# Detecting Duplicates in Arrays: Hashing Patterns in Action

Welcome, future coding interview champion! Today, we're going to explore one of the most common and versatile techniques in algorithm design: **using hash-based data structures to efficiently detect duplicates and satisfy certain constraints in arrays**.

This pattern is a must-have in your toolbelt. Whether you're trying to find if a list contains any duplicates, or you want to know if some duplicates are "close" together (by index or by value), you'll often reach for hash sets, hash maps, or even specialized trees for fast lookups.

Let's break this down:

- **What is the hashing pattern?**

It's about leveraging fast access in HashSet, HashMap, or similar structures. Hash-based structures allow you to check if an element exists, add or remove elements, and even track additional info (like indices) all in constant or near-constant time.

- **Why is it powerful?**

It lets you solve what would be quadratic brute-force searches in linear time. This is a huge win in interviews where performance matters!

- **When should you use it?**

Anytime you're asked to check for duplicates, find pairs with certain relationships, or maintain a "window" of recent elements.

**A Simple Example:**
Suppose you're asked:
"Given an array, return true if any value appears at least twice."

You could check every pair (slow!), but with a HashSet:

```
nums = [1, 2, 3, 1]
seen = set()
for num in nums:
    if num in seen:
        print(True)   # Found a duplicate
    seen.add(num)
```

Fast. Simple. Elegant.

Now, let's see how this core idea expands across three classic interview problems. Each builds on the last — let's dive in!

## Problem 1: Contains Duplicate

Leetcode 217: Contains Duplicate

**Problem Restated:**
Given an array of integers, return true if any value appears at least twice in the array, and false if every element is distinct.

### Example Walkthrough

**Input:** `[1, 2, 3, 1]`

**Output:** `True`

**Why?** The number `1` appears twice.

**Input:** `[1, 2, 3, 4]`

**Output:** `False`

**Why?** Every element is unique.

**Your Turn:**

Try a quick example: What should `[7, 8, 9, 10, 7, 11]` return?

## Pause and Think

Take a moment to try solving this on your own before reading the solution.

## Solution: HashSet for Fast Duplicate Detection

- Initialize an empty hash set.
- Iterate through each number in the array:
  - If the number is already in the set, you've found a duplicate — return `True`.
  - Otherwise, add the number to the set.
- If you finish the loop without finding a duplicate, return `False`.

Here's the code:

```
def containsDuplicate(nums):
    seen = set()
    for num in nums:
        if num in seen:
            return True  # Duplicate found
        seen.add(num)
    return False  # No duplicates found
```

## Time and Space Complexity

- **Time:** O(N), where N is the length of the array. Each lookup and insert in the set is O(1) on average.
- **Space:** O(N), for the hash set storing up to all unique elements.

## Walkthrough (with `[1, 2, 3, 1]`):

- Seen: `{}`
- Add 1 → Seen: `{1}`
- Add 2 → Seen: `{1, 2}`
- Add 3 → Seen: `{1, 2, 3}`

- See 1 again, already in `Seen` → Return `True`

**Try this yourself with `[7, 8, 9, 10, 7, 11]` and walk through the code!**

**Did you know this could also be solved by sorting first and looking for adjacent duplicates? Try implementing that after this!**

With this basic pattern down, let's step up the challenge. What if we care not just about duplicates, but about their relative positions?

## Problem 2: Contains Duplicate II

Leetcode 219: Contains Duplicate II

**Problem Restated:**
Given an array of integers and an integer `k`, return true if there are two distinct indices `i` and `j` in the array such that `nums[i] == nums[j]` and the absolute difference between `i` and `j` is at most `k`.

In other words, check if the same value appears at least twice, and those occurrences are within `k` indices of each other.

### Example Walkthrough

**Input:** `nums = [1, 2, 3, 1], k = 3`
**Output:** `True`
**Why?** The two `1`s are 3 indices apart (positions 0 and 3).

**Input:** `nums = [1, 0, 1, 1], k = 1`
**Output:** `True`
**Why?** The last two `1`s are right next to each other (indices 2 and 3).

**Input:** `nums = [1,2,3,1,2,3], k = 2`
**Output:** `False`
**Why?** No duplicates are within 2 spaces.

**Try this:**
What about `nums = [5, 6, 7, 5, 8], k = 2`?

### What's New Here?

Unlike the previous problem, now we don't just care about *any* duplicate — we care about *close* duplicates.
This means we need to remember more information: not just what values we've seen, but **where** we saw them.

### Pause and Think

Take a moment to try solving this on your own before reading the solution.

### Solution: HashMap to Track Last Seen Indices

- Use a hash map (`num_to_index`) to record the most recent index of each number.
- For each number, if we've seen it before and the current index minus the last index is at most `k`, return `True`.
- Otherwise, update the hash map with the current index for that number.

Here's the code:

```
def containsNearbyDuplicate(nums, k):
    num_to_index = {}
    for idx, num in enumerate(nums):
        if num in num_to_index and idx - num_to_index[num] <= k:
            return True  # Found a duplicate within k distance
        num_to_index[num] = idx  # Update last seen index
    return False
```

### Time and Space Complexity

- **Time:** O(N), one pass through the array.
- **Space:** O(N), for the hash map.

### Code Explanation

- `num_to_index` keeps track of the **last** index where each number was seen.
- When we see a number again, we check if the difference in indices is at most `k`.
- If so, we found a "nearby" duplicate!

### Let's Walk Through `[1, 0, 1, 1], k = 1`:

- idx 0, num 1: Not seen before. `num_to_index = {1: 0}`
- idx 1, num 0: Not seen before. `num_to_index = {1: 0, 0: 1}`
- idx 2, num 1: Seen at idx 0. Difference: 2 - 0 = 2 > 1. Update `num_to_index` to `{1: 2, 0: 1}`
- idx 3, num 1: Seen at idx 2. Difference: 3 - 2 = 1 <= 1 → **Return True**

**Try a dry-run with your earlier example: `[5, 6, 7, 5, 8], k = 2`!**

**As you might guess, you could also solve this problem with a sliding window and a HashSet for values in the window. See if you can write that after this!**

Ready for the final boss? What if, in addition to nearby indices, we care about *how close* the values are?

## Problem 3: Contains Duplicate III

Leetcode 220: Contains Duplicate III

**Problem Restated:**

Given an array of integers, and two integers `k` and `t`, return true if there are two distinct indices `i` and `j` such that:

- The absolute difference between `i` and `j` is at most `k`
- The absolute difference between `nums[i]` and `nums[j]` is at most `t`

So now, you're looking for *close* and *similar* values!

## Example Walkthrough

**Input:** `nums = [1,2,3,1], k = 3, t = 0`
**Output:** `True`
**Why?** The two `1`s are within 3 indices, and the value difference is 0 (identical).

**Input:** `nums = [1, 5, 9, 1, 5, 9], k = 2, t = 3`
**Output:** `False`
**Why?** Every duplicate is too far apart in index.

**Try this:**
What about `nums = [2, 4, 6, 8], k = 2, t = 2`?

## What's New Here?

This is a step up from Problem 2:

- In Problem 2, we only cared about nearby indices and *exact* duplicates.
- In Problem 3, we care about nearby indices and *values that are close* (at most `t` apart).

We need a data structure that can quickly check:
"Is there a value in the last `k` elements whose value is within `t` of the current value?"

HashSets and HashMaps can't do efficient "range queries", but **balanced search trees** (like Python's `SortedList` from `sortedcontainers` module, or Java's `TreeSet`) can.

## Pause and Think

Take a moment to try solving this on your own before reading the solution.

## Solution: Sliding Window + Balanced BST (TreeSet/SortedList)

We'll use a `SortedList` (from `sortedcontainers` package) to keep a sliding window of the last `k` numbers. For each number:

- Use `bisect_left` to find the position where `num - t` would go in the window.
- If that position is within the window and the value at that index is within `t` of `num`, return `True`.

- Add the current number to the window.
- If the window size exceeds `k`, remove the oldest number.

**Note:**

If you don't have `sortedcontainers`, you can mention the logic here and code for Leetcode's Java `TreeSet` is similar.

## Python Code (uses `SortedList`):

```
from sortedcontainers import SortedList  # pip install sortedcontainers

def containsNearbyAlmostDuplicate(nums, k, t):
    if k <= 0 or t < 0:
        return False  # edge case

    window = SortedList()
    for i, num in enumerate(nums):
        # Find smallest number >= num - t
        pos = window.bisect_left(num - t)
        # Check if such number exists and is within t of num
        if pos < len(window) and abs(window[pos] - num) <= t:
            return True
        window.add(num)
        # Keep window size at most k
        if i >= k:
            window.remove(nums[i - k])
    return False
```

## Time and Space Complexity

- **Time:** O(N log k), for each element, we may insert and remove from the balanced BST (SortedList) of size k.
- **Space:** O(k), for the window.

## Code Explanation

- `window` maintains the set of last `k` numbers, always sorted.
- For the current `num`, we want to check if there's a number in `window` within `t` of `num`.
- `bisect_left(num - t)` gives us the earliest possible candidate.
- We check if that candidate is close enough (difference <= t).
- Slide the window by removing the (i - k)th element as we go.

## Example Step-by-Step: `[1, 2, 3, 1], k = 3, t = 0`

- i=0, num=1, window=[], add 1 → window=[1]
- i=1, num=2, window=[1], bisect_left(2-0=2)=1 (out of bounds), add 2 → window=[1,2]
- i=2, num=3, window=[1,2], bisect_left(3)=2, add 3 → window=[1,2,3]
- i=3, num=1, window=[1,2,3], bisect_left(1)=0, window[0]=1, abs(1-1)=0<=0 → **Return True**

**Try this on your own:**

```
nums = [2, 4, 6, 8], k = 2, t = 2
```

**While bucket sort can also be used for this problem for O(N) expected time, stick to this TreeSet/SortedList solution to master the pattern! Try bucket sort as a personal challenge.**

# Summary and Next Steps

Congrats! You've now seen how **hashing and window-based data structures** help you solve a whole class of "duplicate detection" problems, from the simplest to the sneakiest:

- **Problem 1:** Basic duplicate detection with HashSet.
- **Problem 2:** Duplicate detection with index constraints using HashMap.
- **Problem 3:** Duplicate detection with both index and value constraints using TreeSet/SortedList.

These problems are grouped because they all use *fast lookups* to avoid slow brute-force comparisons, evolving the technique as constraints increase.

**Key Patterns and Variations:**

- Use a HashSet to track seen values for O(1) duplicate checks.
- Use a HashMap to track both values and their positions for index-based constraints.
- Use a balanced BST when you need to check for "close values" in a range.

**Common Mistakes to Avoid:**

- Forgetting to update or remove entries when sliding the window (in Problems 2 and 3).
- Not handling edge cases: k=0, negative t.
- Using brute-force O(N^2) solutions when a hash-based structure can do it in O(N).
- Not considering integer overflow when dealing with value differences in Problem 3 (important in languages like Java or C++).

**Action List for You:**

- Go back and implement all 3 problems yourself from scratch.
- Practice the variant solutions (sorting for Problem 1, sliding window HashSet for Problem 2, bucket sort for Problem 3).
- Check out top Leetcode discussions and see how others handled edge cases or optimized further.
- If you get stuck on a harder problem, that's perfectly normal. Review the patterns, try pen and paper with simple cases, and keep practicing!
- Look for similar problems: "Longest Substring Without Repeating Characters", "Minimum Window Substring", and others that use these patterns.

Remember: mastering these patterns lets you tackle a huge range of coding interview questions with confidence. Keep the learning momentum going, and see you at the next daily prep!

Happy coding!