🐆

Explanation

🧠 Problem: Contains Duplicate III

Given an integer array `nums`, and two integers `indexDiff` and `valueDiff`,

return `True` if there exist two distinct indices `i` and `j` such that:

- `|i - j| <= indexDiff`
- `|nums[i] - nums[j]| <= valueDiff`

Else return `False`.

🔍 Intuition

We are asked to check **two constraints simultaneously**:

1. **Indices constraint:** `|i - j| <= indexDiff`

   → means the two elements must be *close in position*.

2. **Value constraint:** `|nums[i] - nums[j]| <= valueDiff`

   → means the two elements must be *close in value*.

💡 Key Idea

We use a **sliding window** + **sorted data structure** to efficiently check for the value constraint within the last `indexDiff` elements.

- As we move through the array, we keep a **sorted set (`SortedSet`)** of the last `indexDiff` elements.
- For each new number `num`, we check:

  > Is there any number already in the window that lies in
  >
  > the range `[num - valueDiff, num + valueDiff]`?

If yes → return `True`.

⚙️ How do we check that efficiently?

We use `window.bisect_left(num - valueDiff)`

→ This gives the **index of the smallest element ≥ (num - valueDiff)**.

Then we check:

```
if pos < len(window) and abs(window[pos] - num) <= valueDiff:
    return True
```

This works because:

- `window` is sorted.
- The first element that is ≥ `(num - valueDiff)` is the **closest possible** to `num` on the left side.
- If even that element is within `valueDiff`, any others beyond it would be larger — hence no need to check more.

🧩 Why only `num - valueDiff`?

Let's think step by step:

For a given number `num`, we want:

```
num - valueDiff <= x <= num + valueDiff
```

- Using `bisect_left(num - valueDiff)` finds the **first candidate** in this range.
- Once we have that candidate, we check if it's ≤ `num + valueDiff` (i.e., within allowed range).
- If yes → return `True`.

We don't need to explicitly bisect `num + valueDiff` because:

- All elements beyond the found `pos` are ≥ `num - valueDiff`.
- We only need the **closest** one (since the set is sorted).

🪟 Why maintain window size?

We only care about pairs where `|i - j| <= indexDiff`.

That means we must **remove** the element that's too far from current `i`.

```
if len(window) > indexDiff:
    window.remove(nums[left])
    left += 1
```

This ensures that our `window` always contains at most `indexDiff` recent elements.

Math Explanation

$$num - valueDiff \leq x \leq num + valueDiff$$

$$\text{num} - \text{valueDiff} \le x \le \text{num} + \text{valueDiff}$$

is the *core inequality* behind the value difference condition, and understanding **how we derive it** will make this concept rock solid in your mind.

Let's break it down step-by-step — from **intuition → math → range**.

🎯 1️⃣ Start with the problem condition

We're told:

> The absolute difference between two numbers must be ≤ `valueDiff`.

So the core condition is:

$$|\,\text{num} - x\,| \le \text{valueDiff}$$

where:

- `num` = current element (`nums[i]`)
- `x` = some other element (`nums[j]`) in the nearby window

🧮 2️⃣ Expand the absolute value inequality

By definition of absolute value:

$$|A| \le k \implies -k \le A \le k$$

So replacing `A` with `(num - x)`:

$$-|valueDiff| \le num - x \le |valueDiff|$$

Since `valueDiff` is always non-negative,
we can drop the absolute on it:

$$-valueDiff \le num - x \le valueDiff$$

🔁 3️⃣ Isolate `x` (the number we're checking)

Now we'll rearrange this inequality to express it in terms of `x`.

Start from:

$$-valueDiff \leq num - x \leq valueDiff$$

Subtract `num` from all sides:

$$-valueDiff - num \leq -x \leq valueDiff - num$$

Multiply through by `-1` (this flips the inequalities):

$$num + valueDiff \geq x \geq num - valueDiff$$

Rewriting in standard ascending order:

$$\boxed{num - valueDiff \leq x \leq num + valueDiff}$$

🧠 4️⃣ Interpretation

This means:

> Any number `x` that lies between `num - valueDiff` and `num + valueDiff` will automatically satisfy the condition `|num - x| <= valueDiff`.

That's why our search range (for possible matches) becomes exactly:

csharp Copy code

```csharp
[num - valueDiff, num + valueDiff]
```

📊 5️⃣ Intuitive Example

Let's say:

```
num = 10
valueDiff = 3
```

Then:

```
num - valueDiff = 10 - 3 = 7
num + valueDiff = 10 + 3 = 13
```

So the valid range for x is:

```
[7, 13]
```

Meaning — any number x in that range (7, 8, 9, 10, 11, 12, 13)

will satisfy:

```
|10 - x| <= 3
```

✅ `|10 - 7| = 3 ≤ 3`

✅ `|10 - 12| = 2 ≤ 3`

✅ `|10 - 13| = 3 ≤ 3`

🔁 6️⃣ How this connects to the algorithm

Now that we know valid `x` must lie in `[num - valueDiff, num + valueDiff]`,

when we check the sorted window, we only need to look for any element in this range.

We do that efficiently using binary search:

```
pos = window.bisect_left(num - valueDiff)
```

→ Finds first element ≥ lower bound.

Then we verify:

```
if abs(window[pos] - num) <= valueDiff:
    return True
```

→ Ensures it's also ≤ upper bound.

🧩 7️⃣ Mini Example

```
num = 6, valueDiff = 2
valid range = [4, 8]
window = [1, 3, 5, 8, 10]
```

- `bisect_left(4)` → finds position of 5 (first ≥ 4)
- `abs(5 - 6) = 1 ≤ 2 ✅`

Even though we only searched using lower bound `4`,

the check `abs(...) ≤ 2` ensures the element is ≤ upper bound `8`.

How do we check if the number falls in the range:

> "bisect_left ensures we start from the first element that's not smaller than the lower limit,
> and the next condition ensures it doesn't exceed the upper limit — so we cover the full valid range."

🧩 Let's restate the setup

We are finding if there exists an `x` in the **sorted window** such that:

$$num - valueDiff \leq x \leq num + valueDiff$$

We do:

```
pos = window.bisect_left(num - valueDiff)
```

This gives us the **first index** where `x >= num - valueDiff`.

🧠 What `bisect_left` guarantees

- Every element **before** `pos` is `< (num - valueDiff)` → too small.
- Element **at** `pos` is the **smallest element that could still be inside the range**.

Now you said:

> The first element could be greater than lower bound but smaller than upper bound, right?

✅ Yes!

That's actually the *sweet spot* — that's exactly the case where a valid answer exists.

🧮 Let's check cases one by one

Case 1️⃣ — Element fits within both bounds

Example:

```
num = 10, valueDiff = 3 → valid range [7, 13]
window = [2, 5, 9, 14]
bisect_left(7) → pos = 2 → element = 9
```

- `9 ≥ 7` ✅ (within lower bound)
- `9 ≤ 13` ✅ (within upper bound)

→ This is a valid element. We can immediately return True. ✅

Case 2️⃣ — Element too large (bigger than upper bound)

```
num = 10, valueDiff = 3 → range [7, 13]
window = [2, 5, 14, 18]
bisect_left(7) → pos = 2 → element = 14
```

Check:

- `14 ≥ 7` ✅ (so it's ≥ lower bound)

- `14 ≤ 13` ❌ (fails upper bound)

Now here's the key reasoning:

> Every element after pos (14) will be ≥ 14 (because window is sorted).

So if 14 already exceeds the upper bound (13),

then **every next element (like 18, 21, …)** is even larger → all will also fail the upper bound.

✅ That's why we can safely stop here — no need to check further.

Case 3️⃣ — Element smaller than lower bound

Impossible, because by definition `bisect_left(num - valueDiff)` gives the *first* element that is **≥ lower bound**.

If `pos` points to an element smaller than that, bisect_left would have returned an earlier index.

So we never hit this case.

🎯 Putting it together

Situation	`window[pos]`	Meaning	What we do
Inside range	Between `[num - valueDiff, num + valueDiff]`	Valid	Return True
Greater than upper bound	> `num + valueDiff`	All later numbers will also be too large	Skip further check
Smaller than lower bound	Impossible due to `bisect_left` definition	—	—

🧠 Why don't we check to the right

Because the **sorted order** ensures:

- The first candidate is the **smallest number ≥ lower bound**.
- If that candidate is already too big for the upper bound, everything after it will be **even bigger**.
- Hence, **one check suffices** — we don't need to sweep to the right.

🔍 Intuitive analogy

Think of it like standing on a **number line**:

- You draw a box from `[num - valueDiff, num + valueDiff]`.
- You have sorted stones laid on the number line (`window`).
- `bisect_left` jumps you directly to the **first stone at or to the right of the box's left edge**.

Now:

- If that stone lands **inside** the box → ✅ success.
- If it lands **beyond** the box → all later stones will be even further right → ❌ fail.

So yes — your observation is correct:

> The element could be bigger than lower bound but smaller than upper bound.

That's exactly what we want to find — and that's why checking just that **first element ≥ lower bound** is enough to cover both conditions.

In Code

🧩 Step 1: What `bisect_left(num - valueDiff)` guarantees

When you call:

```
pos = window.bisect_left(num - valueDiff)
```

`bisect_left` Returns the **index of the first element** in `window` that is

$$x \geq (num - valueDiff)$$

So from this line alone, we already know:

> Every element before pos is too small (less than num - valueDiff).
>
> The element **at** `pos`, if it exists, is the **smallest one that might still be valid**.

🧩 Step 2: How the next condition ensures the full range check

Then we do:

```
if pos < len(window) and abs(window[pos] - num) <= valueDiff:
    return True
```

Now, this `abs(window[pos] - num) <= valueDiff` means:

$$|x - num| \leq valueDiff$$

which expands to:

$$num - valueDiff \leq x \leq num + valueDiff$$

So — that single check enforces **both the lower and upper bounds**.

✅ Combined effect

Together, these two lines ensure:

1. `bisect_left` → ensures `x` is **not smaller than the lower bound**.
2. The `abs(...) <= valueDiff` check → ensures `x` is **not greater than the upper bound**.

If both hold true → the number lies **within the valid range**.

If not → no element in the window satisfies both conditions.

🧠 Intuition summary (to remember for interviews or notes)

Step	What it checks	Why it works
`bisect_left(num - valueDiff)`	Finds the **first element ≥ lower bound**	Anything before is too small
`abs(window[pos] - num) <= valueDiff`	Ensures this element is also ≤ upper bound	If it fails, all after it are larger

✅ Hence, together they confirm whether there exists any element `x` such that

$$num - valueDiff \leq x \leq num + valueDiff$$

in the current sliding window.

So yes — in your own words:

> "bisect_left ensures we start from the first element that's not smaller than the lower limit,
>
> and the next condition ensures it doesn't exceed the upper limit — so we cover the full valid range."

That's 100% correct. 👏

🧮 Dry Run Example

```
nums = [1, 5, 9, 1]
indexDiff = 2
valueDiff = 3
```

Step 1

`window = []`

- num = 1

  bisect_left(1 - 3) = bisect_left(-2) → position 0

  No element to compare → add 1

  window = [1]

Step 2

num = 5

bisect_left(5 - 3 = 2) → position 1 (after 1)

No valid element at pos (window[1] doesn't exist)

→ add 5

window = [1, 5]

Step 3

num = 9

bisect_left(9 - 3 = 6) → position 2

No valid element at pos

→ add 9

window = [1, 5, 9]

Now len(window) = 3 > indexDiff (=2),

so remove nums[left]=1

→ window = [5, 9]

Step 4

num = 1

bisect_left(1 - 3 = -2) → position 0

window[0] = 5

abs(5 - 1) = 4 > valueDiff

→ no match

add 1

window = [1, 5, 9]

remove nums[left]=5

→ window = [1, 9]

No match found → return False.

🧩 Complexity

Operation	Complexity
`bisect_left`	O(log n)
`add` / `remove` (SortedSet)	O(log n)
Overall per element	O(log indexDiff)
Total	**O(n log indexDiff)**
Space	**O(indexDiff)**

🧱 Summary (Cheat Sheet)

Concept	Explanation
`SortedSet`	Keeps sliding window elements sorted
`bisect_left(num - valueDiff)`	Finds first element ≥ `num - valueDiff`
Condition check	See if that element ≤ `num + valueDiff`
Window size ≤ `indexDiff`	Ensures index difference condition
Return True	If both conditions satisfied
Time complexity	`O(n log indexDiff)`

🧠 Intuitive Thought Process

1. "I need to compare recent numbers only" → use sliding window.
2. "I need quick range lookup by value" → use sorted structure.
3. "Binary search lower bound for (num - valueDiff)" → candidate for smallest within range.
4. "If candidate within ±valueDiff" → condition satisfied → return True.

---

let's walk through this second example carefully.

This one is excellent because it helps **cement your understanding** of why the window slides and why we don't find any valid pair here.

🧮 Given

```
nums = [1, 5, 9, 1, 5, 9]
indexDiff = 2
valueDiff = 3
```

We must find indices `(i, j)` such that:

1. `i != j`
2. `|i - j| <= 2` ✅ (only last 2 elements matter)
3. `|nums[i] - nums[j]| <= 3`

⚙️ Setup

```
window = SortedSet()
left = 0
```

We'll track:

- The `window` contents
- What happens at each step
- Whether we find any `abs(window[pos] - num) <= 3`

🧩 Step-by-Step Execution

Step 1 — i = 0 → num = 1

```
window = []
num = 1
num - valueDiff = 1 - 3 = -2
pos = bisect_left(-2) → 0
```

No element yet.

➡️ Add `1` → `window = [1]`

✅ Window size = 1 ≤ 2

Step 2 — i = 1 → num = 5

```
window = [1]
num = 5
num - valueDiff = 2
pos = bisect_left(2) → 1
```

(pos = 1 means first element ≥ 2 would be after 1)

No valid element at pos (pos == len(window))

→ No candidate to check.

➡️ Add `5` → `window = [1, 5]`

✅ Window size = 2 ≤ 2

Step 3 — i = 2 → num = 9

```
window = [1, 5]
num - valueDiff = 9 - 3 = 6
pos = bisect_left(6) → 2
```

(pos = 2, since 6 > 5)

pos == len(window) → no element ≥ 6

No candidate found.

➡️ Add `9` → `window = [1, 5, 9]`

⚠️ Window size = 3 > indexDiff (=2)

➡️ Remove `nums[left]` = `nums[0] = 1`

➡️ `left += 1`

✅ Now window = `[5, 9]`

Step 4 — i = 3 → num = 1

```
window = [5, 9]
num - valueDiff = 1 - 3 = -2
pos = bisect_left(-2) → 0
```

Candidate at pos = 0 → window[pos] = 5

Check:

```
abs(5 - 1) = 4 > 3 ❌
```

No match.

➡️ Add `1` → `window = [1, 5, 9]`

⚠️ Window size = 3 > 2

➡️ Remove `nums[left]` = `nums[1] = 5`

➡️ `left += 1`

✅ Now window = `[1, 9]`

Step 5 — i = 4 → num = 5

```
window = [1, 9]
num - valueDiff = 2
pos = bisect_left(2) → 1 (first element ≥ 2 is 9)
```

Candidate = window[1] = 9

```
abs(9 - 5) = 4 > 3 ❌
```

No match.

➡️ Add `5` → `window = [1, 5, 9]`

⚠️ Window size = 3 > 2

➡️ Remove `nums[left]` = `nums[2] = 9`

➡️ `left += 1`

✅ Now window = `[1, 5]`

Step 6 — i = 5 → num = 9

```
window = [1, 5]
num - valueDiff = 6
```

```
pos = bisect_left(6) → 2
```

pos == len(window) → no element ≥ 6 → no candidate.

➡️ Add `9` → `window = [1, 5, 9]`

⚠️ Window size = 3 > 2

➡️ Remove `nums[left]` = `nums[3] = 1`

➡️ `left += 1`

✅ Now window = `[5, 9]`

No valid pair found throughout.

✅ Final Answer: **False**

🧠 Step Summary Table

i	num	window before	num - valueDiff	bisect pos	window[pos]	abs diff	Found?
0	1	[]	-2	0	—	—	❌
1	5	[1]	2	1	—	—	❌
2	9	[1,5]	6	2	—	—	❌
3	1	[5,9]	-2	0	5	4	❌
4	5	[1,9]	2	1	9	4	❌
5	9	[1,5]	6	2	—	—	❌

🧩 Why this return False

Even though the same numbers repeat (`1, 5, 9`),

their **index distance is always > 2**, so they **fall outside the sliding window**.

Value	Indices	Distance	Allowed?
1	0, 3	3	❌ (must be ≤ 2)
5	1, 4	3	❌
9	2, 5	3	❌

So none of the duplicates satisfy `|i - j| <= indexDiff`.

💡 Summary Takeaways

Concept	Explanation
Sliding window	Keeps last `indexDiff` elements only → enforces `
SortedSet	Keeps window sorted → binary search in O(log k)
bisect_left(num - valueDiff)	Finds first candidate ≥ lower bound
abs(window[pos] - num) <= valueDiff	Checks full range `[num - valueDiff, num + valueDiff]`
Result	No candidate satisfies both index & value conditions → `False`

let's work through your example `nums = [1, 2, 3, 1]` step-by-step **exactly as the algorithm runs.**

We'll see **how the sliding window forms**, how `bisect_left` finds candidates, and when we return `True`.

🧮 Given:

```
nums = [1, 2, 3, 1]
indexDiff = 3
valueDiff = 0
```

🎯 Conditions to satisfy

We need to find **i, j** such that:

1. `i != j`
2. `|i - j| <= 3` ✅ (within the last 3 elements)
3. `|nums[i] - nums[j]| <= 0` → means **nums[i] == nums[j]**

🧠 Algorithm recap:

We use:

```
window = SortedSet()
left = 0
```

Then for each `num` in `nums`:

1. Find position: `pos = bisect_left(num - valueDiff)`
2. Check if `abs(window[pos] - num) <= valueDiff`
3. Add current `num` to window.
4. If `len(window) > indexDiff`, remove the leftmost element (`nums[left]`).

🧩 Step-by-step Execution

Step 1 — i = 0 → num = 1

```
window = []
num = 1
```

👉 `num - valueDiff = 1 - 0 = 1`

```
pos = bisect_left(1) → pos = 0 (since window is empty)
```

No element to check yet.

➡️ Add `1` to window → `window = [1]`

✅ Window size = 1 ≤ 3 → nothing to remove.

Step 2 — i = 1 → num = 2

```
window = [1]
num = 2
```

```

```

👉 `num - valueDiff = 2 - 0 = 2`

```
pos = bisect_left(2) → pos = 1
```

Check if `pos < len(window)` → 1 < 1 ❌ (no)

→ No candidate.

➡️ Add `2` to window → `window = [1, 2]`

✅ Window size = 2 ≤ 3 → nothing to remove.

Step 3 — i = 2 → num = 3

```
window = [1, 2]
num = 3
```

👉 `num - valueDiff = 3 - 0 = 3`

```
pos = bisect_left(3) → pos = 2
```

Check if `pos < len(window)` → 2 < 2 ❌ (no)

➡️ Add `3` to window → `window = [1, 2, 3]`

✅ Window size = 3 ≤ 3 → nothing to remove.

Step 4 — i = 3 → num = 1

```
window = [1, 2, 3]
num = 1
```

👉 `num - valueDiff = 1 - 0 = 1`

```
pos = bisect_left(1) → pos = 0
```

✅ Check:

```
abs(window[pos] - num) = abs(1 - 1) = 0 ≤ 0 ✅
```

🎉 Condition satisfied → **return True**

✅ Final Answer

Output: `True`

🔍 Step Summary Table

i	num	window before	num - valueDiff	bisect_left pos	window[pos]	abs diff	Found?
0	1	[]	1	0	—	—	❌
1	2	[1]	2	1	—	—	❌
2	3	[1,2]	3	2	—	—	❌
3	1	[1,2,3]	1	0	1	0	✅

💡 Intuition Recap

- **Sliding window** = only last 3 elements (`indexDiff = 3`)
- **SortedSet** = fast binary search (`O(log k)`)
- **bisect_left(num - valueDiff)** = find first possible candidate ≥ `num - valueDiff`
- **abs check** = confirms if it's within `[num - valueDiff, num + valueDiff]`

Since when `i=3`, we find a duplicate `1` already in window (and within 3 indices),the algorithm returns **True**.

InShort

🧠 Understanding the Trick Behind `bisect_left` in the "Nearby Almost Duplicate" Problem

While solving the problem

> "Find if there exist indices (i, j) such that
>
> • |i - j| ≤ indexDiff
>
> • |nums[i] - nums[j]| ≤ valueDiff"

I realized the entire logic revolves around one simple range:

$$num - valueDiff \leq x \leq num + valueDiff$$

Where,

> Lower Bound: `num - valueDiff`

> Upper Bound: `num + valueDiff`

The question is — how do we efficiently check if any number `x` in our sliding window falls inside this range?

That's where this line does the magic:

```
pos = window.bisect_left(num - valueDiff)
```

🔷 `bisect_left` returns the first element in the sorted window that's **≥ (num - valueDiff)**. ie, `x ≥ (num - valudDiff)`

🔷 Then we check if that element also satisfies the upper bound:

```
if pos < len(window) and abs(window[pos] - num) <= valueDiff:
```

If true, the number lies within `[num - valueDiff, num + valueDiff]`.

If it's already greater than the upper bound, every next number (since the list is sorted) will only be larger — so we stop there.

In essence:

> bisect_left anchors us at the lower limit,
>
> the `abs` check enforces the upper limit,
>
> and together, they efficiently confirm whether any element lies inside the valid range.

Such small observations make these algorithms beautifully elegant 🔥
