

■ Single Element in Sorted Array

Find the one lonely number that appears exactly once

● Binary Search

■ MEDIUM

■ What You Will Learn

- How pairs break the index pattern after the single element
- Why even/odd index positions reveal which side the single element is on
- A beautiful parity-based binary search

■ Intuition First

The array has every element twice, except one. `[1,1,2,3,3,4,4,5,5]`. Key observation: before the single element, pairs sit at (even, odd) indices. After the single element, pairs shift to (odd, even). At any `mid`, check if its pair is to the right or left — this tells you which side the singleton is on.

■ Index Parity Trick

In a pair-dominated array, `arr[mid]` should equal `arr[mid^1]` (its pair partner). `mid^1` flips the last bit: if `mid` is even, `mid^1 = mid + 1`; if `mid` is odd, `mid^1 = mid - 1`. If `arr[mid] == arr[mid^1]`, the single element is to the RIGHT. If not, the single element is to the LEFT (or IS `mid`).

■ Visual Explanation

```
Array: [1, 1, 2, 3, 3, 4, 4, 5, 5]
Index: 0  1  2  3  4  5  6  7  8
```

Before single element (index 2 = value '2'):
pairs are at indices (0,1), (3,4), (5,6), (7,8)
← even-odd, even-odd odd-even, odd-even →
 ^2 is here^

Step 1: L=0, R=8, mid=4 → nums[4]=3
mid=4 (even) → check mid+1=5 → nums[5]=4 ≠ 3
pairs broken here → single element is on LEFT → R=4

Step 2: L=0, R=4, mid=2 → nums[2]=2
mid=2 (even) → check mid+1=3 → nums[3]=3 ≠ 2
pairs broken → R=2

Step 3: L=0, R=2, mid=1 → nums[1]=1
mid=1 (odd) → check mid-1=0 → nums[0]=1 == 1
pairs intact → single element on RIGHT → L=2

Step 4: L=2, R=2 → nums[2]=2 → FOUND ✓

■ Java Code

```
1  class Solution {
2  public int singleNonDuplicate(int[] nums) {
3      int left = 0, right = nums.length - 1;
4
5      while (left < right) {
6          int mid = left + (right - left) / 2;
7
8          // Make mid even (so we compare with mid+1)
9          if (mid % 2 == 1) mid--;
10
11         // If pair is intact, single element is to the right
12         if (nums[mid] == nums[mid + 1]) {
13             left = mid + 2;
14         } else {
15             // Pair broken, single element at mid or to the left
16             right = mid;
17         }
18     }
19     return nums[left];
20 }
21 }
```

■ Complexity

Complexity	Value	What it means
Time	$O(\log n)$	Halving the search space each iteration
Space	$O(1)$	No extra data structures needed

■ Common Mistakes

■ Mistake

Searching on odd-length arrays without ensuring mid is even. Always normalize mid to even before comparing.

■ Pro Tips

■ XOR Trick

You can also solve this in $O(n)$ with XOR: XOR all elements. Since duplicates cancel out ($a \oplus a = 0$), only the single element remains. But binary search achieves $O(\log n)$ which is better!

■ Quick Revision

✓	Pairs before single element: (even,odd) index pattern
✓	Pairs after single element: (odd,even) index pattern
✓	Force mid to be even, then compare with mid+1
✓	If $\text{nums}[\text{mid}] == \text{nums}[\text{mid}+1]$: single is right $\rightarrow \text{left} = \text{mid}+2$
✓	If $\text{nums}[\text{mid}] != \text{nums}[\text{mid}+1]$: single is here or left $\rightarrow \text{right} = \text{mid}$