<https://leetcode.com/problems/number-of-pairs-satisfying-inequality/description/>

You are given two **0-indexed** integer arrays nums1 and nums2, each of size n, and an integer diff. Find the number of **pairs** (i, j) such that:

0 <= i < j <= n - 1 **and**

nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff.

Return**the number of pairs that satisfy the conditions.**

**Example 1:**

**Input:** nums1 = [3,2,5], nums2 = [2,2,1], diff = 1

**Output:** 3

**Explanation:**

There are 3 pairs that satisfy the conditions:

1. i = 0, j = 1: 3 - 2 <= 2 - 2 + 1. Since i < j and 1 <= 1, this pair satisfies the conditions.

2. i = 0, j = 2: 3 - 5 <= 2 - 1 + 1. Since i < j and -2 <= 2, this pair satisfies the conditions.

3. i = 1, j = 2: 2 - 5 <= 2 - 1 + 1. Since i < j and -3 <= 2, this pair satisfies the conditions.

Therefore, we return 3.

**Example 2:**

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

**Output:** 0

**Explanation:**

Since there does not exist any pair that satisfies the conditions, we return 0.

**Constraints:**

n == nums1.length == nums2.length

2 <= n <= 10^5

-10^4 <= nums1[i], nums2[i] <= 10^4

-10^4 <= diff <= 10^4

**Attempt 1: 2024-01-27**

**Solution 1: Brute Force (10 min, TLE 59/61)**

class Solution {

    public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

        int n = nums1.length;

        int[] arr = new int[n];

        // nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff

        // -> nums1[i] - nums2[i] <= nums1[j] - nums2[j] + diff

        // -> arr[i] <= arr[j] + diff

        for(int i = 0; i < n; i++) {

            arr[i] = nums1[i] - nums2[i];

        }

        // Try to find for i < j, how many pairs 'arr[i] <= arr[j] + diff'

        long count = 0;

        for(int i = 0; i < n; i++) {

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

                if(arr[i] <= arr[j] + diff) {

                    count++;

                }

            }

        }

        return count;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(N)

**Solution 1: Binary Indexed Tree (720 min)**

class Solution {

    public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

        // Using 30001 for shift, 20001 is to handle the -10000 to +10000 range

        // (as -10000 - (+10000) = minimum = -20000, need +20001 to make it

        // positive) and additional 10000 to handle diff value of the range

        // -10000 to 10000

        int shift = 30001;

        // Initialize a Binary Indexed Tree (bit array) with a sufficient range

        // to cover all possible values, the most extreme case is potential max

        // value in nums1 minus potential min value in nums2 as 10000 - (-10000)

        // plus shift 30001 and plus max diff 10000, as 60001, need another '+ 1'

        // as BIT start from index = 1, and its guaranteed no OutOfIndexBound

        // exception to be 60003, to make it simple just given 100000

        //int[] bit = new int[60003];

        int[] bit = new int[100000];

        int n = nums1.length;

        long count = 0;

        // Loop through elements in the given arrays

        // Inside the for loop it designed to query first then update, which means

        // in Round 1, query as 'j' won't work, only update as 'i' will work, and

        // in further Rounds(start from Round 2), query as 'j' will naturally have

        // 1 index ahead of 'i', since we always query on last round's update result,

        // so auto satisfy condition i < j

        for(int i = 0; i < n; i++) {

            // Calculate the difference of elements at the same index

            int val = nums1[i] - nums2[i];

            // Query the BIT for the count of elements that are at most 'val + diff'

            // plus offset to handle negative indices

            count += query(bit, val + shift + diff);

            // Update the BIT to increment the count (always 1 as frequency)

            // for the value 'val' with an offset to handle negative indices

            update(bit, val + shift, 1);

        }

        return count;

    }

    private long query(int[] bit, int x) {

        long result = 0;

        // Padding with 1 since BIT index start from 1

        x++;

        while(x > 0) {

            result += bit[x];

            x -= (x & -x);

        }

        return result;

    }

    private void update(int[] bit, int x, int freq) {

        // Padding with 1 since BIT index start from 1

        x++;

        while(x <= bit.length) {

            bit[x] += freq;

            x += (x & -x);

        }

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Why we can use Binary Indexed Tree (BIT) to solve this question ?**

The condition to use Binary Indexed Tree is the question can be transformed into a cumulative sum or cumulative frequency problem, in our example here is a cumulative frequency problem.

The original inequality is: nums1[i]−nums1[j] <= nums2[i]−nums2[j]+diff, the inference of the original inequality change to below:

Define nums[i] = nums1[i] - nums2[i], then nums[j] = nums1[j] - nums2[j], then we have original inequality change to nums[i] <= nums[j] + diff.

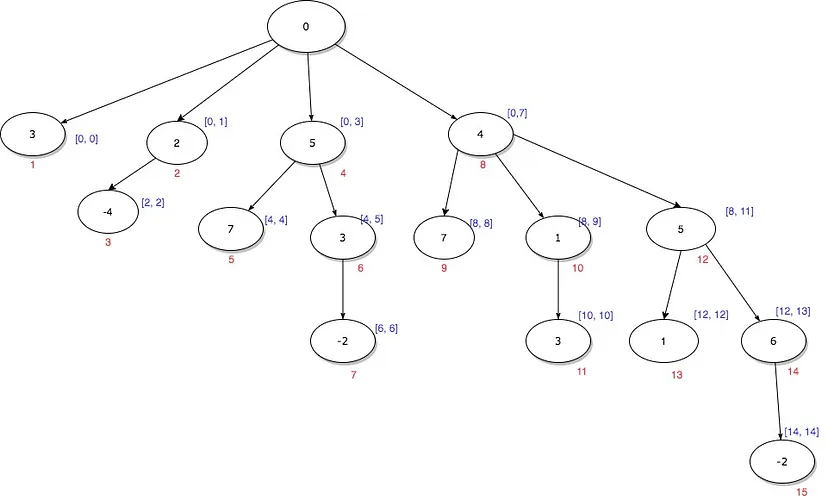
Based on this new inequality, what is the idea to transform it into a cumulative frequency problem ?

We have nums[j] and diff, since i < j, for each j, find the total number of ith indexes that satisfy equation, in another word, find the cumulative frequency of 'i' in BIT.

Use Binary indexed Tree (array representation) where each array index is nums[i] (with normalization since nums1[i] - nums2[i] maybe negative, but index cannot be negative), and each index's value is count (accumulative frequency) of indexes with value equal to nums[i], this is the critical point how BIT works, we don't simply only add 1 more frequency at index equal to nums[i], instead, since we have to treat the array as a BIT, based on BIT update method, we will also add 1 more frequency at all corresponding indexes (based on lowBit relation, let's say these are parent indexes), even these parent indexes not equal to nums[i], but since its a BIT, a parent index will include count (accumulative frequency) of nums[i] at their indexes, which gives a privilige in future's query method, since when we query the count (accumulative frequency) of a number, its actually query on our BIT (array representation)'s array index, and its not simply query out the only count at that index, based on BIT query method, a query on one index (in our example here a query on a certain number mapping to a query at an array index, since we define the array's index value equal to the potential query number) leading to a query on all its corresponding children indexes (based on lowBit relation, let's say these are children indexes), and we will add up all their count (accumulative frequency) in one run (a while loop).

For example, we have i < j, for given one 'j' we need to find all 'i' satisfy inequality: nums[i] <= nums[j] + diff, in a nomral array, we have to spend O(N) time to scan from 1 to j - 1, but here we build up a BIT as array representation, the BIT array index will be nums[i] value, the BIT array index's value will be count (accumulative frequency) of indexes with value equal to nums[i], are we still spend O(N) time to scan from 1 to j - 1 to find all satisfied 'i' ? No, we don't have to, because when we build up the BIT array, one index's parent index will include its count (accumulative frequency), which means when a query hit its parent index, it will easily query out its count (accumulative frequency) also, in our example here, if we know i = 0, 1, 2, 3 as nums[0], nums[1], nums[2], nums[3] all satisfy the inequation, will we take O(N) to find them and get total count (accumulative frequency) as 4 ? No, we don't have to, because after build up as a BIT, a query on j = 4 will easily include the range search on [0, 3], refer to below diagram:

**The Prefix Range of Each Node**



The original tree diagram, with added **blue** text

The text in **blue** represents the information this node will contain from **[start\_index, end\_index]**

For example, the node that represents *position 4* will contain information from index 0 to index 3 in the original array

The node that represents *position 5,* which we can reach from the node in the last bullet point,will contain information from: [0, 3] and [4, 4] => [0, 4]

**Its better to take a close look by go through a sample test code:**

import java.util.\*;

import java.util.stream.Collectors;

public class Solution {

class BinaryIndexedTree {

private int size; // Size of the original array

private int[] tree; // The Binary Indexed Tree array

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

public BinaryIndexedTree(int size) {

this.size = size;

tree = new int[size + 1]; // Since the BIT indices start at 1

}

// Method to compute the least significant bit (LSB)

public static final int lowbit(int x) {

return x & -x;

}

// Update method for the BIT, increments the value at index x by delta

public void update(int x, int delta) {

// Padding with 1 since BIT index start from 1

x++;

while (x <= size) {

tree[x] += delta; // Increment the value at index x

x += lowbit(x); // Move to the next index to update

}

}

// Query method for the BIT, gets the prefix sum up to index x

public int query(int x) {

int sum = 0;

// Padding with 1 since BIT index start from 1

x++;

while (x > 0) {

sum += tree[x]; // Add the value at index x to the sum

x -= lowbit(x); // Move to the previous sum index

}

return sum;

}

}

public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

// Initialize a Binary Indexed Tree with a sufficient range to cover possible values

//BinaryIndexedTree tree = new BinaryIndexedTree(100000);

BinaryIndexedTree tree = new BinaryIndexedTree(10);

long ans = 0; // Variable to store the count of valid pairs

int shift = 4;

// Loop through elements in the given arrays

for (int i = 0; i < nums1.length; ++i) {

int v = nums1[i] - nums2[i]; // Calculate the difference of elements at the same index

// Query the BIT for the count of elements that are at most 'v + diff' plus offset to handle negative indices

//ans += tree.query(v + diff + 40000);

ans += tree.query(v + diff + shift);

// Update the BIT to increment the count for the value 'v' with an offset to handle negative indices

//tree.update(v + 40000, 1);

tree.update(v + shift, 1);

}

return ans; // Return the total count of valid pairs

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums1 = new int[]{3,2,5};

int[] nums2 = new int[]{2,2,1};

int diff = 1;

long result = so.numberOfPairs(nums1, nums2, diff);

System.out.println(result);

}

}

**Step by Step**

We actually have a 'nums' array based on nums1 - nums2

int[] nums1 = new int[]{3,2,5};

int[] nums2 = new int[]{2,2,1};

-> nums = {1,0,4}, diff = 1

To handle negative, we also have shift = 4

We initialize a BIT tree with size = 11 which will cover all potential values for nums in this given example,

in our sample here each tree array index is nums[i], its 0 to 9 mapping on index (the max 9 coming from

nums[2] + diff + shift = 4 + 1 + 4), the value of each index as tree[index] is count (cumulative frequency)

of indexes with value equal to nums[i]

-> tree = {0,0,0,0,0,0,0,0,0,0,0} -> BIT always start at index = 1

^ ^ ^ ... ^ ^

0 1 2 8 9 -> bit tree array index start from 1 mapping to 0 - 9

Inside the for loop it designed to query first then update, which means in Round 1, query as 'j' won't work,

only update as 'i' will work, and in further Rounds(start from Round 2), query as 'j' will naturally have 1

index ahead of 'i', since we always query on last round's update result, so auto satisfy condition i < j

====================================================================================================

Round 1:

i = 0, v = nums1[0] - nums2[0] -> nums[0] = 3 - 2 = 1

tree.query(v + diff + shift) = tree.query(1 + 1 + 4) = tree.query(6) = 0

-> padding 1 as 6 + 1 = 7, start query index in tree begin from 7

-> because tree still empty

----------------------------------------------------------------------------------------------------

tree.update(v + shift, 1) = tree.update(1 + 4, 1) = tree.update(5, 1)

-> padding 1 as 5 + 1 = 6, start update index in tree begin from 6

-> we record count = 1(frequency) for Round 1's encountering nums[0] = 1 into BIT tree, prepare

for further query. Like what described in "Why we can use Binary Indexed Tree (BIT) to solve

this question ?", the 'update' of the count = 1(frequency) for nums[0] = 1 in BIT tree not only

happen for index = 6, but also include all its parent indexes to match BIT tree requirement and

give priviledge for further query to speed up to O(logN), it will finally include index = 6, 8

based on lowBit mechanism (x += x & -x), so both these two indexes mark its value as 1 means

"we find one nums[i] = 1 in Round 1", the update result will start impact on Round 2, since query

won't work in Round 1 based on empty BIT tree, that's the designation how query first update

second works

-> tree = {0,0,0,0,0,0,1,0,1,0,0} -> update tree[6] and tree[8] to 1

====================================================================================================

Round 2:

i = 1, v = nums1[1] - nums2[1] -> nums[1] = 2 - 2 = 0

tree.query(v + diff + shift) = tree.query(0 + 1 + 4) = tree.query(5) = 1

-> padding 1 as 5 + 1 = 6, start query index in tree begin from 6

-> sum += tree[6] -> sum = 1, x -= lowBit(x) = 4

-> sum += tree[4] -> sum keep 1, x -= lowBit(x) = 0 exit

-> it means for j = 1(remember how for loop designed to query first then update ? Now i = 1 actually

means j = 1, implicitly finish i -> j convert), based on BIT tree recorded count(cumulative frequency)

from index range = {0, j - 1} as {0, 0} we find one matched i = 0 satisfy the inequality by searching

on tree[6]'s covering range [0, 5]

----------------------------------------------------------------------------------------------------

tree.update(v + shift, 1) = tree.update(0 + 4, 1) = tree.update(4, 1)

-> padding 1 as 4 + 1 = 5, start update index in tree begin from 5

-> we record count = 1(frequency) for Round 2's encountering nums[1] = 0 into BIT tree, prepare

for further query. Like what described in "Why we can use Binary Indexed Tree (BIT) to solve

this question ?", the 'update' of the count = 1(frequency) for nums[1] = 0 in BIT tree not only

happen for index = 5, but also include all its parent indexes to match BIT tree requirement and

give priviledge for further query to speed up to O(logN), it will finally include index = 5, 6, 8

based on lowBit mechanism (x += x & -x), so both these three indexes mark its value as 1 means

"we find one nums[i] = 0 in Round 2", the update result will start impact on Round 3

-> tree = {0,0,0,0,0,1,2,0,2,0,0} -> update tree[5] to 1, tree[6] and tree[8] from 1 to 2

-> the count(cumulative frequency) stored on tree[6] and tree[8] increasing because that's how we sync up

the count change happen on BIT, its not merely change the target index itself(here as index = 5), it has

to reflect also on the parent indexes(here as index = 6 and 8), in future, when query on index = 6 or 8

based on lowBit mechanism, it will automatically go down till index = 5 and retrieve the count stored on

index = 5, this range search way is speed up a O(N) plain loop search to O(logN)

====================================================================================================

Round 3:

i = 2, v = nums1[2] - nums2[2] -> nums[2] = 5 - 1 = 4

tree.query(v + diff + shift) = tree.query(4 + 1 + 4) = tree.query(9) =

-> padding 1 as 9 + 1 = 10, start query index in tree begin from 10

-> sum += tree[10] -> sum = 0, x -= lowBit(x) = 8

-> sum += tree[8] -> sum = 2, x -= lowBit(x) = 0 exit

-> it means for j = 2(remember how for loop designed to query first then update ? Now i = 2 actually

means j = 2, implicitly finish i -> j convert), based on BIT tree recorded count(cumulative frequency)

from index range = {0, j - 1} as {0, 1} we find both i = 0, i = 1 satisfy the inequality by searching

on tree[10]'s covering range [0, 9]

----------------------------------------------------------------------------------------------------

tree.update(v + shift, 1) = tree.update(4 + 4, 1) = tree.update(8, 1)

-> padding 1 as 8 + 1 = 9, start update index in tree begin from 9

-> we record count = 1(frequency) for Round 3's encountering nums[2] = 4 into BIT tree, prepare

for further query. Like what described in "Why we can use Binary Indexed Tree (BIT) to solve

this question ?", the 'update' of the count = 1(frequency) for nums[2] = 4 in BIT tree not only

happen for index = 9, but also include all its parent indexes to match BIT tree requirement and

give priviledge for further query to speed up to O(logN), it will finally include index = 9, 10

based on lowBit mechanism (x += x & -x), so both these three indexes mark its value as 1 means

"we find one nums[i] = 4 in Round 3", the update result will start impact on Round 4 (in example here

no Round 4, terminate at Round 3)

-> tree = {0,0,0,0,0,1,2,0,2,1,1} -> update tree[9] and tree[10] to 1

====================================================================================================

So after all total count after Round 1(=0), Round 2(=1), Round 3(=2) is 0 + 1 + 2 = 3

**Refer to**

<https://algo.monster/liteproblems/2426>

**Problem Description**

In this problem, we are given two 0-indexed integer arrays nums1 and nums2, both of the same size n. We are also given an integer diff. Our goal is to find the number of pairs (i, j) that satisfy two conditions:

The indices i and j are within the bounds of the arrays, and i is strictly less than j (that is, 0 <= i < j <= n - 1).

The difference nums1[i] - nums1[j] is less than or equal to the difference nums2[i] - nums2[j] plus diff. In other words, the difference between the elements at positions i and j in nums1 is not greater than the difference between the corresponding elements in nums2 when diff is added to it.

We are asked to return the number of pairs that satisfy these conditions.

**Intuition**

The naive approach to solve this problem would be to check every possible pair (i, j) and count the number of pairs that satisfy the second condition. However, this approach would have a time complexity of O(n^2), which is inefficient for large arrays.

Instead, a more efficient solution is to use a Binary Indexed Tree (BIT), also known as a Fenwick Tree. This data structure is useful for problems that involve prefix sums, especially when the array is frequently updated.

The intuition behind the solution is to transform the second condition into a format that can be handled by BIT. Instead of directly comparing each pair (i, j), we can compute a value v = nums1[i] - nums2[i] for each element i and update the BIT with this value. When processing element j, we query the BIT for the number of elements i such that v = nums1[i] - nums2[i] <= nums1[j] - nums2[j] + diff. This simplifies the problem to counting the elements that have a value less than or equal to a certain value.

To accomplish this, the BIT needs to be able to handle both positive and negative index values. As such, a normalization step is added to map negative values to positive indices in the BIT.

The final solution iterates over the array only once, and for each element j, we get the count of valid i's using the BIT, which has O(log n) time complexity for both update and query operations. The result is a more efficient solution with a time complexity of O(n log n).

**Solution Approach**

The code solution uses a Binary Indexed Tree (BIT) to manage the frequencies of the differences between the nums1 and nums2 elements. Here's a step-by-step walkthrough of the implementation:

**Initialization:** Create a Binary Indexed Tree named tree which has a size that can contain all possible values after normalization (to take care of negative numbers).

**Normalization:** A fixed number (40000, in this case) is added to all the indices to handle negative numbers, as BIT cannot have negative indices. This supports the update and query operations with negative values by shifting the index range.

**Update Procedure:** For every element a in nums1 and b in nums2, calculate the difference v = a - b. Then, update the BIT (tree) with v - effectively counting this value.

**Query Procedure:** For the same values a and b, the query will search for the count of all values in the BIT that are less than or equal to v + diff. This is because we're looking for all previous indices i, where nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff.

**Collecting Results:** The query method returns the count of valid i indices leading up to the current index j, satisfying our pairs condition. This count is added to our answer ans for each j.

**Lowbit Function:** A method named lowbit is defined statically in the BinaryIndexedTree class. It is used to calculate the least significant bit that is set to 1 for a given number x. In the context of the BIT, this function helps in navigating to parent or child nodes during update and query operations.

The BIT handles frequencies of indices that correspond to the differences within nums1 and nums2. Instead of re-computing the differences between nums1 and nums2 elements exhaustively, we leverage the BIT's ability to efficiently count the elements so far that satisfy the given constraint relative to the current element being considered.

Overall, this approach takes advantage of the BIT's efficient update and query operations, leading to an O(n log n) time complexity, which is a dramatic improvement over the naive O(n^2) approach.

**Example Walkthrough**

Let's illustrate the solution approach with a small example. Consider the following:

nums1 = [1, 4, 2, 6]

nums2 = [3, 1, 3, 2]

diff = 2

Follow these steps to understand how the solution works:

**Initialization:** A Binary Indexed Tree (tree) would be created, but for simplicity in this example, we'll just mention it has been initialized and can handle the range of differences we'll encounter after normalization.

**Normalization:** We normalize differences by adding 40000 to indices to deal with possible negative numbers. (The numbers themselves won't change, just how we index them in the BIT.)

**Update Procedure:** We compute the differences for each element in nums1 and nums2:

For i=0: v = nums1[0] - nums2[0] = 1 - 3 = -2 (after normalization, index is -2 + 40000 = 39998)

Update tree at index 39998.

We'll repeat this for each element i.

Query Procedure: When examining each element j, we look for how many indices i have a difference v = nums1[i] - nums2[i] such that v <= nums1[j] - nums2[j] + diff.

For j=1: v = nums1[1] - nums2[1] = 4 - 1 = 3, so we query the tree for how many indices have a difference <= 3 + diff (remembering to normalize if necessary).

We repeat this for each element j, moving forward through the arrays.

Collecting Results:

For j=1, we query the tree and find there is 1 valid i (from index 0 with normalized difference 39998), so ans += 1.

We then move to j=2 and repeat the steps, updating the tree and querying for the range of valid i.

Lowbit Function: (Explained in a general sense, as it’s a bit abstract for a concrete example.)

After iterating through all the elements in nums1 and nums2, we would have found all pairs (i, j) where i < j and nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff. In this case, let's say we found two pairs that satisfy the conditions: (0, 1) and (0, 3), then our ans would be 2.

This walkthrough with a small example gives us a glimpse of how the BIT is used to efficiently manage and query the cumulative differences between the two arrays, optimizing the solution to an O(n log n) time complexity.

**Java Solution**

class BinaryIndexedTree {

    private int size; // Size of the original array

    private int[] tree; // The Binary Indexed Tree array

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

    public BinaryIndexedTree(int size) {

        this.size = size;

        tree = new int[size + 1]; // Since the BIT indices start at 1

    }

    // Method to compute the least significant bit (LSB)

    public static final int lowbit(int x) {

        return x & -x;

    }

    // Update method for the BIT, increments the value at index x by delta

    public void update(int x, int delta) {

        while (x <= size) {

            tree[x] += delta; // Increment the value at index x

            x += lowbit(x); // Move to the next index to update

        }

    }

    // Query method for the BIT, gets the prefix sum up to index x

    public int query(int x) {

        int sum = 0;

        while (x > 0) {

            sum += tree[x]; // Add the value at index x to the sum

            x -= lowbit(x); // Move to the previous sum index

        }

        return sum;

    }

}

class Solution {

    public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

        // Initialize a Binary Indexed Tree with a sufficient range to cover possible values

        BinaryIndexedTree tree = new BinaryIndexedTree(100000);

        long ans = 0; // Variable to store the count of valid pairs

        // Loop through elements in the given arrays

        for (int i = 0; i < nums1.length; ++i) {

            int v = nums1[i] - nums2[i]; // Calculate the difference of elements at the same index

            // Query the BIT for the count of elements that are at most 'v + diff' plus offset to handle negative indices

            ans += tree.query(v + diff + 40000);

            // Update the BIT to increment the count for the value 'v' with an offset to handle negative indices

            tree.update(v + 40000, 1);

        }

        return ans; // Return the total count of valid pairs

    }

}

**Time and Space Complexity**

The time complexity of the numberOfPairs method is O(n \* log m), where n is the length of the nums1 and nums2 lists and m is the value after offsetting in the BinaryIndexedTree (10^5 in this case). The log m factor comes from the operations of the Binary Indexed Tree methods update and query, since each operation involves traversing up the tree structure which can take at most the logarithmic number of steps in relation to the size of the tree array c.

The space complexity of the code is O(m), where m is the size of the BinaryIndexedTree which is defined as 10\*\*5. This is due to the tree array c that stores the cumulative frequencies, and it is the dominant term since no other data structure in the solution grows with respect to the input size n.

**Refer to**

<https://leetcode.com/problems/number-of-pairs-satisfying-inequality/solutions/2646651/java-fenwick-tree-solution-explained-o-n-log-n/>

**Intuition:**

nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff

can be refactored into

nums1[i] - nums2[i] <= nums1[j] - nums2[j] + diff

Create a new array nums3 where nums3[i] = nums1[i] - nums2[i]

Then equation will be

nums3[i] <= nums3[j] + diff

Since i < j, for each j, find the total number of ith indexes that satisfy equation.

**Use fenwick tree where each key is nums3[i], and each value is number of indexes with value equal to nums3[i].**

Therefore sum(nums3[j] + diff) would get total number of indexes that satisfy equation, with value from 0 <= value <= nums3[j] + diff.

**Edge cases:**

Since key can possibly be negative and arrays cannot have negative indexes, an offset of 50000 is added.

class Solution {

public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

int n = nums1.length;

int[] nums3 = new int[n];

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

nums3[i] = nums1[i] - nums2[i];

}

long count = 0;

int[] arr = new int[100000];

int offset = 50000;

Fenwick f = new Fenwick(arr);

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

int key = offset + nums3[j] + diff;

count += f.sum(key);

f.update(offset + nums3[j], 1);

}

return count;

}

}

public class Fenwick {

long[] bit;

public Fenwick(int arr[]) {

bit = new long[arr.length + 1]; // index starts from 1 not 0.

for (int i = 0; i < arr.length; i++) {

int index = i+1;

bit[index] += arr[i];

int parent = index + (index & -index); // index of parent

if (parent < bit.length) {

bit[parent] += bit[index];

}

}

}

// return sum of arr[0..index].

public long sum(int index) {

long totalSum = 0;

index = index + 1;

while (index > 0) {

totalSum += bit[index];

index -= index & (-index);

}

return totalSum;

}

public void update(int index, int val) {

// index in BIT[] starts from 1

index = index + 1;

while (index < bit.length) {

bit[index] += val;

index += index & (-index);

}

}

}

**Refer to**

<https://leetcode.com/problems/number-of-pairs-satisfying-inequality/solutions/2648104/2-o-nlogn-solutions-merge-sort-bit-java/>

**Approach**

To find the number of pairs we first need to rearrange the equation:

nums1[i] - nums1[j] <= nums2[i] - nums2[j] + diff

Move all i to the left and all j to the right:

nums1[i] - nums2[i] <= nums1[j] - nums2[j] + diff

Define nums[i] = nums1[i] - nums2[i]

Find the number of pairs i,j such that nums[i] <= nums[j] + diff

This can be done with a Binary Tree data structure like a Fenwick Tree - BIT (Solution 1) or a segment tree

This can also be done using a modified merge sort (Solution 2)

Time complexity O(nlogn)

Related problem: [493 Reverse pairs](https://leetcode.com/problems/reverse-pairs/)

Also read [this thread for a very detailed explanation of the principles](https://leetcode.com/problems/reverse-pairs/discuss/97268/General-principles-behind-problems-similar-to-%22Reverse-Pairs%22)

**Solution 1 with a Fenwick tree (BIT)**

Use SHIFT to map all negative numbers into a positive number range

Get the count of numbers already in the BIT satisfying x <= current number + diff

Then add the current number to the BIT

private int SHIFT = 30000;

public long numberOfPairs(int[] nums1, int[] nums2, int diff) {

int n = nums1.length;

BIT bit = new BIT(SHIFT \* 2);

long res = 0L;

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

int ndiff = nums1[i] - nums2[i];

res += bit.get(ndiff + diff + SHIFT);

bit.add(ndiff + SHIFT);

}

return res;

}

// BIT: both add & get O(logN)

private class BIT {

private int[] bit;

public BIT(int n) {

bit = new int[n+1];

}

public void add(int val) {

if (val <= 0) return;

for (int i = val; i < bit.length; i += (i & -i)) bit[i]++;

}

public int get(int val) {

int res = 0;

for (int i = val; i > 0; i -= (i & -i)) res += bit[i];

return res;

}

}

**Refer to**

<https://leetcode.com/problems/number-of-pairs-satisfying-inequality/solutions/3369458/detailed-explanation-3-approaches-merge-sort-segment-tree-bit-o-nlogn-multiple-languages/>

**Intuition & Approach for BIT/Fenwick Tree:**

The Expression A[i] - A[j] <= B[i] - B[j] + diff can be written as A[i] - B[i] - A[j] + B[j] <= diff

=> A[i] - B[i] - (A[j] - B[j]) <=diff

=> X[i] - X[j] <= diff, where X[k] = A[k] - B[k]

From the above expression we can see that we need to find a pair of (i,j) in X, such that it's less than or equal to diff. And X[k] can be calculated by doing A[k]-B[k] for all the array elements.

**We will be using BIT here to solve the problem, the BIT is intended for sums of number of occurrences of each X[i]. The idea is to for every j try to find all such i, such that X[i] - X[j] <= diff is satisfied. This will implicitly ensure i < j. So, for every X[j], we will try to find all X[i] in the range [1 to (X[j] + diff)]. Once the count is obtained. We update the BIT with X[j] so that it will get considered as a candidate for X[i], for any future values of X[j].**

Also, Since the input can contain both +ve & -ve numbers. We will shift all numbers to the right by adding some constant to all the numbers, such that the smallest number in the input range gets mapped to 1. second smallest in the input range gets mapped to 2 and so on.

**Here, we are using 30001 for shift, 20001 is to handle the -10000 to +10000 range and additional 10000 to handle diff value of the range -10000 to 10000.**

**Complexity**

**Time complexity:**

O(n + nlog(n)) = O(nlog(n)), n is the size of the input array.

**Space complexity:**

O(n), where n is the length of the input array.

**Code**

class Solution {

public long numberOfPairs(int[] A, int[] B, int diff) {

return new UsingBIT().numberOfPairs(A, B, diff);

}

}

class UsingBIT {

public long numberOfPairs(int[] A, int[] B, int diff) {

int SHIFT = 30001;

for (int i = 0; i < A.length; i++) {

A[i] = A[i] - B[i] + SHIFT;

}

int[] BIT = new int[SHIFT \* 2 + 1];

long count = 0;

for (int j : A) {

count += query(BIT, j + diff);

update(BIT, j);

}

return count;

}

private int query(int[] BIT, int n) {

int count = 0;

while (n >= 1) {

count += BIT[n];

n -= (n & -n);

}

return count;

}

private void update(int[] BIT, int n) {

while (n < BIT.length) {

BIT[n]++;

n += (n & -n);

}

}

}