📘 Single Element in a Sorted Array — Complete Documentation

📌 1. Problem Statement

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Your task is to return that single element.

- You must write an algorithm with:
  - Time complexity: O(log n)
  - Space complexity: O(1)

🔢 Examples:

Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2

Input: nums = [3,3,7,7,10,11,11]
Output: 10

💡 2. Intuition

Since the array is sorted and most elements appear in pairs, any disruption in the pairing sequence must be caused by the single element. Using this idea, we can perform binary search to efficiently locate the unpaired element.

🔍 3. Key Observations

- In a properly paired sorted array:
  - Before the single element: First element of the pair occurs at an even index.
  - After the single element: First element of the pair occurs at an odd index.
- Using this pattern, we can narrow down the search space using binary search.

⚙ 4. Approach

- Initialize low = 0 and high = len(nums) - 1.
- Use binary search:
  - Compute mid = (low + high) // 2.
  - Ensure mid is even (because pairs start from even index).
  - If nums[mid] == nums[mid + 1], the unique element lies to the right.
  - Else, it lies to the left (or could be mid).
- Loop continues until low == high.
- Return nums[low] as the unique element.

⚠ 5. Edge Cases

- Array with only one element: [7] → return 7
- Unique element at the beginning: [2,3,3,4,4,5,5]
- Unique element at the end: [1,1,2,2,3,3,9]

📊 6. Complexity Analysis

□ Time Complexity:

- O(log n) due to binary search.

💾 Space Complexity:

- O(1) since we're using constant space.

🔄 7. Alternative Approaches

1. Linear Scan (O(n) Time, O(1) Space)

Loop through the array in steps of 2 and compare nums[i] and nums[i + 1]. If they are not equal, return nums[i].

2. Bit Manipulation (O(n) Time, O(1) Space)

Use XOR: res = 0; for num in nums: res ^= num; return res.

Doesn't work here due to requirement of O(log n) time.

✅ 8. Test Cases

Input	Output	Description
[1,1,2,3,3,4,4,8,8]	2	Unique element in the middle
[3,3,7,7,10,11,11]	10	Unique element just before the end
[1,2,2,3,3,4,4]	1	Unique element at the beginning
[1,1,2,2,3,3,4]	4	Unique element at the end
[5]	5	Only one element in array

□ 9. Final Thoughts

- This problem is a classic example of applying binary search over index patterns instead of values.
- Key to the solution is recognizing how the parity of indices changes around the single element.
- Always make mid even to ensure you're comparing the start of a pair.