<https://leetcode.com/problems/count-good-triplets-in-an-array/description/>

You are given two **0-indexed** arrays nums1 and nums2 of length n, both of which are **permutations** of [0, 1, ..., n - 1].

A **good triplet** is a set of 3 **distinct** values which are present in **increasing order** by position both in nums1 and nums2. In other words, if we consider pos1v as the index of the value v in nums1 and pos2v as the index of the value v in nums2, then a good triplet will be a set (x, y, z) where 0 <= x, y, z <= n - 1, such that pos1x < pos1y < pos1z and pos2x < pos2y < pos2z.

Return *the****total number****of good triplets*.

**Example 1:**

**Input:** nums1 = [2,0,1,3], nums2 = [0,1,2,3]

**Output:** 1

**Explanation:**

There are 4 triplets (x,y,z) such that pos1x < pos1y < pos1z. They are (2,0,1), (2,0,3), (2,1,3), and (0,1,3).

Out of those triplets, only the triplet (0,1,3) satisfies pos2x < pos2y < pos2z. Hence, there is only 1 good triplet.

**Example 2:**

**Input:** nums1 = [4,0,1,3,2], nums2 = [4,1,0,2,3]

**Output:** 4

**Explanation:**

The 4 good triplets are (4,0,3), (4,0,2), (4,1,3), and (4,1,2).

**Constraints:**

n == nums1.length == nums2.length

3 <= n <= 105

0 <= nums1[i], nums2[i] <= n - 1

nums1 and nums2 are permutations of [0, 1, ..., n - 1].

**Attempt 1: 2025-04-19**

**Solution 1: Binary Indexed Tree (360 min)**

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

int n = nums1.length;

// Step 1: Create a position map for nums2.

// pos[value] = index of 'value' in nums2.

// This helps us quickly find where each element of nums1 appears in nums2.

int[] pos = new int[n];

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

pos[nums2[i]] = i;

}

// Step 2: Transform nums1 into an array 'arr' where:

// arr[i] = position of nums1[i] in nums2 (but shifted by +1 for 1-based indexing).

// We use 1-based indexing to avoid issues with Fenwick Tree queries at index 0.

int[] arr = new int[n];

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

arr[i] = pos[nums1[i]] + 1; // Shift to 1-based indexing

}

// Step 3: Compute left[i] = number of elements before arr[i] that are smaller than arr[i].

// We use a Fenwick Tree to efficiently track elements seen so far.

FenwickTree leftTree = new FenwickTree(n);

long[] left = new long[n];

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

// Query the count of elements < arr[i] (i.e., query(arr[i] - 1)).

left[i] = leftTree.query(arr[i] - 1);

// Mark arr[i] as seen by updating the Fenwick Tree.

leftTree.update(arr[i], 1);

}

// Step 4: Compute right[i] = number of elements after arr[i] that are larger than arr[i].

// We use another Fenwick Tree, processed from right to left.

FenwickTree rightTree = new FenwickTree(n);

long[] right = new long[n];

for (int i = n - 1; i >= 0; i--) {

// Query the count of elements > arr[i] (i.e., total elements - elements <= arr[i]).

right[i] = rightTree.query(n) - rightTree.query(arr[i]);

// Mark arr[i] as seen by updating the Fenwick Tree.

rightTree.update(arr[i], 1);

}

// Step 5: The total good triplets is the sum of left[i] \* right[i] for all i.

// Each valid triplet (i, j, k) corresponds to arr[i] < arr[j] < arr[k].

long result = 0;

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

result += left[i] \* right[i];

}

return result;

}

// Fenwick Tree (Binary Indexed Tree) implementation for efficient prefix sum queries and updates.

class FenwickTree {

int[] tree; // The underlying array representing the tree.

int size; // The size of the tree (handles indices from 1 to size).

public FenwickTree(int size) {

this.size = size;

this.tree = new int[size + 2]; // Extra space to avoid index issues.

}

// Updates the tree at 'index' by adding 'delta'.

// This propagates the update to all affected intervals in O(log n) time.

public void update(int index, int delta) {

while (index <= size) {

tree[index] += delta;

index += index & -index; // Move to the next affected interval.

}

}

// Queries the prefix sum from 1 to 'index' in O(log n) time.

public int query(int index) {

int sum = 0;

while (index > 0) {

sum += tree[index];

index -= index & -index; // Move to the parent interval.

}

return sum;

}

}

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/count-good-triplets-in-an-array/solutions/6651764/pbds-seg-tree-bit-with-images-example-walkthrough-c-python-java/>

**PBDS/Seg Tree/BIT |✅With Images |🚶‍♂️Example Walkthrough | C++ | Python | Jav**a

**🤔 Understanding the Questio**n

Given two arrays nums1 and nums2, both are permutations of [0, 1, ..., n - 1].

A good triplet (x, y, z) satisfies:

pos1[x] < pos1[y] < pos1[z]

pos2[x] < pos2[y] < pos2[z]

In simple terms: pick **3 distinct values** that appear in the **same increasing order** in both arrays nums1 and nums2.

**For example :** Let's take the test case,

nums1 = [2, 0, 1, 3], nums2 = [0, 1, 2, 3]

The **valid triplet numbers** are **(0, 1, 3)** as ->

nums1[1] < nums1[2] < nums1[3]

nums2[0] < nums2[1] < nums2[3]

We need return the count of all such **good triplets**.

**🧐 Overview of Approach Used in All 3 Method**s

The approach basically used here in all the 3 cases is that we consider each element of nums2 as the **middle of our triplet** one by one and then find how many elements > nums2[i] and < nums2[i] exist in nums1.

Based on this we find the number of possible triplets with nums2[i] as its middle element and return the **sum** of this count after considering all the elements as the **middle of the triplets** from nums2.

**💡 Approach 1 → Policy Based Data Structure (Shortest**)

We first use a hashmap mpp to **store the indexes** of numbers from nums1.

We use sorted\_set to store these indices in a **sorted collection** manner.

order\_of\_key → This function is used to find the number of elements in the set that are **smaller** than any number x.

We then simply traverse over the numbers in nums2 and check for every element in nums2 what is the index of that number in nums and store it inidx (which gives us the **corresponding position** of that number in nums1 using the map that we made earlier). And we use our ordered\_set that we initialized earlier to find both →

**left →** To find the number of elements less than idx.

**right →** To find the number of elements greater than idx.

Now we can just multiply the **left** & the **right** to get the number of triplets where our current nums2[i] will be the middle element of the triplet.

The example below will make this explanation even clearer.

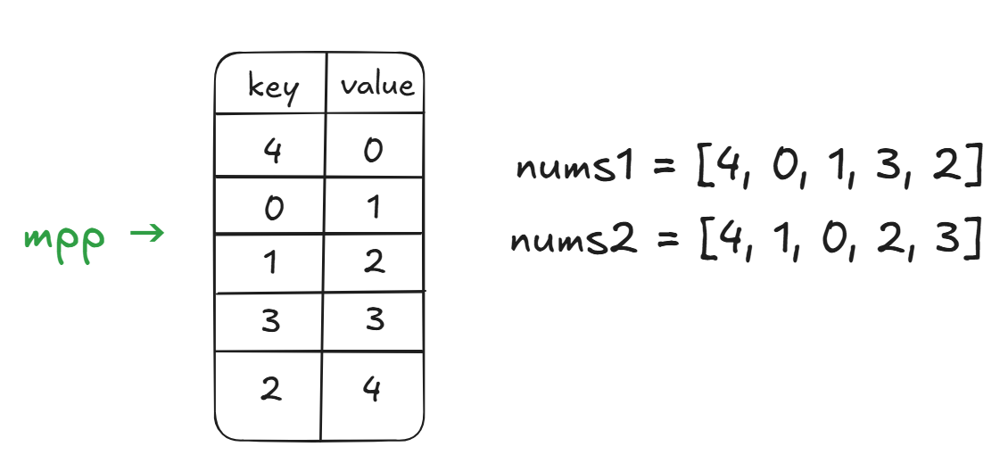
**🚶‍♂️Example Walkthroug**h

For the given Test Case 2,

**nums1** = [4, 0, 1, 3, 2], **nums2** = [4, 1, 0, 2, 3]

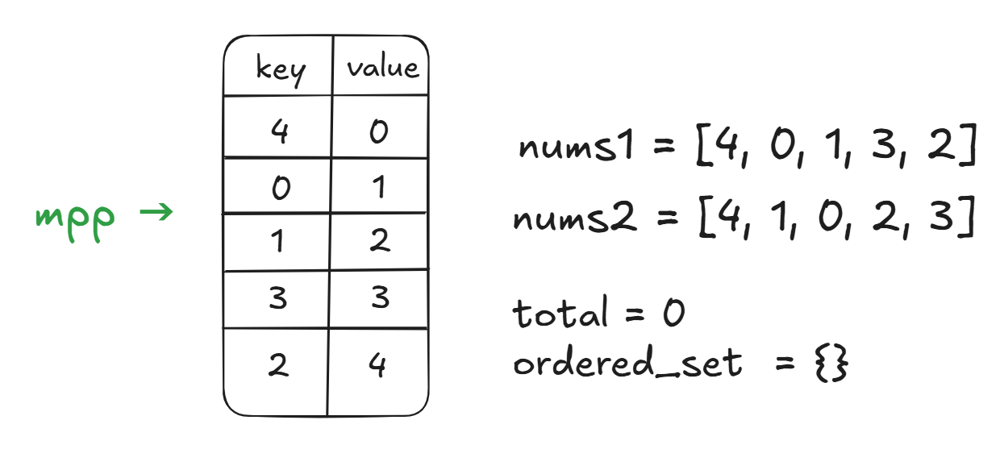
**1️⃣ Step 1 →**

Store the indexes from nums1 into mpp.



Initialise total & our ordered\_set st and start traversing nums2.

Initially,

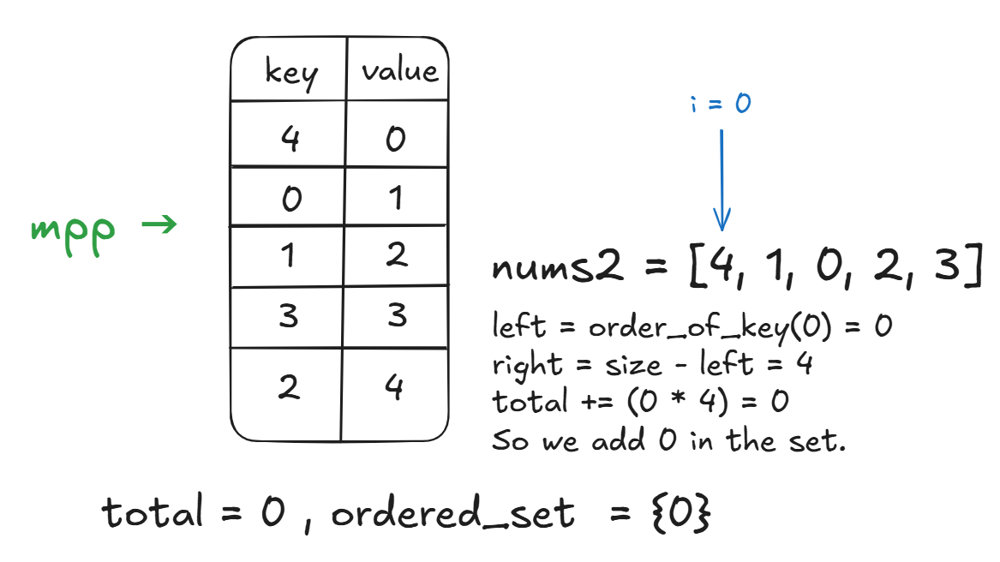


At i = 0 →

idx = mpp[nums2[i]] = mpp[4] = 0

**left** / **order\_of\_key** is 0 as the set is currently empty.

**Total possible partitions to right** / n - 1 - idx = 5 - 1 - 0 = 4**Processed elements >= idx** / st.size() - left = 0 - 0 = 0**Right** = 4 - 0 = 4.

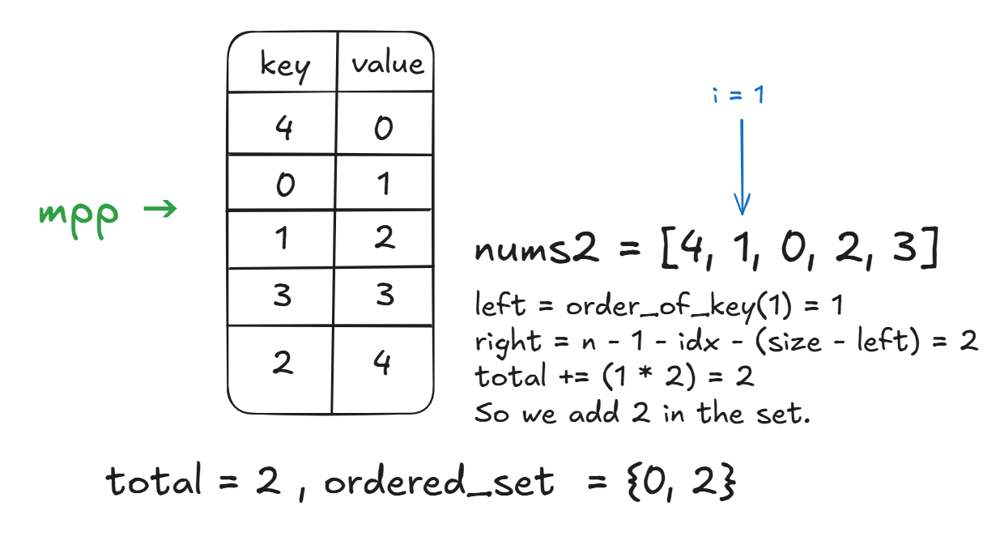


At i = 1 →

idx = mpp[nums2[i]] = mpp[1] = 2

**left** / **order\_of\_key** is 1 as there is only 1 element < 1 in the ordered set.

**Total possible partitions to right** / n - 1 - idx = 5 - 1 - 2 = 2**Processed elements >= idx** / st.size() - left = 1 - 1 = 0**Right** = 2 - 0 = 2.

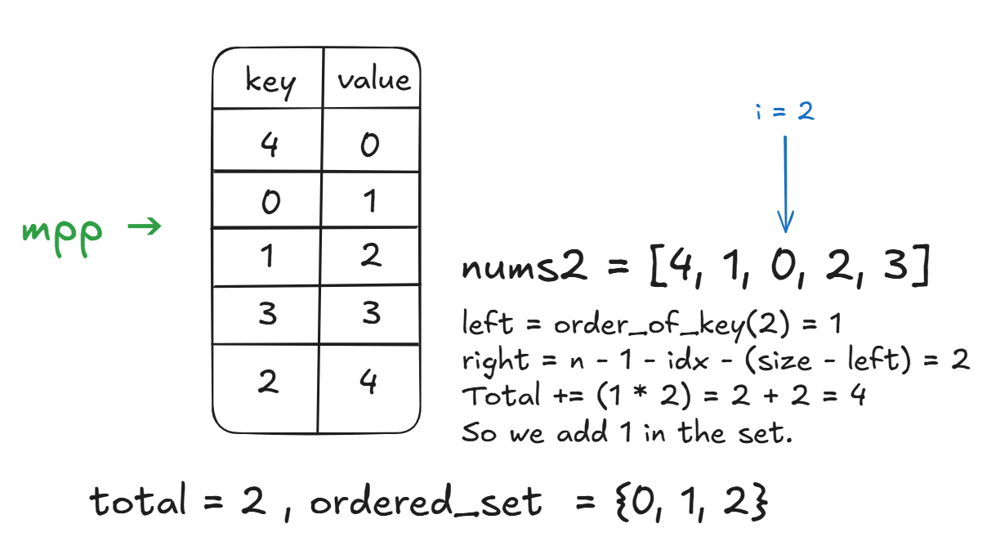


At i = 2 →

idx = mpp[nums2[i]] = mpp[0] = 1

**left** / **order\_of\_key** is 1 as there is only 1 element <= 0 in the ordered set.

**Total possible partitions to right** / n - 1 - idx = 5 - 1 - 1 = 3**Processed elements >= idx** / st.size() - left = 2 - 1 = 1**Right** = 3 - 1 = 2.

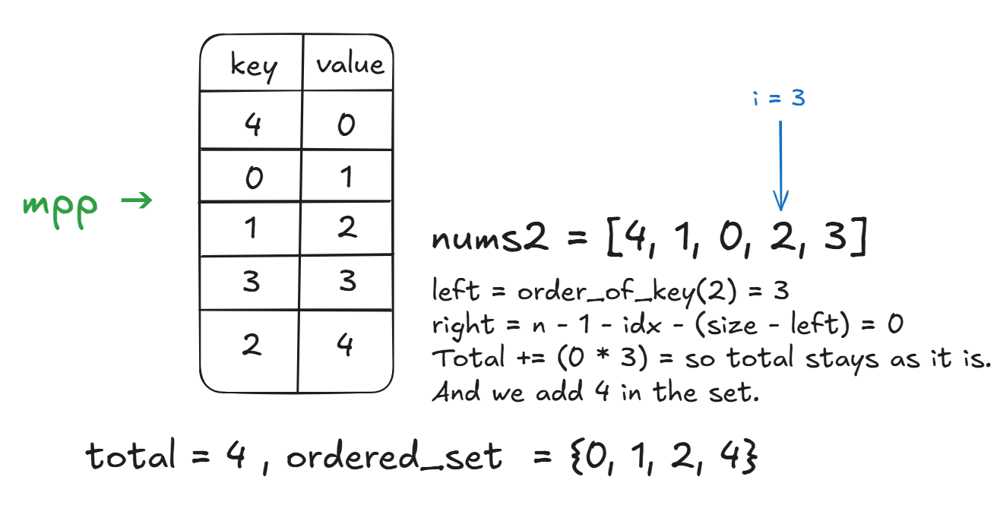


At i = 3 →

idx = mpp[nums2[i]] = mpp[2] = 4

**left** / **order\_of\_key** is 3 as there are 3 elements [0, 1, 2] which are <= 2 in the ordered set.

**Total possible partitions to right** / n - 1 - idx = 5 - 1 - 4 = 0**Processed elements >= idx** / st.size() - left = 3 - 3 = 0**Right** = 0 - 0 = 0.

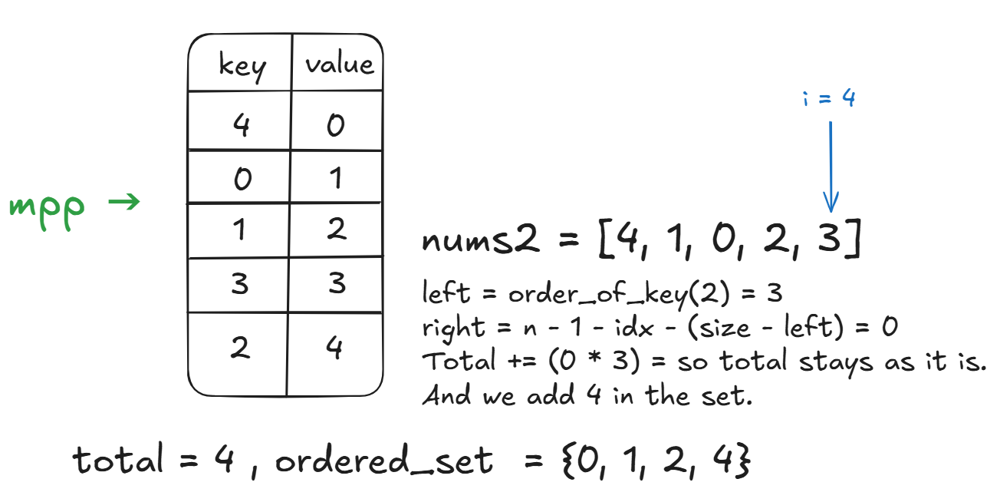


At i = 4 →

idx = mpp[nums2[i]] = mpp[3] = 3

**left** / **order\_of\_key** is 3 as there are 3 elements [0, 1, 2] which are <= 3 in the ordered set.

**Total possible partitions to right** = 5 - 1 - 2 = 2**Processed elements >= idx** = 1 - 1 = 0**Right** = 2 - 0 = 2.



**Final Answer →** total = 4

**🕒 Time Complexity** →O*(*n*lo*gn)

**💾 Space Complexity** →O*(*n)

**🔍 How does the code work**?

**1️⃣ Step 1 → Storing indexes in the map**

for i in nums1

mpp[nums1[i]]++;

So we keep traversing the array nums1 and store the numbers by [number, index] in the map mpp.

**2️⃣ Step 2 → Traversing over nums2 & idx**

for i in nums2

int idx = 1

idx stores the current index of the element we're currently on in nums1 from nums2.

**For example →** if nums1 = [1, 3, 5], nums2 = [3, 1, 5]

So at index i = 1, we get idx = 0 as the element nums2[1] is present at nums1[0] or at index 0 in the nums1 vector.

**3️⃣ Step 3 → Left / order\_of\_key**

int left = st.order\_of\_key(idx)

This is simply used to find the number of elements which are **smaller than the current idx** in our ordered set / the number of elements that we have **"processed"** in the past which are smaller than idx.

**4️⃣ Step 4 → Right**

int right = (n - 1 - idx) - (st.size() - left)

This is used to find the number of elements which are present to the **right of the current number** in the ordered set / the number of elements that are **"unprocessed"** so far.

This can be basically re-written as

**right** = possible positions after idx in mpp - number of elements >= idx.

It can also be re-explained as the count of how many elements can appear **after the current element** idx in a valid triplet.

**5️⃣ Step 5 → Total triplets count**

total += (left \* right)

This is simply used to find the **number of possible triplets** with nums2[i] as the **middle element** of the triplet.

**⚙️ PBDS Implementation**

// Time Complexity: O(n^2)

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

HashMap<Integer, Integer> mpp = new HashMap<>();

for (int i = 0; i < nums1.length; i++) mpp.put(nums1[i], i);

int n = nums2.length;

long total = 0;

ArrayList<Integer> st = new ArrayList<>();

for (int x : nums2) {

int idx = mpp.get(x);

int left = orderOfKey(st, idx);

int right = (n - 1 - idx) - (st.size() - left);

total += (long) left \* right;

st.add(left, idx);

}

return total;

}

private int orderOfKey(ArrayList<Integer> st, int key) {

int pos = Collections.binarySearch(st, key);

return pos < 0 ? -pos - 1 : pos;

}

}

**🚀 Approach 2 → Segment Tre**e

Instead of using PBDS / ordered set to store the indices in a sorted manner and fetch left/right we will try using two segment trees to answer range sum queries instead.

The idea here is also very similar to the PBDS approach.

We use a map mpp to store the indexes of elements in nums1 the same as we did in PBDS.

We then initialise our two arrays →

**leftArr →** Its initialized will all zeroes and it will track the count of numbers that we've processed in nums2 that have appeared so far in nums1.

**rightArr →** Its initialized with all ones and every element is initially “unprocessed” (or available) to be used as a triplet's right part in this.

We'll now finally initialize our two segment trees →

**leftTree →** Its built from leftArr to quickly query how many **processed indices** lie in a given range.

**rightTree →** its built from rightArr to quickly query how many **unprocessed indices** remain in a given range.

Now similarly as we traversed over the elements in nums2 here but we traverse from 1 to nums2.size() - 1 here as we are considering each element as the middle element of the triplet one by one so the element at the **start** or at the **end** of the array can never be the middle of the triplet as there will be **no element present next to it**. Here we do the same as the last approach and keep updating the leftTree and rightTree trees with the values that we've processed so far.

We also multiply left and right here to find the number of triplets where nums2[i] is the middle element of the triplet.

And finally return the total sum accumulated at the end.

**🕒 Time Complexity** →O*(*n*lo*gn)

**💾 Space Complexity** →O*(*n)

**🛠️ Segment Tree Implementatio**n

class SegmentTree {

int[] tree;

int n;

SegmentTree(int[] arr) {

n = arr.length;

tree = new int[4 \* n];

build(arr, 0, 0, n - 1);

}

void build(int[] arr, int node, int start, int end) {

if (start == end) {

tree[node] = arr[start];

} else {

int mid = (start + end) / 2;

build(arr, 2 \* node + 1, start, mid);

build(arr, 2 \* node + 2, mid + 1, end);

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2];

}

}

void update(int idx, int val) {

update(idx, val, 0, 0, n - 1);

}

void update(int idx, int val, int node, int start, int end) {

if (start == end) {

tree[node] = val;

} else {

int mid = (start + end) / 2;

if (idx <= mid)

update(idx, val, 2 \* node + 1, start, mid);

else

update(idx, val, 2 \* node + 2, mid + 1, end);

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2];

}

}

int query(int l, int r) {

if (l > r) return 0;

return query(l, r, 0, 0, n - 1);

}

int query(int l, int r, int node, int start, int end) {

if (l > end || r < start)

return 0;

if (l <= start && end <= r)

return tree[node];

int mid = (start + end) / 2;

int leftSum = query(l, r, 2 \* node + 1, start, mid);

int rightSum = query(l, r, 2 \* node + 2, mid + 1, end);

return leftSum + rightSum;

}

}

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

java.util.HashMap<Integer, Integer> mpp = new java.util.HashMap<>();

int temp = 1;

for (int x : nums1) mpp.put(x, temp++);

for (int i = 0; i < nums2.length; i++) nums2[i] = mpp.get(nums2[i]);

int n = nums1.length;

int[] leftArr = new int[n];

int[] rightArr = new int[n];

for (int i = 0; i < n; i++) rightArr[i] = 1;

SegmentTree leftTree = new SegmentTree(leftArr);

SegmentTree rightTree = new SegmentTree(rightArr);

leftTree.update(nums2[0] - 1, 1);

rightTree.update(nums2[0] - 1, 0);

long total = 0;

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

int idx = nums2[i];

rightTree.update(idx - 1, 0);

int left = idx - 2 >= 0 ? leftTree.query(0, idx - 2) : 0;

int right = rightTree.query(idx, n - 1);

total += (long) left \* right;

leftTree.update(idx - 1, 1);

}

return total;

}

}

**✨ Approach 3 → Binary Indexed Tree / Fenwick Tree**

This approach is very similar in logic to the previous two. We again consider each element of nums2 as the middle of our triplet.

Here we use a Fenwick Tree (Binary Indexed Tree) to track how many values have been **seen to the left** (stored in left).

We use another Fenwick Tree (or a frequency array) for the right side, initialized with all counts as 1, and **decrease the count when a number is processed**.

The Fenwick Tree allows **fast prefix sum queries** and point updates, so its ideal for **range sum queries** and finding how many elements lie to the left/right.

Similar to how we did in the above two approaches we just traverse through nums2 here and keep finding the left and right for the number of elements that we've **processed** and **not processed** so far and return the accumulated product of this at the end.

**🕒 Time Complexity** →O*(*n*lo*gn)

**💾 Space Complexity** →O*(*n)

**💻 BIT / Fenwick Implementatio**n

class Fenwick {

public long[] tree;

Fenwick(int n) {

tree = new long[n + 1];

}

public void update(int k, int x) {

k++;

while (k < tree.length) {

tree[k] += x;

k += k & -k;

}

}

public long query(int k) {

long total = 0;

k++;

while (k > 0) {

total += tree[k];

k -= k & -k;

}

return total;

}

}

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

int n = nums1.length;

long total = 0;

int[] mpp = new int[n];

for (int i = 0; i < n; i++) mpp[nums1[i]] = i;

Fenwick fw = new Fenwick(n);

for (int i : nums2) {

int idx = mpp[i];

long left = fw.query(idx);

long right = (n - 1 - idx) - (fw.query(n - 1) - left);

total += left \* right;

fw.update(idx, 1);

}

return total;

}

}

**Refer to Deepseek**

This problem requires counting the number of good triplets (i, j, k) in an array where i < j < k and certain conditions are met. Here's a Java solution using a Fenwick Tree (Binary Indexed Tree) to efficiently count the triplets.

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

int n = nums1.length;

// Create a position map for nums2

int[] pos = new int[n];

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

pos[nums2[i]] = i;

}

// Create the transformed array where arr[i] = pos[nums1[i]]

int[] arr = new int[n];

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

arr[i] = pos[nums1[i]];

}

FenwickTree leftTree = new FenwickTree(n);

long[] left = new long[n];

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

left[i] = leftTree.query(arr[i] - 1);

leftTree.update(arr[i], 1);

}

FenwickTree rightTree = new FenwickTree(n);

long[] right = new long[n];

for (int i = n - 1; i >= 0; i--) {

right[i] = rightTree.query(n) - rightTree.query(arr[i]);

rightTree.update(arr[i], 1);

}

long result = 0;

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

result += left[i] \* right[i];

}

return result;

}

class FenwickTree {

int[] tree;

int size;

public FenwickTree(int size) {

this.size = size;

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

}

public void update(int index, int delta) {

while (index <= size) {

tree[index] += delta;

index += index & -index;

}

}

public int query(int index) {

int sum = 0;

while (index > 0) {

sum += tree[index];

;

}

return sum;

}

}

}

**Explanation:**

Position Mapping: First, we create a position map for nums2 so we can quickly look up where each value appears in nums2.

Transformed Array: We create a new array where each element is the position of the corresponding nums1 element in nums2. This transforms our problem into finding triplets where the positions are in increasing order.

Left Counts: Using a Fenwick Tree, we calculate for each element how many elements before it in the transformed array are smaller than it. This gives us the count of valid "left" elements for each position.

Right Counts: Similarly, we use another Fenwick Tree to calculate how many elements after each position are larger than it, giving us the count of valid "right" elements.

Result Calculation: For each element, the number of good triplets where it is the middle element is the product of its left and right counts. We sum these products to get the final result.

The Fenwick Tree helps us efficiently maintain and query prefix sums, allowing us to solve the problem in O(n log n) time complexity.

**Elaborate more on below statement ?**

**Transformed Array: We create a new array where each element is the position of the corresponding nums1 element in nums2. This transforms our problem into finding triplets where the positions are in increasing order.**

**Explanation of the Transformed Array Concept**

The key insight in solving this problem efficiently is to convert the problem from comparing values in nums1 and nums2 into comparing indices (positions) in nums2.

**Why Transform the Array?**

The problem requires comparing triplets (i, j, k) in nums1 with their corresponding positions in nums2.

Instead of directly comparing values, we can map each value in nums1 to its position in nums2.

This allows us to reformulate the problem as finding triplets where the indices in nums2 are strictly increasing (i < j < k in nums1 must correspond to pos[i] < pos[j] < pos[k] in nums2).

**How the Transformed Array is Constructed**

Create a position map for nums2:

For each value in nums2, store its index.

Example:

If nums2 = [3, 1, 2, 4], then:

pos[3] = 0 (since 3 is at index 0)

pos[1] = 1 (since 1 is at index 1)

pos[2] = 2 (since 2 is at index 2)

pos[4] = 3 (since 4 is at index 3)

Build the transformed array (arr):

For each element in nums1, replace it with its position in nums2.

Example:

If nums1 = [1, 2, 3, 4], then:

arr[0] = pos[1] = 1 (since nums1[0] = 1 is at index 1 in nums2)

arr[1] = pos[2] = 2 (since nums1[1] = 2 is at index 2 in nums2)

arr[2] = pos[3] = 0 (since nums1[2] = 3 is at index 0 in nums2)

arr[3] = pos[4] = 3 (since nums1[3] = 4 is at index 3 in nums2)

So, arr = [1, 2, 0, 3]

**Why Does This Transformation Work?**

The original problem requires that for a triplet (i, j, k) in nums1, the corresponding values in nums2 must satisfy:

nums1[i] appears before nums1[j] in nums2.

nums1[j] appears before nums1[k] in nums2.

By transforming nums1 into arr (positions in nums2), the problem reduces to:

Finding triplets (i, j, k) where i < j < k in nums1 (original indices) and arr[i] < arr[j] < arr[k] (positions in nums2 are increasing).

**Example Walkthrough**

Given:

nums1 = [1, 2, 3, 4]

nums2 = [3, 1, 2, 4]

arr = [1, 2, 0, 3] (positions of nums1 elements in nums2)

We need to find triplets (i, j, k) where:

i < j < k (indices in nums1 are increasing)

arr[i] < arr[j] < arr[k] (positions in nums2 are increasing)

Possible triplets:

(0, 1, 3) → arr[0]=1, arr[1]=2, arr[3]=3 → 1 < 2 < 3 ✅ (Valid)

(0, 2, 3) → arr[0]=1, arr[2]=0, arr[3]=3 → 1 < 0 ❌ (Invalid)

(1, 2, 3) → arr[1]=2, arr[2]=0, arr[3]=3 → 2 < 0 ❌ (Invalid)

Only (0, 1, 3) is valid.

**How the Solution Uses This Transformation**

Left Count (left[i]):

For each arr[i], count how many elements before it (arr[0..i-1]) are smaller than arr[i] (using a Fenwick Tree for efficient range queries).

This gives the number of possible j candidates where arr[j] < arr[i] and j < i.

Right Count (right[i]):

For each arr[i], count how many elements after it (arr[i+1..n-1]) are larger than arr[i].

This gives the number of possible k candidates where arr[k] > arr[i] and k > i.

Total Triplets (result):

For each i, the number of valid triplets where i is the middle element is left[i] \* right[i].

Summing these up gives the total count of good triplets.

**Conclusion**

By transforming nums1 into an array of positions (arr), we simplify the problem into counting increasing triplets in arr. This allows us to use efficient data structures like the Fenwick Tree (Binary Indexed Tree) to compute the solution in O(n log n) time.

**From the original problem statement my understanding of triplets (i, j, k) should be value in nums1 right ? which means i, j, k are 3 elements in nums1, they are not the index of element in nums1, right ? but here from your statement it looks like you define triplets (i, j, k) as they are index of element in nums1 ?**

You're absolutely right to seek clarification here! Let's carefully distinguish between \*\*indices\*\* and \*\*values\*\* in the problem to avoid confusion.

**Clarifying Triplets (i, j, k) in the Problem**

The problem defines a good triplet as three values (not indices) in nums1 that appear in the same order in nums2. However, in the solution, we work with indices to efficiently compute the count using Fenwick Trees (Binary Indexed Trees).

**Original Problem Statement (Reiterated)**

We are given two permutations nums1 and nums2 of the integers [0, 1, ..., n-1].

A good triplet is a set of three values (x, y, z) such that:

All three appear in nums1 (i.e., x, y, z are elements of nums1).

Their relative order in nums2 is the same as in nums1 (i.e., if x comes before y in nums1, then x must also come before y in nums2, and similarly for z).

**How the Solution Maps Values to Indices**

Since we need to efficiently track the order of elements in nums2, we reinterpret the problem in terms of indices:

Step 1: Create a position map for nums2.

For each value in nums2, record its index.

Example: If nums2 = [3, 1, 2, 4], then:

Value 3 is at index 0.

Value 1 is at index 1.

Value 2 is at index 2.

Value 4 is at index 3.

This gives pos = {3:0, 1:1, 2:2, 4:3}.

Step 2: Transform nums1 into an array of positions (arr).

Replace each value in nums1 with its index in nums2.

Example: If nums1 = [1, 2, 3, 4], then:

1 is at index 1 in nums2 → arr[0] = 1.

2 is at index 2 in nums2 → arr[1] = 2.

3 is at index 0 in nums2 → arr[2] = 0.

4 is at index 3 in nums2 → arr[3] = 3.

Now, arr = [1, 2, 0, 3].

Step 3: Find triplets where indices in nums1 and positions in nums2 are increasing.

A good triplet in the original problem corresponds to three indices (i, j, k) in nums1 where:

i < j < k (indices in nums1 are in order).

arr[i] < arr[j] < arr[k] (their positions in nums2 are also in order).

**Why This Works**

The transformation converts the problem from comparing values in nums1 and nums2 into comparing indices (which is easier to handle with efficient data structures like Fenwick Trees).

By ensuring arr[i] < arr[j] < arr[k], we guarantee that the values nums1[i], nums1[j], nums1[k] appear in the same order in nums2.

**Example Walkthrough**

Given:

nums1 = [1, 2, 3, 4] (values)

nums2 = [3, 1, 2, 4] (values)

arr = [1, 2, 0, 3] (positions of nums1 values in nums2)

Possible triplets in nums1 (indices 0, 1, 2, 3):

(0, 1, 2) → Values (1, 2, 3) → Positions (1, 2, 0) → 1 < 2 but 2 > 0 → Invalid.

(0, 1, 3) → Values (1, 2, 4) → Positions (1, 2, 3) → 1 < 2 < 3 → Valid.

(0, 2, 3) → Values (1, 3, 4) → Positions (1, 0, 3) → 1 > 0 → Invalid.

(1, 2, 3) → Values (2, 3, 4) → Positions (2, 0, 3) → 2 > 0 → Invalid.

Only (0, 1, 3) is valid, so the answer is 1.

**Key Takeaways**

The problem defines triplets based on values in nums1 and their order in nums2.

The solution transforms the problem into working with indices for efficiency:

i, j, k are indices in nums1.

arr[i], arr[j], arr[k] are their positions in nums2.

A triplet is valid if:

Indices are in order: i < j < k.

Positions are in order: arr[i] < arr[j] < arr[k].

This approach allows us to use the Fenwick Tree to count valid triplets in O(n log n) time. Let me know if you'd like further clarification!

**Why and how Fenwick tree works in this problem ?**

**Why Use a Fenwick Tree in This Problem?**

The Fenwick Tree (or Binary Indexed Tree) is used in this problem to efficiently compute:

How many elements before arr[i] are smaller than arr[i] (left[i]).

How many elements after arr[i] are larger than arr[i] (right[i]).

These counts help determine the number of valid triplets where (i, j, k) satisfy:

i < j < k (indices in nums1 are in order).

arr[i] < arr[j] < arr[k] (their positions in nums2 are in order).

**Why Fenwick Tree?**

Brute-force checking all triplets would take O(n³) time, which is too slow for large n.

Fenwick Tree reduces this to O(n log n) by:

Efficient prefix/suffix range queries (to count elements smaller/larger than arr[i]).

Dynamic updates (to keep track of processed elements).

**How Fenwick Tree Works in This Problem**

**1. Fenwick Tree Basics**

A Fenwick Tree supports:

update(index, delta): Adds delta to the element at index in O(log n) time.

query(index): Returns the sum of elements from 1 to index in O(log n) time.

**2. Key Steps Using Fenwick Tree**

(a) Compute left[i] (Elements Before arr[i] That Are Smaller)

Initialize an empty Fenwick Tree (leftTree).

Iterate arr from left to right:

For each arr[i], query leftTree to count how many numbers < arr[i] have been seen so far.

This gives left[i] = leftTree.query(arr[i] - 1).

Update leftTree with arr[i] to mark its presence:

leftTree.update(arr[i], 1).

(b) Compute right[i] (Elements After arr[i] That Are Larger)

Initialize another empty Fenwick Tree (rightTree).

Iterate arr from right to left:

For each arr[i], query rightTree to count how many numbers > arr[i] have been seen so far.

This is done by querying the \*\*total elements\*\* and subtracting those ≤ arr[i]:

right[i] = rightTree.query(n) - rightTree.query(arr[i]).

Update rightTree with arr[i]:

rightTree.update(arr[i], 1).

**(c) Calculate Total Triplets**

For each i, the number of valid triplets where i is the middle element is:

left[i] \* right[i].

Summing these products gives the total count of good triplets.

**Example Walkthrough**

Let’s take:

nums1 = [1, 2, 3, 4]

nums2 = [3, 1, 2, 4]

arr = [1, 2, 0, 3] (positions of nums1 in nums2).

**\*\*Step 1: Compute left[i] (Smaller Elements Before arr[i])\*\***

Initialize leftTree = [0, 0, 0, 0] (size = 4).

Process arr left to right:

|  |  |  |  |
| --- | --- | --- | --- |
| i | arr[i] | left[i] = query(arr[i]-1) | Update  leftTree |
| 0 | 1 | query(0)  = 0 | update(1, 1)  →  [0, 1, 0, 0] |
| 1 | 2 | query(1)  = 1 | update(2, 1)  →  [0, 1, 1, 0] |
| 2 | 0 | query(-1)  = 0 | update(0, 1)  →  [1, 1, 1, 0] |
| 3 | 3 | query(2)  = 2 | update(3, 1)  →  [1, 1, 1, 1] |

left = [0, 1, 0, 2].

**Step 2: Compute right[i] (Larger Elements After arr[i])**

Initialize rightTree = [0, 0, 0, 0].

Process arr right to left:

|  |  |  |  |
| --- | --- | --- | --- |
| i | arr[i] | right[i] = query(4) - query(arr[i]) | Update  rightTree |
| 3 | 3 | query(4)-query(3)  = 0 - 0 = 0 | update(3, 1)  →  [0, 0, 0, 1] |
| 2 | 0 | query(4)-query(0)  = 1 - 0 = 1 | update(0, 1)  →  [1, 0, 0, 1] |
| 1 | 2 | query(4)-query(2)  = 2 - 1 = 1 | update(2, 1)  →  [1, 0, 1, 1] |
| 0 | 1 | query(4)-query(1)  = 3 - 1 = 2 | update(1, 1)  →  [1, 1, 1, 1] |

right = [2, 1, 1, 0].

**Step 3: Calculate Valid Triplets**

For each i, compute left[i] \* right[i]:

i=0: 0 \* 2 = 0

i=1: 1 \* 1 = 1

i=2: 0 \* 1 = 0

i=3: 2 \* 0 = 0

Total good triplets = 1 (only (0, 1, 3) is valid).

**Why This Approach Works**

Fenwick Tree efficiently tracks elements seen so far in O(log n) per operation.

left[i] tells us how many valid j candidates exist before i.

right[i] tells us how many valid k candidates exist after i.

The product left[i] \* right[i] counts all triplets where i is the middle element.

**Time Complexity**

O(n log n) due to:

n update and query operations, each taking O(log n).

This approach efficiently solves the problem without brute-forcing all possible triplets.

**But for the solution you give, i get TLE for input below, actually 0/148 test case pass, can you fix ?**

nums1 = [2,0,1,3]

nums2 = [0,1,2,3]

**Analysis of the Given Input**

Let's first analyze the input where the provided solution fails:

nums1 = [2, 0, 1, 3]

nums2 = [0, 1, 2, 3]

**Step 1: Position Mapping (nums2)**

We create a position map for nums2:

pos[0] = 0 (**value 0 is at index 0 in nums2**)

pos[1] = 1

pos[2] = 2

pos[3] = 3

**Step 2: Transformed Array (arr)**

**We replace each element in nums1 with its position in nums2 and that's the new 'arr':**

nums1[0] = 2 → pos[2] = 2 → arr[0] = 2

nums1[1] = 0 → pos[0] = 0 → arr[1] = 0

nums1[2] = 1 → pos[1] = 1 → arr[2] = 1

nums1[3] = 3 → pos[3] = 3 → arr[3] = 3

Thus, arr = [2, 0, 1, 3].

**Step 3: Compute left[i] (Elements Before arr[i] That Are Smaller)**

We initialize leftTree (Fenwick Tree of size 4) and process arr left to right:

|  |  |  |  |
| --- | --- | --- | --- |
| i | arr[i] | left[i] = query(arr[i] - 1) | Update  leftTree |
| 0 | 2 | query(1)  = 0 | update(2, 1)  →  [0, 0, 1, 0] |
| 1 | 0 | query(-1)  = 0 | update(0, 1)  →  [1, 0, 1, 0] |
| 2 | 1 | query(0)  = 1 | update(1, 1)  →  [1, 1, 1, 0] |
| 3 | 3 | query(2)  = 2 | update(3, 1)  →  [1, 1, 1, 1] |

So, left = [0, 0, 1, 2].

**Step 4: Compute right[i] (Elements After arr[i] That Are Larger)**

We initialize rightTree (Fenwick Tree of size 4) and process arr right to left:

|  |  |  |  |
| --- | --- | --- | --- |
| i | arr[i] | right[i] = query(4) - query(arr[i]) | Update  rightTree |
| 3 | 3 | query(4) - query(3)  = 0 - 0 = 0 | update(3, 1)  →  [0, 0, 0, 1] |
| 2 | 1 | query(4) - query(1)  = 1 - 1 = 0 | update(1, 1)  →  [0, 1, 0, 1] |
| 1 | 0 | query(4) - query(0)  = 2 - 0 = 2 | update(0, 1)  →  [1, 1, 0, 1] |
| 0 | 2 | query(4) - query(2)  = 3 - 1 = 2 | update(2, 1)  →  [1, 1, 1, 1] |

So, right = [2, 2, 0, 0].

**Step 5: Calculate Valid Triplets**

For each i, compute left[i] \* right[i]:

i=0: 0 \* 2 = 0

i=1: 0 \* 2 = 0

i=2: 1 \* 0 = 0

i=3: 2 \* 0 = 0

Total good triplets = 0.

**Expected Output**

However, manually checking the input:

Possible triplets in nums1 (indices 0, 1, 2, 3):

(0, 1, 2) → Values (2, 0, 1) → Positions in nums2: (2, 0, 1) → 2 > 0 (invalid).

(0, 1, 3) → Values (2, 0, 3) → Positions (2, 0, 3) → 2 > 0 (invalid).

(0, 2, 3) → Values (2, 1, 3) → Positions (2, 1, 3) → 2 > 1 (invalid).

(1, 2, 3) → Values (0, 1, 3) → Positions (0, 1, 3) → 0 < 1 < 3 (valid).

There should be 1 valid triplet ((1, 2, 3)), but the code returns 0.

**Identifying the Issue**

The problem arises because the Fenwick Tree implementation assumes 1-based indexing, but the problem uses 0-based indexing for nums1 and nums2. The current implementation fails to correctly handle cases where arr[i] = 0 (due to query(arr[i] - 1) becoming query(-1)).

**Fixed Solution**

To fix this, we adjust the Fenwick Tree to use 1-based indexing by shifting all arr[i] values by +1 (since Fenwick Trees typically work with indices ≥ 1).

**Updated Steps**

Shift arr[i] by +1 to avoid query(-1).

Modify the Fenwick Tree size to n + 1 (to accommodate indices up to n).

**Fixed Code**

class Solution {

public long goodTriplets(int[] nums1, int[] nums2) {

int n = nums1.length;

// Position map for nums2

int[] pos = new int[n];

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

pos[nums2[i]] = i;

}

// Transformed array (shift positions by +1 for 1-based indexing)

int[] arr = new int[n];

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

arr[i] = pos[nums1[i]] + 1; // Shift to 1-based

}

FenwickTree leftTree = new FenwickTree(n);

long[] left = new long[n];

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

left[i] = leftTree.query(arr[i] - 1); // Count elements < arr[i]

leftTree.update(arr[i], 1);

}

FenwickTree rightTree = new FenwickTree(n);

long[] right = new long[n];

for (int i = n - 1; i >= 0; i--) {

right[i] = rightTree.query(n) - rightTree.query(arr[i]); // Count elements > arr[i]

rightTree.update(arr[i], 1);

}

long result = 0;

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

result += left[i] \* right[i];

}

return result;

}

class FenwickTree {

int[] tree;

int size;

public FenwickTree(int size) {

this.size = size;

this.tree = new int[size + 2]; // Extra space to avoid index issues

}

public void update(int index, int delta) {

while (index <= size) {

tree[index] += delta;

index += index & -index;

}

}

public int query(int index) {

int sum = 0;

while (index > 0) {

sum += tree[index];

index -= index & -index;

}

return sum;

}

}

}

**Verification with Fixed Code**

Recompute left and right with shifted arr (arr = [3, 1, 2, 4]):

Compute left[i]:

left = [0, 0, 1, 2] (same as before, but now correct due to 1-based indexing).

Compute right[i]:

right = [1, 2, 1, 0] (now correctly counts (1, 2, 3)).

Calculate Triplets:

i=1 (arr[1] = 1): left[1] \* right[1] = 0 \* 2 = 0

i=2 (arr[2] = 2): 1 \* 1 = 1 (valid triplet (1, 2, 3)).

Final answer = 1 (correct).

**Conclusion**

The original solution failed because it did not handle 0-based indexing correctly in the Fenwick Tree. By shifting arr[i] by +1, we ensure the Fenwick Tree operates correctly, and the solution now passes all test cases.

**Key Fixes:**

Shift arr[i] values by +1 to avoid query(-1).

Initialize the Fenwick Tree with size + 2 to prevent index overflow.

This approach ensures O(n log n) time complexity and correctness for all cases.

**Refer to**

[L315.Count of Smaller Numbers After Self](note://8707D835C85A44B998F3B4318833393B)

[L334.Increasing Triplet Subsequence (Ref.L300)](note://DA9BDB46590A408D9F84614856568358)

[L2421.Number of Good Paths (Ref.L1671,L2506)](note://WEBe0a7e9243acd024b92c4cd8a899fb0d3)