

## Topic Introduction

Welcome back, interview prepper! Today's PrepLetter is all about *permutations*: generating them, handling duplicates, and even finding the next one in order. Permutations show up everywhere in coding interviews, especially when the interviewer wants to see if you can generate all possible arrangements, handle edge cases, or optimize backtracking code.

### The Core Concept: Permutation Generation

A **permutation** is an ordered arrangement of elements. For example, all permutations of [1, 2, 3] are:

- [1, 2, 3]
- [1, 3, 2]
- [2, 1, 3]
- [2, 3, 1]
- [3, 1, 2]
- [3, 2, 1]

#### How it works:

To generate permutations, we often use **backtracking**. The idea is simple: build up an arrangement one element at a time, exploring all options by swapping or picking unused elements, and then backtrack to try something else.

#### Why is this useful in interviews?

Permutation problems test your understanding of recursion, backtracking, and sometimes subtle edge cases like handling duplicates. They're also a great way to show you can code clean, systematic solutions under pressure.

#### Quick Example (not from today's problems):

Suppose you want all permutations of [A, B]:

- Start with an empty path.
- Add A → path is [A], remaining: [B]
  - Add B → [A, B] (one permutation)
- Backtrack, start with [B], remaining: [A]
  - Add A → [B, A] (another permutation)

## Today's Problems: Why These Three?

We're looking at three classic LeetCode problems:

- [Permutations](#)
- [Permutations II](#)
- [Next Permutation](#)

They're grouped together because they all deal with **permutation generation or manipulation**:

- **Permutations**: Generate all possible orderings (classic backtracking).
- **Permutations II**: Like the first, but the input can have duplicates. Must avoid duplicate results.

- **Next Permutation:** Instead of generating all, find the *next* lexicographically larger arrangement.

Let's start with classic permutation generation, move to handling duplicates, and finish with the "next permutation" algorithm.

### Problem 1: Permutations

#### Problem:

Given a list of  $n$  distinct numbers, return all possible orderings (permutations) of those numbers.

[Permutations - LeetCode](#)

#### Example:

Input: [1, 2, 3]

Output:

```
[  
    [1, 2, 3],  
    [1, 3, 2],  
    [2, 1, 3],  
    [2, 3, 1],  
    [3, 1, 2],  
    [3, 2, 1]  
]
```

#### How do we think about this?

Try pen and paper: Pick a number, then generate all permutations of the remaining numbers, and repeat. This is classic **backtracking**.

#### Try this case manually:

Input: [0, 1]

What are all the permutations?

#### Brute-force approach:

Generate all possible arrangements by swapping at every position recursively.

- Time complexity:  $O(n! * n)$  (since there are  $n!$  permutations, each  $n$  long)
- Space complexity:  $O(n! * n)$  (for all results)

#### Optimal approach: Backtracking

Here's the pattern:

- For each position, try every number that hasn't been used yet.
- Add it to the current path, mark it as used.
- Continue to the next position.
- When the path's length is  $n$ , add it to results.
- Backtrack: unmark the number, remove it from the path.

### Python Solution

```
def permute(nums):
    results = []
    path = []

    def backtrack():
        # If the path is as long as nums, we've built a permutation
        if len(path) == len(nums):
            results.append(path[:]) # Make a copy
            return

        for num in nums:
            if num in path:
                continue # Skip already used numbers
            path.append(num)
            backtrack()
            path.pop() # Remove and try next

    backtrack()
    return results

# Example:
# Input: [1, 2, 3]
# Output: [
#   [1, 2, 3],
#   [1, 3, 2],
#   [2, 1, 3],
#   [2, 3, 1],
#   [3, 1, 2],
#   [3, 2, 1]
# ]
```

**Time Complexity:**  $O(n! * n)$

**Space Complexity:**  $O(n! * n)$  (for storing all permutations and the recursion stack)

### Explaining the Code

- **results**: Stores all found permutations.
- **path**: Current building permutation.
- **backtrack()**: Tries every unused number for the next slot.
  - If **path** is complete, copy and add to **results**.
  - Otherwise, for each number in **nums**, if not used, add to path, recurse, then remove (backtrack).

**Trace for `[1, 2]`:**

- Start: path = []
- Add 1: path = [1]
  - Add 2: path = [1, 2] (full, add to results)
  - Backtrack: remove 2
- Backtrack: remove 1, path = []
- Add 2: path = [2]
  - Add 1: path = [2, 1] (full, add to results)

User dry-run:

Try [3, 4, 5] as input and see what permutations you generate.

**Take a moment to solve this on your own before jumping into the solution.**

## Problem 2: Permutations II

[Permutations II - LeetCode](#)

### Problem:

Given a list of numbers that may contain duplicates, return all unique orderings (permutations) of those numbers.

### How is this different?

Some numbers may repeat. We must **avoid returning duplicate permutations**.

### Example:

Input: [1, 1, 2]

Output:

```
[  
    [1, 1, 2],  
    [1, 2, 1],  
    [2, 1, 1]  
]
```

Notice [1, 1, 2] appears only once, even though swapping the two 1's produces the same permutation.

### Try this case manually:

Input: [2, 2, 1]

What unique permutations can you make?

### Brute-force approach:

Generate all permutations, then use a set to filter duplicates.

- Time:  $O(n! * n)$
- Space:  $O(n! * n)$  for storing all, then deduplicating.

### Optimal approach: Sort and skip duplicates in backtracking

Pattern:

- Sort the array so duplicates are adjacent.
- Use a boolean **used** array to track which elements are in the current path.

- Before choosing a number, skip it if it's the **same as the previous number and the previous was not used** (avoids duplicates).

### Pseudocode

```
function permuteUnique(nums):  
    sort(nums)  
    results = []  
    used = [False] * len(nums)  
  
    function backtrack(path):  
        if length(path) == length(nums):  
            results.append(copy(path))  
            return  
  
        for i from 0 to len(nums)-1:  
            if used[i]:  
                continue  
            if i > 0 and nums[i] == nums[i-1] and not used[i-1]:  
                continue // Skip duplicates  
            used[i] = True  
            path.append(nums[i])  
            backtrack(path)  
            path.pop()  
            used[i] = False  
  
    backtrack([])  
    return results
```

### Step-by-step for `[1, 1, 2]`:

- Start with empty path, all numbers unused.
- Try index 0 (1): used[0]=True, path=[1]
  - Try index 1 (1): used[1]=True, path=[1,1]
    - Try index 2 (2): used[2]=True, path=[1,1,2] (add)
    - Backtrack: used[2]=False, path=[1,1]
    - Backtrack: used[1]=False, path=[1]
    - Try index 2 (2): used[2]=True, path=[1,2]
      - Try index 1 (1): used[1]=True, path=[1,2,1] (add)
  - ...and so on.

### Another test case to dry-run:

Input: [1, 2, 2]

What are all unique permutations?

### Time Complexity:

Still  $O(n! * n)$  in the worst case, but avoids producing duplicate results.

### Space Complexity:

$O(n! * n)$  for results and recursion stack.

## Problem 3: Next Permutation

[Next Permutation - LeetCode](#)

### Problem:

Given an array of numbers, rearrange it into the lexicographically next greater permutation. If no such permutation exists (i.e., it's the last permutation), rearrange it as the lowest possible order (sorted ascending).

*This must be done in-place.*

### How is this different?

Instead of generating all permutations, we want just the *next one* in "dictionary" order.

### Example:

Input: [1, 2, 3]

Output: [1, 3, 2]

Input: [3, 2, 1]

Output: [1, 2, 3] (since it's already the last permutation, we wrap around)

### Brute-force approach:

Generate all permutations, sort them, and pick the next one.

- Time:  $O(n! * n)$
- Space:  $O(n! * n)$  — not feasible.

### Optimal approach: In-place "next permutation" algorithm

Pattern:

- **Find the first decreasing element from the end:** Scan from right to left, find the first index *i* where `nums[i] < nums[i+1]`. If none found, reverse entire array.
- **Find the element just larger than `nums[i]`:** From the end, find the first number larger than `nums[i]`, say at index *j*.
- **Swap `nums[i]` and `nums[j]`.**
- **Reverse the subarray from `i+1` to end.**

### Pseudocode

```
function nextPermutation(nums):  
    n = length(nums)  
    i = n - 2  
    while i >= 0 and nums[i] >= nums[i+1]:  
        i -= 1  
    if i >= 0:  
        j = n - 1
```

```
while nums[j] <= nums[i]:  
    j -= 1  
    swap(nums[i], nums[j])  
reverse(nums, i+1, n-1)
```

### Step-by-step for `[1, 3, 2]`:

- Find **i**: `nums[1]=3 > nums[2]=2`, so move left. `nums[0]=1 < nums[1]=3`, so `i=0`
- Find **j**: From end, `nums[2]=2 > nums[0]=1`, so `j=2`
- Swap: **[2, 3, 1]**
- Reverse from `i+1=1` to end: **[2, 1, 3]**

### Try this test case:

Input: **[2, 3, 1]**

What should the output be?

**Time Complexity:**  $O(n)$

**Space Complexity:**  $O(1)$ , since it's all in-place.

### Reflect:

Notice how this is a neat, efficient way to jump to the next permutation, without generating them all. Try implementing this yourself and dry-running on **[1, 5, 1]**.

## Summary and Next Steps

Today, you tackled three central permutation problems:

- **Permutation generation, with and without duplicates:** Using backtracking, with care to avoid repeats when needed.
- **Next permutation:** Finding the next lexicographical arrangement, efficiently and in-place.

### Key patterns and insights:

- Backtracking is the backbone for generating permutations.
- Handling duplicates? Sort and skip repeated elements smartly.
- In-place next permutation uses a clever scan-swap-reverse trick.

### Common interview mistakes:

- Forgetting to skip duplicates in Permutations II.
- Not copying the path when storing permutations (results may all end up the same!).
- Off-by-one errors in the next permutation scan and reverse.

## Action List

- Solve all three problems on your own, even if you've read the explanations!
- For Problem 2 and 3, try to devise alternative approaches or optimize your code.
- Explore other backtracking problems, like combinations or subsets.
- Compare your solutions to others, especially for handling duplicates and edge cases.

- If you get stuck, that's normal — review, practice, and you'll improve!

Keep practicing, and soon you'll be permuting (and acing interviews) with confidence!