;
       // List to store all indices that provide maximum score
       List<Integer> resultIndices = new ArrayList<>();
       // Add index 0 as a valid split (no elements to the left)
        resultIndices.add(0);
       // Iterate over the array to find all valid split indices
        for (int i = 0; i < nums.length; ++i) {</pre>
           // If current element is 0, increase zeros count
           if (nums[i] == 0) {
               ++zerosCount;
           } else {
                // If current element is 1, decrease ones count
                --onesCount;
           // Current score is the sum of zeros to the left and ones to the right
           int currentScore = zerosCount + onesCount;
           // If current score equals the max score, add index to result
           if (maxScore == currentScore) {
                resultIndices.add(i + 1);
            } else if (maxScore < currentScore) {</pre>
                // If current score exceeds max score, update max score and clear previous indices
                maxScore = currentScore;
                resultIndices.clear();
                resultIndices.add(i + 1);
       // Return the list of all indices that provide maximum score
       return resultIndices;
```

C++

public:

class Solution {

```
// Check if the current score matches the maximum score found so far
            if (maxScore == currentScore) {
                // If it matches, add the current index + 1 to the result
                resultIndices.push_back(i + 1);
            } else if (maxScore < currentScore) {</pre>
                // If the current score is greater, update the maxScore
                maxScore = currentScore;
                // Clear the resultIndices vector and start again with the current index + 1
                resultIndices.clear();
                resultIndices.push_back(i + 1);
       return resultIndices; // Return the vector with all indices where the max score occurs
};
TypeScript
// Calculates the indices where the score is maximized when splitting the array 'nums' into two parts
function maxScoreIndices(nums: number[]): number[] {
    // Calculate the length of the numbers array
    const length = nums.length;
    // Compute the total count of '1's in the array
    const totalOnes = nums.reduce((accumulator, current) => accumulator + current, 0);
    let zerosToLeft = 0; // Number of `0`s to the left of the current index
    let onesToRight = totalOnes; // Number of `1`s to the right of the current index
    // Initialize an array to store the combined score at each index
    let scoreAtEachIndex: Array<number> = [totalOnes]; // Include the score when split index is 0
    // Compute the score for each index in the input array
    for (const num of nums) {
       // Update corresponding scores based on the current number
       if (num === 0) {
            zerosToLeft++; // Increment as we found a `0`
        } else {
            onesToRight--; // Decrement as we found a `1`
       // Record the current score
       scoreAtEachIndex.push(zerosToLeft + onesToRight);
    // Determine the maximum score out of all scores
    const maxScore = Math.max(...scoreAtEachIndex);
    let maxIndices: Array<number> = []; // Array to store indices with max score
    // Collect all indices where the score equals the maximum score
    for (let i = 0; i <= length; i++) {</pre>
```

```
current_score = score_left + score_right

# Check if the current score equals the max score found so far
if max_score == current_score:
    # This index yields a score equal to the current max score
    best_indices.append(index + 1)
```

return best_indices

Time and Space Complexity

The time complexity of the code is O(n), where n is the length of the input list nums. This is because the code iterates once over all the elements of the list, performing a constant amount of work for each element (incrementing left if it's a 0, decrementing right if it's a 1, and comparing and possibly updating the maximum score).

The space complexity of the code is 0(k), where k is the number of indices at which the maximum score is achieved. In the

worst-case scenario, every index could be a part of the answer, in which case the space complexity would be O(n). However, under normal circumstances where not every index is a solution, k is typically much less than n. Fundamental to the space complexity is the list ans, which stores the indices. Besides that, the algorithm uses only a constant amount of additional space for variables like left, right, mx, and t.