**Java Implementation of Finding Five Repeated Elements**

We will implement the described algorithm using a **HashSet** to track unique numb**List** to store the five repeat

**Algorithm Breakdown:**

1. Create a **HashSet** (seen) to store distinct elements encountered so far.
2. Create a **List** (duplicates) to store the five repeated elements.
3. Iterate through array B:
   * If an element is **not in seen**, add it to seen.
   * If an element **is already in seen**, add it to duplicates (only if we haven't found 5 duplicates yet).
4. Stop once we find exactly **5 repeated elements**.
5. Return the list of repeated elements.

**Java Code:**

import java.util.\*;

public class FindFiveDuplicates {

public static List<Integer> findRepeatedElements(int[] B) {

Set<Integer> seen = new HashSet<>();

List<Integer> duplicates = new ArrayList<>();

for (int num : B) {

if (seen.contains(num)) {

if (duplicates.size() < 5) { // Ensure we only store 5 repeated elements

duplicates.add(num);

}

} else {

seen.add(num);

}

}

return duplicates; // List containing the five repeated elements

}

public static void main(String[] args) {

int[] B = {1, 3, 4, 2, 5, 6, 3, 7, 8, 9, 4, 2, 10, 5, 6, 11, 12}; // Example input

System.out.println("Repeated elements: " + findRepeatedElements(B));

}

}

**Complexity Analysis**

✅ **Time Complexity: O(n)**

* We iterate through the array **once**, and each look**HashSet** is **O(1)** on average.

✅ **Space Complexity: O(n-5) ≈ O(n)**

* The **set seen** stores up to n-5 elements.
* The **list duplicates** stores **5 elements** (constant space).

**Example Execution**

**Input:**

int[] B = {1, 3, 4, 2, 5, 6, 3, 7, 8, 9, 4, 2, 10, 5, 6, 11, 12};

**Output:**

less

نسختحرير

Repeated elements: [3, 4, 2, 5, 6]

**Alternative Approach Using Sorting (O(n log n))**

If modifying the original array is allowed, we could **sort** the array and find consecutive duplicates.

import java.util.\*;

public class FindFiveDuplicatesSorted {

public static List<Integer> findRepeatedElements(int[] B) {

Arrays.sort(B); // O(n log n)

List<Integer> duplicates = new ArrayList<>();

for (int i = 1; i < B.length; i++) {

if (B[i] == B[i - 1] && (duplicates.isEmpty() || B[i] != duplicates.get(duplicates.size() - 1))) {

duplicates.add(B[i]);

if (duplicates.size() == 5) break;

}

}

return duplicates;

}

public static void main(String[] args) {

int[] B = {1, 3, 4, 2, 5, 6, 3, 7, 8, 9, 4, 2, 10, 5, 6, 11, 12};

System.out.println("Repeated elements: " + findRepeatedElements(B));

}

}

❌ **Slower (O(n log n))** due to sorting.  
✅ **No extra space (O(1))**, except for the list storing repeated elements.

**Best Choice?**

| **Method** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| **HashSet Approach** | **O(n)** | **O(n)** | Fastest, simplest. Best for large n. |
| **Sorting Approach** | **O(n log n)** | **O(1)** | Uses no extra space, but sorting is slower. |

For large datasets, **the HashSet approach is optimal (O(n))**.