2426. Number of Pairs Satisfying Inequality

My Submissions (/contest/biweekly-contest-88/problems/number-of-pairs-satisfying-inequality/submissions/) Back to Contest (/contest/biweekly-contest-88/)

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.*

User Accepted:	1303
User Tried:	2778
Total Accepted:	1406
Total Submissions:	5012
Difficulty:	Hard

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`

Discuss (https://leetcode.com/problems/number-of-pairs-satisfying-inequality/discuss)

JavaScript

```
class Node {
    constructor(value) {
        this.parent_ = null;
        this.left_ = null;
        this.right_ = null;
        this.value_ = value;
        this.size_ = 1;
    }
    Update() {
        this.size_ = (this.left_ != null ? this.left_.size_ : 0) + (this.right_ != null ? this.right_.size_ : 0) + 1;
    }
    IsLeft() {
        return this.parent_ != null && this.parent_.left_ == this;
    }
    IsRight() {
        return this.parent_ != null && this.parent_.right_ == this;
    }
    IsRoot(guard = null) {
        return this.parent_ == guard;
    }

}

class SplayTree {
```

```
    constructor() {
        this.root_ = null;
        this.cmp_ = (x, y) => x >= y ? 0 : 1;
    }
    Zig(x) {
        let y = x.parent_;
        if (x.right_ != null) x.right_.parent_ = y;
        y.left_ = x.right_;
        x.right_ = y;
        if (y.IsLeft()) {
            y.parent_.left_ = x;
        } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
            y.parent_.right_ = x;
        }
        x.parent_ = y.parent_;
        y.parent_ = x;
        y.Update();
        x.Update();
    }

    // Zag:
    //   y               x
    //  / \             / \
    // A   x    --.    y   C
    //    / \          / \
    //   B   C        A   B
    Zag(x) {
        let y = x.parent_;
        if (x.left_ != null) x.left_.parent_ = y;
        y.right_ = x.left_;
        x.left_ = y;
        if (y.IsLeft()) {
            y.parent_.left_ = x;
        } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
            y.parent_.right_ = x;
        }
        x.parent_ = y.parent_;
        y.parent_ = x;
        y.Update();
        x.Update();
    }
    ZigZig(x) {
        this.Zig(x.parent_);
        this.Zig(x);
    }
    ZigZag(x) {
        this.Zig(x);
        this.Zag(x);
    }
    ZagZag(x) {
        this.Zag(x.parent_);
        this.Zag(x);
    }
    ZagZig(x) {
        this.Zag(x);
        this.Zig(x);
    }

    // Splay a "node" just under a "guard", which is default to splay to the "root_".
    Splay(node, guard = null) {
        while (!node.IsRoot(guard)) {
            if (node.parent_.IsRoot(guard)) {
                if (node.IsLeft()) {
                    this.Zig(node);
                } else {
                    this.Zag(node);
                }
            } else {
                if (node.parent_.IsLeft()) {
                    if (node.IsLeft()) {
                        this.ZigZig(node);
                    } else {
                        this.ZagZig(node);
                    }
                }
                else {
                    if (node.IsRight()) {
```

```
                        this.ZagZag(node);
                    } else {
                        this.ZigZag(node);
                    }
                }
            }
        }
        if (guard == null) this.root_ = node; // reset "root_" to "node".
    }
    LastNode(x) {
        this.Splay(x);
        let node = x.left_;
        if (node == null) return null;
        while (node.right_ != null) node = node.right_;
        this.Splay(node);
        return node;
    }
    NextNode(x) {
        this.Splay(x);
        let node = x.right_;
        if (node == null) return null;
        while (node.left_ != null) node = node.left_;
        this.Splay(node);
        return node;
    }
    Find(value) {
        return this.FindFirstOf(value);
    }
    FindFirstOf(value) {
        let node = this.root_, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp_(value, node.value_)) {
                node = node.left_;
            } else if (this.cmp_(node.value_, value)) {
                node = node.right_;
            } else {
                res = node;
                node = node.left_;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindLastOf(value) {
        let node = this.root_, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp_(value, node.value_)) {
                node = node.left_;
            } else if (this.cmp_(node.value_, value)) {
                node = node.right;
            } else {
                res = node;
                node = node.right_;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindRankOf(node) {
        this.Splay(node);
        return node.left_ == null ? 0 : node.left_.size_;
    }
    FindSuccessorOf(value) {
        let node = this.root_, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp_(value, node.value_)) {
                res = node;
                node = node.left_;
            } else {
                node = node.right_;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
```

```
    }
    FindPrecursorOf(value) {
        let node = this.root_, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp_(node.value_, value)) {
                res = node;
                node = node.right_;
            } else {
                node = node.left_;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindKth(rank) {
        if (rank < 0 || rank >= this.Size()) return null;
        let node = this.root_;
        while (node != null) {
            let left_size = node.left_ == null ? 0 : node.left_.size_;
            if (left_size == rank) break;
            if (left_size > rank) {
                node = node.left_;
            } else {
                rank -= left_size + 1;
                node = node.right_;
            }
        }
        this.Splay(node);
        return node;
    }
    NewNode(value) {
        return new Node(value);
    }
    DeleteNode(node) {
        node = null;
    }

    // ------------------------------- Public Usage -------------------------------------
    Size() {
        return this.root_ == null ? 0 : this.root_.size_;
    }
    IsEmpty() {
        return this.root_ == null;
    }

    // Insert an element into the container O(log(n))
    Insert(value) {
        if (this.root_ == null) {
            this.root_ = this.NewNode(value);
            return this.root_;
        }
        let node = this.root_;
        while (node != null) {
            if (this.cmp_(value, node.value_)) {
                if (node.left_ == null) {
                    node.left_ = this.NewNode(value);
                    node.left_.parent_ = node;
                    node = node.left_;
                    break;
                }
                node = node.left_;
            } else {
                if (node.right_ == null) {
                    node.right_ = this.NewNode(value);
                    node.right_.parent_ = node;
                    node = node.right_;
                    break;
                }
                node = node.right_;
            }
        }
        this.Splay(node);
        return node;
    }

    // Delete an element from the container if it exists O(log n)
```

```
    Delete(value) {
        let node = this.Find(value);
        if (node == null) return false;
        this.Splay(node);
        if (node.left_ == null) {
            this.root_ = node.right_;
            if (node.right_ != null) node.right_.parent_ = null;
            this.DeleteNode(node);
            return true;
        }
        if (node.right_ == null) {
            this.root_ = node.left_;
            if (node.left_ != null) node.left_.parent_ = null;
            this.DeleteNode(node);
            return true;
        }
        let last_node = this.LastNode(node);
        let next_node = this.NextNode(node);
        this.Splay(last_node);
        this.Splay(next_node, last_node);
        // After the above operations, the tree becomes:
        //      last_node
        //     /         \
        //    A       next_node
        //           /         \
        //         node         B
        // Then "next_node.left_" is "node".
        this.DeleteNode(next_node.left_);
        next_node.left_ = null;
        next_node.Update();
        last_node.Update();
        return true;
    }

    // Whether the splay tree contains value O(log n).
    Contains(value) {
        return this.CountOf(value) > 0;
    }

    // The number of ocurrences of value O(log n)
    CountOf(value) {
        let x = this.FindFirstOf(value);
        if (x == null) return 0;
        let rank_x = this.FindRankOf(x);
        let y = this.FindLastOf(value);
        let rank_y = this.FindRankOf(y);
        return rank_y - rank_x + 1;
    }

    // The number of elements strictly less than value O(log n)
    RankOf(value) {
        let x = this.FindPrecursorOf(value);
        return x == null ? 0 : this.FindRankOf(x) + 1;
    }

    // Get the k-th element (0-indexed) O(log n).
    Kth(rank) {
        let x = this.FindKth(rank);
        return x == null ? null : (x.value_);
    }

    // Find the smallest element that is strictly greater than value > , if it exists O(log n).
    SuccessorOf(value) {
        let node = this.FindSuccessorOf(value);
        return node == null ? null : (node.value_);
    }

    // Find the largest element that is strictly less than value < , if it exists O(log n).
    PrecursorOf(value) {
        let node = this.FindPrecursorOf(value);
        return node == null ? null : (node.value_);
    }

    // Get sorted values in the splay tree O(n).
    GetValues() {
        let res = [];
        const dfs = (x) => {
```

```
            if (x == null) return;
            dfs(x.left_);
            res.push(x.value_);
            dfs(x.right_);
        };
        dfs(this.root_);
        return res;
    }
}

const numberOfPairs = (a, b, diff) => {
    let tree = new SplayTree(), n = a.length, res = 0;
    let d = [];
    for(let i = 0; i < n; i++) d.push(a[i]-b[i]);
    for (let i = 0; i < n; i++) {
        let x = d[i], y = x + diff;
        res += tree.RankOf(y + 1);
        // pr(tree.RankOf(y+1));
        tree.Insert(x);
    }
    return res;
};
```

Custom Testcase | Use Example Testcases

Run | Submit

Submission Result: Accepted (/submissions/detail/877142049/) | More Details > (/submissions/detail/877142049/)

Share your acceptance!

1