<https://leetcode.ca/2021-10-15-2031-Count-Subarrays-With-More-Ones-Than-Zeros/>

You are given a binary array nums containing only the integers 0 and 1. Return *the number of* ***subarrays*** *in nums that have* ***more***1'*s than* 0*'s. Since the answer may be very large, return it* ***modulo***10^9 + 7.

A **subarray** is a contiguous sequence of elements within an array.

**Example 1:**

Input: nums = [0,1,1,0,1]

Output: 9

Explanation:

The subarrays of size 1 that have more ones than zeros are: [1], [1], [1]

The subarrays of size 2 that have more ones than zeros are: [1,1]

The subarrays of size 3 that have more ones than zeros are: [0,1,1], [1,1,0], [1,0,1]

The subarrays of size 4 that have more ones than zeros are: [1,1,0,1]

The subarrays of size 5 that have more ones than zeros are: [0,1,1,0,1]

**Example 2:**

Input: nums = [0]

Output: 0

Explanation:

No subarrays have more ones than zeros.

**Example 3:**

Input: nums = [1]

Output: 1

Explanation:

The subarrays of size 1 that have more ones than zeros are: [1]

**Constraints:**

* 1 <= nums.length <= 10^5
* 0 <= nums[i] <= 1

**Attempt 1: 2023-11-16**

**Solution 1: Hash Table  + Presum (120 min)**

class Solution {

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length;

for(int i = 0; i < n; i++) {

if(nums[i] == 0) {

nums[i] = -1;

}

}

int[] preSum = new int[n + 1];

for(int i = 1; i <= n; i++) {

preSum[i] = preSum[i - 1] + nums[i - 1];

}

int result = 0;

int mod = 1000000007;

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

for(int i = 0; i <= n; i++) {

freq.put(preSum[i], freq.getOrDefault(preSum[i], 0) + 1);

for(Map.Entry<Integer, Integer> e : freq.entrySet()) {

if(e.getKey() < preSum[i]) {

result = (result + e.getValue()) % mod;

}

}

}

return result % mod;

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums = {0,1,1,0,1};

int result = so.subarraysWithMoreZerosThanOnes(nums);

System.out.println(result);

}

}

Time Complexity: O(N^2)

Space Complexity: O(N)

**最重要的一个处理：把原数组中所有的0都变成-1，这个处理的意义在于在计算presum的时候所有原来的0变成-1后可以与所有存在的1在数值上一一对应的相互抵消，这样就变成了在一个nums[i, j]的区间上如果1的数量大于0的数量，抵消后的加权和大于0，同理，1的数量等于0的数量，抵消后的加权和等于0，1的数量小于0的数量，抵消后的加权和小于0。而不采用这种处理会导致presum[i]只记录了区间段[0, i]内1的个数，而对于0的个数毫无记录，因为累计加权中没有1和-1一一对应的相互抵消，只有1的不断累积，而0对于累加和没有贡献，所以无论0有多少个都无法反映在presum中，下面来看一下为何一定需要把0改变成-1后再获得presum的效果和不做改变直接获得presum的错误效果：**

**把0改变成-1后再获得presum的效果：**

(nums) 0,1,1,0,1

presum compare

0 0 1 2 2 3 wrong

0 -1 0 1 0 1 correct

Step by step for correct one:

i = 0

freq = {{0 -> 1}}

e.getKey() < preSum[i](=0)

result no change

i = 1

freq = {{0 -> 1}, {-1 -> 1}}

e.getKey() < preSum[i](=-1)

result no change

i = 2

freq = {{0 -> 2}, {-1 -> 1}}

e.getKey() < preSum[i](=0)

result += 1 => 1

i = 3

freq = {{0 -> 2}, {-1 -> 1}, {1 -> 1}}

e.getKey() < preSum[i](=1)

result += 2 => 3

result += 1 => 4

i = 4

freq = {{0 -> 3}, {-1 -> 1}, {1 -> 1}}

e.getKey() < preSum[i](=0)

result += 1 => 5

i = 5

freq = {{0 -> 3}, {-1 -> 1}, {1 -> 2}}

e.getKey() < preSum[i](=1)

result += 3 => 8

result += 1 => 9

**不做改变直接获得presum的错误效果：**

(nums) 0,1,1,0,1

presum compare

0 0 1 2 2 3 wrong

0 -1 0 1 0 1 correct

Step by step for wrong one:

i = 0

freq = {{0 -> 1}}

e.getKey() < preSum[i](=0)

result no change

i = 1

freq = {{0 -> 2}}

e.getKey() < preSum[i](=0)

result no change

i = 2

freq = {{0 -> 2}, {1 -> 1}}

e.getKey() < preSum[i](=1)

result += 2 => 2

i = 3

freq = {{0 -> 2}, {1 -> 1}, {2 -> 1}}

e.getKey() < preSum[i](=2)

result += 2 => 4

result += 1 => 5

i = 4

freq = {{0 -> 2}, {1 -> 1}, {2 -> 2}}

e.getKey() < preSum[i](=2)

result += 2 => 7

result += 1 => 8

i = 5

freq = {{0 -> 2}, {1 -> 1}, {2 -> 2}, {3 -> 1}}

e.getKey() < preSum[i](=3)

result += 2 => 10

result += 1 => 11

result += 2 => 13

**其次一个重要的处理就是不单纯依靠presum[i]来判断[0, i]区间内1的个数大于0，而是基于j < i并且presum[j] < presum[i]，即presum[j, i] = presum[i] - presum[i] > 0，则所代表的区间[j, i]上1的个数一定大于0。因为虽然代表区间[0, i]的加权和的presum[i] > 0时可以推导出在区间[0, i]上一定是1的数量大于0的，但无法保证其内部所有子区间也满足这个推定，比如nums=[0,1,1]，改变后[-1,1,1]，presum[2] = 1 > 0可以推定出1的个数大于0(两个1，一个-1即0)，但是其子区间[-1,1]，presum[1] = 0就不满足，1的个数等于0的个数(一个1，一个-1代表0)，所以改变策略，为了保证nums[j, i]区间的1的个数一定大于0的个数，我们需要保证presum[j, i]一定大于0，这也就有了获取当j < i时同时满足presum[j] < presum[i]的j的个数的缘由**

**最后一个重要的处理就是基于presum(这个presum本身基于把所有0改造为-1后建立的新nums)建立frequency hash table，这样就可以加快获得当j < i时同时满足presum[j] < presum[i]的j的个数的速度**

**Refer to**

<https://algomonster.medium.com/2031-count-subarrays-with-more-ones-than-zeros-d1f684a17361>

**Brute Force (Hash Table + Presum)**

Note that this problem is pretty tricky and definitely **not a medium** question. The categorization is likely wrong on leetcode. Don’t feel bad if you don’t get it the first time. It took me a whole night to figure it out.

Let’s first pretend 0s are actually -1s so that 1 and -1 cancels each other out and we just need to find subarrays with sum > 0.

On first look, this problem looks like a [sliding window](https://algo.monster/problems/two_pointers_intro) problem. We loop though each right pointer and expand the window and adjust the left pointer to the right if the window sum is ≤ 0. And the number of subarrays within the window is right — left + 1 which we add to the total count.

On further examination, we come to the realization that this approach is flawed because although the subarray of the window may have a sum greater than 0, the individual subarrays within the window may not necessarily have a sum greater than 0. For example, [0, 1, 1] has sum > 0 but [0, 1] does not.

Since the problem is about incremental sum changes, a natural thing to think about is [prefix sum](https://algo.monster/problems/subarray_sum). For each subarray ending at index i with a sum prefix\_sum[i], what we want to find is the number of prefix subarrays (subarray that starts from 0 and ending where j < i) whose sum is smaller than prefix\_sum[i] so that sum(nums[i:j]) > 0.

To calculate the count quickly, we can use a hashmap of sum: count of prefix subarrays to keep track of the frequency. Let’s call it the sum\_to\_count hashmap. And we can go through each element, use it as the last element of the subarray, track the running sum, and go through every pair in the hashmap and add the count to the result if a sum is smaller than the current sum.

class Solution:

# O(n^2) solution, does not pass all test cases

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

nums = [i if i else -1 for i in nums]

n = len(nums)

prefix\_sum = [0] \* (n + 1)

for i in range(1, n+1):

# sum of nums from nums[0] to nums[i - 1]

prefix\_sum[i] = prefix\_sum[i-1] + nums[i-1]

cnt = Counter()

res = 0

mod = 1000000007

for i in range(n+1):

cnt[prefix\_sum[i]] += 1

res += sum([v for k, v in cnt.items() if k < prefix\_sum[i]]) % mod

return res % mod

This passes the smaller test cases but fails the larger cases. And the problem is `sum([v for k, v in cnt.items() if k < prefix\_sum[i]]) % mod` takes O(n) time and the overall time complexity is O(n²). The constraints is n ≤ 10⁵ and O(n²) is too slow. As a rule of thumb, 10⁴ is the max for O(n²) solution that could pass all tests in most online coding platform ([a list of runtime to constraints mapping](https://algo.monster/problems/runtime_summary)).

**Solution 2: DP + Linear Scan (120 min)**

class Solution {

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length;

int mod = 1000000007;

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

// Empty array has sum 0

sumFreq.put(0, 1);

int result = 0;

int cur\_sum = 0;

// The count of subarrays up until index i — 1 whose sum = 0

int prev\_count\_0 = 0;

// The count of subarrays until index i — 1 whose sum > 1

int prev\_count\_1 = 0;

// The count of subarrays up until index i whose sum = 1

int cur\_count\_1 = 0;

for(int num : nums) {

if(num == 1) {

cur\_sum += 1;

} else {

cur\_sum -= 1;

}

if(num == 1) {

cur\_count\_1 = (prev\_count\_1 + prev\_count\_0 + 1) % mod;

} else {

// 减掉map中存储的cur\_sum的出现次数的逻辑在于，当num == 0时，我们预先的处理是cur\_sum -= 1，

// 为什么减1呢？因为假设当前新出现的num == 1，那么子数组加上这个1之前的加权和就是cur\_sum - 1,

// 然而当新出现的num不是1而是0，就会使得子数组加权和不再是cur\_sum，而是cur\_sum - 1，子数组是

// 无论如何无法通过加0达到cur\_sum的，这些加0后无法达到cur\_sum的子数组的个数就是在当前num == 0

// 时受影响的无法保持加权和>=1的子数组的个数

cur\_count\_1 = (prev\_count\_1 - sumFreq.getOrDefault(cur\_sum, 0) + mod) % mod;

}

prev\_count\_0 = sumFreq.getOrDefault(cur\_sum, 0);

prev\_count\_1 = cur\_count\_1;

sumFreq.put(cur\_sum, sumFreq.getOrDefault(cur\_sum, 0) + 1);

result = (result + cur\_count\_1) % mod;

}

return result;

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums = {0,1,1,0,1};

int result = so.subarraysWithMoreZerosThanOnes(nums);

System.out.println(result);

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algomonster.medium.com/2031-count-subarrays-with-more-ones-than-zeros-d1f684a17361>

So we need to optimize the inner loop to O(n). Another piece of information we haven’t used is: at index i, we already have the result from index i-1 and there are only two possibility for nums[i], 1 or 0, so we can build on the previous result and consider each case of 1 or 0.

Let’s loop through each element, keeping tracking of the running sum and two states for the previous step

* prev\_count\_0, the count of subarrays up until index i — 1 whose sum = 0
* prev\_count\_1, the count of subarrays until index i — 1 whose sum > 1

We also have two states for the current step

* current\_count\_0, the count of subarrays up until index i whose sum = 0
* current\_count\_1, the count of subarrays up until index i whose sum = 1

What we can see is

* if nums[i] == 1, since the current element contributes 1 to the sum, all previous subarrays with sum >1 still have sum > 1 with 1 added, so we add the count. Additionally, we can add 1 to all previous subarrays with sum = 0 and make it a qualifying subarray with sum = 1 > 0. The overall transition formula is current\_count\_1 = prev\_count\_1 + prev\_count\_0 + 1
* if nums[i] == 0, the current element contribute net negative 1 to the running sum. What this means is that out of all the previous subarrays whose sum > 1, some of the subarrays would now fail to qualify after adding the current element’s negative 1 contribution. And what is this count? Remember we add the current element to the running sum, and we know the count of that sum from sum\_to\_count[sum]. All the previous subarrays with sum\_to\_count[sum] as prefix with the current element added would have their sums go to 0 and disqualify. The formula is current\_count\_1 = current\_count\_1 + (previous\_count\_1 — sum\_to\_count[sum]).

At the end of each step, we move current\_count\_1 to previous\_count\_1 and current\_count\_0 to previous\_count\_0.

class Solution:

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

MOD = 1000000007

from collections import defaultdict

cur\_sum = 0

prev\_count\_0 = prev\_count\_1 = 0

sum\_to\_count = defaultdict(int)

sum\_to\_count[0] = 1 # empty array has sum 0

ans = 0

for num in nums:

if num == 1:

cur\_sum += 1

else:

cur\_sum -= 1

if num == 1:

cur\_count\_1 = (prev\_count\_1 + prev\_count\_0 + 1) % MOD

elif num == 0:

cur\_count\_1 = (prev\_count\_1 - sum\_to\_count[cur\_sum] + MOD) % MOD

prev\_count\_0, prev\_count\_1 = sum\_to\_count[cur\_sum], cur\_count\_1

sum\_to\_count[cur\_sum] = sum\_to\_count[cur\_sum] + 1

ans = (ans + cur\_count\_1) % MOD

return ans

**Solution 3: Binary Indexed Tree (Fenwick Tree) (600 min)**

**本题在Binary Indexed Tree的使用方法上是基于L315.Count of Smaller Numbers After Self的类型：BIT array存储的是频率，而对于频率的存储我们又分为用Harsh Table实现的frequency map筛选过后留下的没有空余的只有频率 > 0的数值才一一对应bucket的初始化方式，或者不经过frequency map筛选保留了所有空余基于给定n范围初始化所有数值一一对应bucket的方式**

**第二个重点是bucket的size开好了但是Binary Indexed Tree是基于原始给定的nums直接建立的吗？并不是，我们需要把原始给定的nums改造为presum，和Solution 1 & 2中的处理方式一致，扫描原始nums，遇到0就算为-1，然后通过加权算出presum，比如nums = [0,1,1,0,1] ->  [-1,1,1,-1,1] -> presum = [0,-1,0,1,0,1]，Binary Indexed Tree是基于presum建立的**

int[] nums = {0, 1, 1, 0, 1};

int n = nums.length;

int[] presum = new int[n + 1];

for(int i = 0; i < n; i++) {

presum[i + 1] = presum[i] + (nums[i] == 1 ? 1 : -1);

}

**第三个重点是我们如何通过presum建立Binary Indexed Tree？**

**用一句话总结presum的意义和建立Binary Indexed Tree的策略：为了保证nums[j, i]区间的1的个数一定大于0的个数，我们需要保证presum[j, i]一定大于0，这也就有了获取当j < i时同时满足presum[j] < presum[i]的j的个数的缘由，而反映到基于presum建立的Binary Indexed Tree上，我们需要Query the number of indices with prefix sums less than the current and update the answer，依然用presum = [0,-1,0,1,0,1]举例，我们用bit数组(也就是bucket)代表正在建立中的Binary Indexed Tree，采用size = 2 \* nums.length + 1策略，bit = [0,0,0,0,0,0,0,0,0,0,0]：**

class Solution {

class BinaryIndexedTree {

private int n;

private int[] bit;

public BinaryIndexedTree(int n) {

this.n = n;

this.bit = new int[2 \* n + 1];

}

public void update(int i, int delta) {

i += n + 1;

while(i < bit.length) {

bit[i] += delta;

i += i & -i;

}

}

public int query(int i) {

i += n + 1;

int sum = 0;

while(i > 0) {

sum += bit[i];

i -= i & -i;

}

return sum;

}

}

private static final int MOD = (int)1e9 + 7;

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length;

int[] presum = new int[n + 1];

for(int i = 0; i < n; i++) {

presum[i + 1] = presum[i] + (nums[i] == 0 ? -1 : 1);

}

BinaryIndexedTree tree = new BinaryIndexedTree(n);

int result = 0;

for(int val : presum) {

result = (result + tree.query(val - 1)) % MOD;

tree.update(val, 1);

}

return result;

}

}

**1.关于为何bit的size开出来是2 \* n + 1？**

**因为我们考虑presum数组最极端的情况，即由全是1的原始数组nums(即nums=[1, 1, 1..., 1])或者全是0的原始数组nums(即nums=[0, 0, 0..., 0])生成，那么presum数组中存储的值的范围就是[-n, n]，所以当我们采用桶(bucket)一一对应所有可能的presum数组中的数值的时候，桶(bucket)的范围就是[-n, n]，而每个桶(bucket)所存储的就是这个数值在presum数组中出现的次数，也即频率。而本题中为了提高处理速度从O(N)到O(logN)，我们引入了Binary Indexed Tree的结构，所以我们设定了一个bit数组，大小为2 \* n + 1满足包含所有可能的presum数组中数值[-n, n]的范围。**

**2.关于检索(query)和更新(update)的时候偏移量为n + 1而不是n？**

**关于偏移量为何设定为n + 1而不是n的问题，首先bit数组的每个下标本应该代表presum数组中对应的每个数值，可以认为是个bucket，比如presum中有个5，对应bit数组下标5，但实际上没这么简单，因为前面说到presum数组中会出现负数的情况，而且极限值是-n，n为原始数组nums的长度，那么问题来了，bit数组下标不可能为负数，怎么办？只能加偏移量，保证哪怕当前的数为-n也能对应到bit数组的下标0，所以必须在检索(query)和更新(update)的第一步就将给定输入i加上n变成(i + n)。但是只加n是会在更新(update)中出现死循环的，举例：比如给定nums = [0,0,0,0,0]，presum = [0,-1,-2,-3,-4,-5]，在update到最后一轮的时候会出现update(-5, 1)，此时i为-5，偏移量n要是为5，则i + n = 0，然后i + i & -i = 0，无限循环，为了避免i为0，我们需要给偏移量n再加1，这样保证遇到极限情况如nums = [0,0,0,0,0]时依然能运作，我们还可以通过极限情况nums = [1,1,1,1,1]来看偏移量n + 1是否可行，在update到最后一轮会出现update(5, 1)，此时i为5，偏移量n + 1 = 6，i = 11，而bit.length = 11，直接通过while(i < bit.length)的方式退出，实际上最后一轮update并不需要，因为不会再有下一次query需要用到最后一轮update后的bit数组**

// Dead loop when i += n => -5 + 5 = 0, because i += i & -i is 0 += 0 & 0 will keep as 0

public void update(int i, int delta) {

i += n;

while(i < bit.length) {

bit[i] += delta;

i += i & -i;

}

}

// Correct way to break the dead loop because i won't be 0, i += n + 1 => -5 + (5 + 1) = 1

public void update(int i, int delta) {

i += n + 1;

while(i < bit.length) {

bit[i] += delta;

i += i & -i;

}

}

**3.关于为何query(val - 1)的时候要减1？**

**因为对于presum[i](第i个presum数值)，我们需要找到在bit[0, i - 1]范围上符合Binary Indexed Tree坐标体系的所有坐标j(通过j = i - i & -i获得)上的数bit[j]之和，这些数值bit[j]本质上是小于presum[i]的某个presum截止到坐标i时出现的频率(这个频率会随着i的变化而变化，因为我们会不断update更新Binary Indexed Tree，也即更新bit)**

**4.关于为何update(val, 1)的时候delta是1？**

**因为我们考虑的是频率，每一个val落在对应的bucket上所代表的次数就是频率，每次加1**

**Style 1: 不经过frequency map筛选保留了所有空余基于给定n范围初始化所有数值一一对应bucket，根据前面详细分析，即bit数组的大小为(2 \* n + 1)个buckets**

class Solution {

class BinaryIndexedTree {

private int n;

private int[] bit;

public BinaryIndexedTree(int n) {

this.n = n;

this.bit = new int[2 \* n + 1];

}

public void update(int i, int delta) {

i += n + 1;

while(i < bit.length) {

bit[i] += delta;

i += i & -i;

}

}

public int query(int i) {

i += n + 1;

int sum = 0;

while(i > 0) {

sum += bit[i];

i -= i & -i;

}

return sum;

}

}

private static final int MOD = (int)1e9 + 7;

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length;

int[] presum = new int[n + 1];

for(int i = 0; i < n; i++) {

presum[i + 1] = presum[i] + (nums[i] == 0 ? -1 : 1);

}

BinaryIndexedTree tree = new BinaryIndexedTree(n);

int result = 0;

for(int val : presum) {

result = (result + tree.query(val - 1)) % MOD;

tree.update(val, 1);

}

return result;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://walkccc.me/LeetCode/problems/2031/>

class FenwickTree {

public FenwickTree(int n) {

this.n = n;

sums = new int[2 \* n + 1];

}

public void update(int i, int delta) {

i += n + 1; // Re-mapping

while (i < sums.length) {

sums[i] += delta;

i += i & -i;

}

}

public int get(int i) {

i += n + 1; // Re-mapping

int sum = 0;

while (i > 0) {

sum += sums[i];

i -= i & -i;

}

return sum;

}

private int n;

private int[] sums;

}

class Solution {

public int subarraysWithMoreZerosThanOnes(int[] nums) {

final int kMod = 1\_000\_000\_007;

int ans = 0;

int prefix = 0;

FenwickTree tree = new FenwickTree(nums.length);

tree.update(0, 1);

for (final int num : nums) {

prefix += num == 0 ? -1 : 1;

ans += tree.get(prefix - 1);

ans %= kMod;

tree.update(prefix, 1);

}

return ans;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/2031>

**结合前面的分析，offset设置为n + 1就足够了，没必要开1e5 + 1这么大(1e5实际上是给定的n的最大范围**1 <= nums.length <= 10^5**)**

class BinaryIndexedTree {

private int size;

private int[] treeArray;

// Constructor to initialize the Binary Indexed Tree with a given size

// The size is increased by an offset to handle negative indices

public BinaryIndexedTree(int size) {

int offset = (int) 1e5 + 1;

this.size = size + offset;

this.treeArray = new int[this.size + 1];

}

// Updates the Binary Indexed Tree at a specific position 'x' with value 'delta'

public void update(int x, int delta) {

int offset = (int) 1e5 + 1;

x += offset;

while (x <= size) {

treeArray[x] += delta;

x += lowbit(x); // Moves to the next index to be updated

}

}

// Queries the sum of values within the range [1, x] in the Binary Indexed Tree

public int query(int x) {

int offset = (int) 1e5 + 1;

x += offset;

int sum = 0;

while (x > 0) {

sum += treeArray[x];

x -= lowbit(x); // Moves down the tree to sum up the values

}

return sum;

}

// Returns the least significant bit of 'x'

// The offset here should be ignored as it's a static utility function

public static int lowbit(int x) {

return x & -x;

}

}

class Solution {

private static final int MOD = (int) 1e9 + 7; // Modulus for avoiding overflow

// Counts the number of subarrays with more zeros than ones in a binary array

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length; // Length of the input array

int[] prefixSums = new int[n + 1]; // Array for storing prefix sums

// Calculate the prefix sums array

for (int i = 0; i < n; ++i) {

prefixSums[i + 1] = prefixSums[i] + (nums[i] == 1 ? 1 : -1);

}

BinaryIndexedTree tree = new BinaryIndexedTree(n + 1);

int answer = 0; // Initialize result

// Iterate over the prefix sums array while updating and querying the tree

for (int value : prefixSums) {

// Query the number of indices with prefix sums less than the current

// and update the answer

answer = (answer + tree.query(value - 1)) % MOD;

// Update the tree's count for the current prefix sum

tree.update(value, 1);

}

return answer;

}

}

**Style 2: 用Harsh Table实现的frequency map筛选过后留下的没有空余的只有频率 > 0的数值才一一对应bucket，当然实际效果是我们依然需要开出一个大小在a.length + n + 1大小的bit数组(a是基于presum中所有unique number的排序数组)，不过也确实比2 \* n + 1的bit数组小，尤其是重复数字多且n很大的情况下，a.length明显小于n，我们只需要对BinaryIndexedTree的creator参数修改一下就行**

class Solution {

class BinaryIndexedTree {

private int n;

private int[] bit;

public BinaryIndexedTree(int n, int len) {

this.n = n;

this.bit = new int[len + n + 1];

}

public void update(int i, int delta) {

i += n + 1;

while(i < bit.length) {

bit[i] += delta;

i += i & -i;

}

}

public int query(int i) {

i += n + 1;

int sum = 0;

while(i > 0) {

sum += bit[i];

i -= i & -i;

}

return sum;

}

}

private static final int MOD = (int)1e9 + 7;

public int subarraysWithMoreZerosThanOnes(int[] nums) {

int n = nums.length;

int[] presum = new int[n + 1];

for(int i = 1; i <= n; i++) {

presum[i] = presum[i - 1] + (nums[i - 1] == 0 ? -1 : 1);

}

// Rebuild original input presum array with unique value only

Set<Integer> numSet = new HashSet<>();

for(int val : presum) {

numSet.add(val);

}

int[] a = new int[numSet.size()];

// Assign each number an index, like a bucket

int index = 0;

for(int num : numSet) {

a[index++] = num;

}

Arrays.sort(a);

// We still need to open bit array size by 'a.length + n + 1'

// e.g extreme case as input nums = [0,0,0,0,0],

// presum = [0,-1,-2,-3,-4,-5] -> a = [-5,-4,-3,-2,-1,0]

// to guarantee mapping to bit array index, we have to add

// offset = n + 1, and bit array size = 'a.length + n + 1'

BinaryIndexedTree tree = new BinaryIndexedTree(n, a.length);

//tree.update(0, 1);

int result = 0;

for(int val : presum) {

result = (result + tree.query(val - 1)) % MOD;

tree.update(val, 1);

}

return result;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)