**1. Using a HashSet (O(n) Time, O(n) Space)**

This method uses a **HashSet** to track seen numbers and returns the first repeated one.

import java.util.HashSet;

public class FindDuplicate {

public static int findRepeatedElement(int[] A) {

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

for (int num : A) {

if (seen.contains(num)) {

return num; // Found the duplicate

}

seen.add(num);

}

return -1; // Should never happen given problem constraints

}

public static void main(String[] args) {

int[] A = {1, 3, 4, 2, 2};

System.out.println("Repeated element: " + findRepeatedElement(A)); // Output: 2

}

}

✅ **Pros:** Simple and efficient.  
❌ **Cons:** Uses extra space **O(n)**.

**2. Sorting Approach (O(n log n) Time, O(1) Space)**

This approach sorts the array and looks for adjacent duplicates.

import java.util.Arrays;

public class FindDuplicate {

public static int findRepeatedElement(int[] A) {

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

for (int i = 0; i < A.length - 1; i++) {

if (A[i] == A[i + 1]) {

return A[i]; // Found duplicate

}

}

return -1; // Should never happen given problem constraints

}

public static void main(String[] args) {

int[] A = {1, 3, 4, 2, 2};

System.out.println("Repeated element: " + findRepeatedElement(A)); // Output: 2

}

}

✅ **Pros:** No extra space used.  
❌ **Cons:** Sorting takes **O(n log n)** time, which is slower than **O(n)** solutions.

**3. Floyd’s Cycle Detection (O(n) Time, O(1) Space)**

This method treats the array as a linked list and finds the duplicate using **Tortoise and Hare (Floyd’s cycle detection algorithm)**.

public class FindDuplicate {

public static int findRepeatedElement(int[] A) {

int slow = A[0];

int fast = A[A[0]];

// Phase 1: Detect cycle

while (slow != fast) {

slow = A[slow];

fast = A[A[fast]];

}

// Phase 2: Find the entrance to the cycle (repeated number)

slow = 0;

while (slow != fast) {

slow = A[slow];

fast = A[fast];

}

return slow;

}

public static void main(String[] args) {

int[] A = {1, 3, 4, 2, 2};

System.out.println("Repeated element: " + findRepeatedElement(A)); // Output: 2

}

}

✅ **Pros:** **O(n) time and O(1) space**, optimal for large datasets.  
❌ **Cons:** More complex to understand and implement.

**Comparison of Approaches**

| **Method** | **Time Complexity** | **Space Complexity** | **Pros** | **Cons** |
| --- | --- | --- | --- | --- |
| **HashSet (Set Approach)** | **O(n)** | **O(n)** | Simple & fast | Extra space |
| **Sorting Approach** | **O(n log n)** | **O(1)** | No extra space | Slower due to sorting |
| **Floyd’s Cycle Detection** | **O(n)** | **O(1)** | Best for space efficiency | Harder to implement |

**Best Choice?**

* **For simplicity:** Use **HashSet (O(n), O(n))**.
* **For space efficiency:** Use **Floyd’s Cycle Detection (O(n), O(1))**.