2426. Number of Pairs Satisfying Inequality

Hard | Binary Indexed Tree | Segment Tree | Array | Binary Search | Divide and Conquer | Ordered Set | Merge Sort

Leetcode Link

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:

1. 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`).
2. 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:

1. **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).
2. **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.
3. **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.
4. **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`.
5. **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`.
6. **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:

1. **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.
2. **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.)
3. **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`.
4. **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.
5. **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`.
6. **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.

Python Solution

```python
class BinaryIndexedTree:
    def __init__(self, n):
        self.size = n
        self.tree_array = [0] * (n + 1)  # Initializing the BIT with n+1 elements

    @staticmethod
    def lowbit(x):
        # Static method to get the largest power of 2 that divides x
        return x & -x

    def update(self, index, delta):
        # Convert the original index range to 1-based for BIT operations
        index += 40000
        while index <= self.size:
            self.tree_array[index] += delta  # Update the tree_array by adding delta
            index += BinaryIndexedTree.lowbit(index)  # Move to the next index to update

    def query(self, index):
        # Similar conversion to 1-based index
        index += 40000
        sum = 0
        while index:
            sum += self.tree_array[index]  # Sum the elements in the tree_array
            index -= BinaryIndexedTree.lowbit(index)  # Move to the next index to query
        return sum


class Solution:
    def numberOfPairs(self, nums1, nums2, diff):
        tree = BinaryIndexedTree(10**5)  # Initialize the Binary Indexed Tree with a given size
        count_pairs = 0
        for a, b in zip(nums1, nums2):
            value = a - b
            count_pairs += tree.query(value + diff)  # Count pairs with difference less or equal to diff
            tree.update(value, 1)  # Update the Binary Indexed Tree
        return count_pairs

```

Java Solution

```java
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
    }
}

```

C++ Solution

```cpp
#include <vector>

using std::vector;

// BinaryIndexedTree supports efficient updating of frequencies
// and querying of prefix sums.
class BinaryIndexedTree {
public:
    int size;                   // Size of the array
    vector<int> treeArray;      // Tree array

    // Constructor initializes the Binary Indexed Tree with the given size.
    BinaryIndexedTree(int size)
        : size(size), treeArray(size + 1, 0) {}

    // Updates the tree with the given value 'delta' at position 'index'.
    void update(int index, int delta) {
        while (index <= size) {
            treeArray[index] += delta;
            index += lowBit(index); // Move to the next index to be updated
        }
    }

    // Queries the cumulative frequency up to the given position 'index'.
    int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += treeArray[index];
            index -= lowBit(index); // Move to the previous index to continue the sum
        }
        return sum;
    }

private:
    // Calculates and returns the least significant bit (low bit) of integer 'x'.
    int lowBit(int x) {
        return x & -x;
    }
};

class Solution {
public:
    // Calculates the number of pairs of elements in nums1 and nums2
    // such that nums1[i] - nums2[i] is not greater than the given 'diff'.
    long long numberOfPairs(vector<int>& nums1, vector<int>& nums2, int diff) {
        const int offset = 40000; // Used to offset negative values for BIT
        const int maxValue = 1e5; // Max value for which BIT is initialized

        // Initialize BinaryIndexedTree.
        BinaryIndexedTree tree(maxValue);

        long long pairsCount = 0; // Initialize count of pairs

        // Loop through all elements
        for (int i = 0; i < nums1.size(); ++i) {
            int valueDifference = nums1[i] - nums2[i];

            // Query the cumulative frequency for the range [0, v + diff + offset]
            pairsCount += tree.query(valueDifference + diff + offset);

            // Update the tree for 'valueDifference + offset' by 1
            tree.update(valueDifference + offset, 1);
        }

        // Returns the total count of valid pairs
        return pairsCount;
    }
};

```

Typescript Solution

```typescript
// Type definition for the tree array
type BinaryIndexedTreeArray = number[];

let treeSize: number; // Size of the array
let treeArray: BinaryIndexedTreeArray; // Tree array

// Initializes the Binary Indexed Tree with the given size.
function initializeBIT(size: number): void {
    treeSize = size;
    treeArray = Array(size + 1).fill(0);
}

// Updates the tree with the given value 'delta' at position 'index'.
function updateBIT(index: number, delta: number): void {
    while (index <= treeSize) {
        treeArray[index] += delta;
        index += lowBit(index); // Move to the next index to be updated
    }
}

// Queries the cumulative frequency up to the given position 'index'.
function queryBIT(index: number): number {
    let sum = 0;
    while (index > 0) {
        sum += treeArray[index];
        index -= lowBit(index); // Move to the previous index to continue summing
    }
    return sum;
}

// Calculates and returns the least significant bit (low bit) of integer 'x'.
function lowBit(x: number): number {
    return x & -x;
}

// Calculates the number of pairs of elements in nums1 and nums2
// such that nums1[i] - nums2[i] is not greater than the given 'diff'.
function numberOfPairs(nums1: number[], nums2: number[], diff: number): number {
    const offset = 40000; // Used to offset negative values for BIT
    const maxValue = 1e5; // Max value for which BIT is initialized

    // Initialize BinaryIndexedTree
    initializeBIT(maxValue);

    let pairsCount: number = 0; // Initialize count of pairs

    // Loop through all elements
    for (let i = 0; i < nums1.length; i++) {
        let valueDifference = nums1[i] - nums2[i];

        // Query the cumulative frequency for the range [0, valueDifference + diff + offset]
        pairsCount += queryBIT(valueDifference + diff + offset);

        // Update the tree for 'valueDifference + offset' by 1
        updateBIT(valueDifference + offset, 1);
    }

    // Returns the total count of valid pairs
    return pairsCount;
}

```

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`.