

■ Reverse an Array Using Recursion

Two-pointer thinking + recursive shrinking = elegant reversal

● Recursion

■ EASY

■ What You Will Learn

- How to pass indices as parameters to guide recursion
- Swap + recurse on the smaller subarray inside
- Why recursion and two-pointer thinking overlap here
- The 'shrinking boundaries' recursive pattern

■ Intuition First

Reversing [1,2,3,4,5]: swap the outermost pair (1 and 5), then recursively reverse the inner part [2,3,4]. That inner part: swap outermost (2 and 4), recurse on [3]. Single element — done! We're solving a smaller version of the same problem each time.

■ Shrinking Boundaries Pattern

Pass left and right indices as parameters. Swap `arr[left]` and `arr[right]`. Then recurse with `left+1` and `right-1`. Base case: when `left >= right` (subarray has 0 or 1 elements). This 'sandwich' pattern appears in palindrome check, quicksort partition, and more!

■ Visual — Recursive Reversal

```
Input: [1, 2, 3, 4, 5]
      L           R
Step 1: swap(1,5) → [5, 2, 3, 4, 1]
      L           R
Step 2: swap(2,4) → [5, 4, 3, 2, 1]
      L   R
Step 3: L==R (middle), base case → STOP

Result: [5, 4, 3, 2, 1] ✓

Recursion tree:
reverse(0,4) → swaps 1&5, calls reverse(1,3)
  reverse(1,3) → swaps 2&4, calls reverse(2,2)
    reverse(2,2) → L==R, returns (done!)
```

■ Java Code

```
1  class Solution {
2  public void reverse(int[] arr, int left, int right) {
3  // BASE CASE: no more elements to swap
```

```

4     if (left >= right) return;
5
6     // Swap the outermost elements
7     int temp = arr[left];
8     arr[left] = arr[right];
9     arr[right] = temp;
10
11    // Recurse on the inner subarray
12    reverse(arr, left + 1, right - 1);
13 }
14
15 // Call it like: reverse(arr, 0, arr.length - 1);
16 }

```

■ Time & Space Complexity

Type	Value	What It Means
Time	$O(n)$	$n/2$ swaps — each element touched once
Space	$O(n/2) = O(n)$	Recursion depth = $n/2$ stack frames

■ Common Mistakes

■ Wrong Base Case

Using `if(left==right)` misses the even-length case where `left>right`. Always use `left>=right`.

■ Not Passing Indices

Writing `reverse(arr)` and using a global counter loses the subarray boundaries. Pass `left` and `right` explicitly.

■ Pro Tips

■ This Pattern Everywhere

The 'swap boundaries, recurse inward' pattern is the backbone of: palindrome check, quicksort partition, merge sort halving. Master it here — you'll use it for years.

■ Practice Problems

#	Problem	Diff
1	Reverse Array recursively	Easy
2	Palindrome check using recursion	Easy

3	Reverse a String (LeetCode 344)	Easy
4	Reverse Linked List recursively	Easy

■ Quick Revision

✓	Parameters: left=0, right=n-1
✓	Base case: if(left >= right) return
✓	Action: swap arr[left] and arr[right]
✓	Recurse: call with left+1, right-1
✓	Time $O(n)$, Space $O(n/2)$ for call stack

■ Visual Thinking

Think of the array as a spool of thread. You grab both ends and swap them, then slide your hands inward and swap again. Each time, you're holding a shorter section of the thread. When your hands meet in the middle, you're done. The recursion IS the sliding of your hands.