

■ Permutations

Every possible ordering of elements — $n!$ arrangements

● Recursion

■ MEDIUM

■ What You Will Learn

- How permutations differ from combinations (order matters!)
- Two approaches: swap-based and visited-array based
- Why permutations generate $n!$ results for n elements
- The classic recursion tree for permutations

■ Intuition First

Permutations of [1,2,3]: every possible order. [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1] — six total ($3! = 6$). Think of filling 3 seats: 3 choices for seat 1, 2 for seat 2, 1 for seat 3 = 6. Recursively: pick one element for the first position, then permute the rest.

■ Swap-Based Approach

Fix position 'start'. Try placing each remaining element at 'start' by swapping. Recurse on positions start+1 onwards. After recursion, swap back (backtrack). This works in-place without extra visited array.

■ Visual — Permutations of [1,2,3]

```
permute([1,2,3], start=0):
  swap(0,0)=[1,2,3] → permute([1,2,3], start=1):
    swap(1,1)=[1,2,3] → permute([1,2,3], start=2):
      start==n → ADD [1,2,3] ✓
    swap(1,2)=[1,3,2] → permute([1,3,2], start=2):
      start==n → ADD [1,3,2] ✓
    swap back → [1,2,3]
  swap(0,1)=[2,1,3] → permute([2,1,3], start=1):
    → ADD [2,1,3] ✓ and [2,3,1] ✓
  swap back → [1,2,3]
  swap(0,2)=[3,2,1] → permute([3,2,1], start=1):
    → ADD [3,2,1] ✓ and [3,1,2] ✓
  swap back → [1,2,3]
```

Total: 6 permutations = $3!$ ✓

■ Java Code — Swap Based

```
1  class Solution {
2  public List<List<Integer>> permute(int[] nums) {
```

```

3     List<List<Integer>> result = new ArrayList<>();
4     permute(nums, 0, result);
5     return result;
6 }
7
8     private void permute(int[] nums, int start, List<List<Integer>> result) {
9         // BASE CASE: all positions filled
10        if (start == nums.length) {
11            List<Integer> perm = new ArrayList<>();
12            for (int n : nums) perm.add(n);
13            result.add(perm);
14            return;
15        }
16
17        for (int i = start; i < nums.length; i++) {
18            swap(nums, start, i); // Place nums[i] at position start
19            permute(nums, start + 1, result); // Permute rest
20            swap(nums, start, i); // Backtrack
21        }
22    }
23
24    private void swap(int[] nums, int i, int j) {
25        int t = nums[i]; nums[i] = nums[j]; nums[j] = t;
26    }
27 }

```

■ ■ Time & Space Complexity

Type	Value	What It Means
Time	$O(n! \times n)$	$n!$ permutations, each takes $O(n)$ to copy
Space	$O(n)$	Recursion depth = n

■ ■ Common Mistakes

■ ■ Forgetting to Swap Back

After the recursive call, you **MUST** swap back. Without this, the array stays modified and subsequent iterations get corrupted data.

■ ■ Shallow Copy

`result.add(nums)` adds a reference to the same array. After more swaps, all entries will look the same! Always create a new list from `nums`.

■ Practice Problems

#	Problem	Diff
1	Permutations (LeetCode 46)	Medium
2	Permutations II — with duplicates (LeetCode 47)	Medium
3	Next Permutation (LeetCode 31)	Medium
4	Permutation Sequence (LeetCode 60)	Hard

■ Quick Revision

✓	Permutations: order matters, all $n!$ arrangements
✓	Swap <code>nums[start]</code> with each <code>nums[i]</code> in <code>[start..n-1]</code>
✓	Recurse with <code>start+1</code>
✓	MUST swap back after recursion (backtrack)
✓	Base case: <code>start==n</code> → add copy of array to result
✓	Time $O(n! \times n)$, Space $O(n)$

■ Visual Thinking

Think of 3 people choosing seats in a photo. Person A can stand in any of 3 spots. Once A picks, B has 2 spots. C gets the last one. Each 'choosing' is a swap — you try every option, take the photo (record the permutation), then tell everyone to go back to their original spot (backtrack). Repeat for all combinations.