Recursion

1. Subsets

Pattern: Backtracking (Subset Generation)

Problem Statement

Given an integer array `nums` of **unique** elements, return all possible subsets (the power set).
The solution set must **not** contain duplicate subsets. Return the solution in **any order**.

Sample Input & Output

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
Explanation: All combinations of elements, including empty set.
```

```
Input: nums = [0]
Output: [[],[0]]
Explanation: Only two subsets possible with one element.
```

```
Input: nums = []
Output: [[]]
Explanation: Empty input → only empty subset.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        # STEP 1: Initialize result list and current path
        #   - result collects all valid subsets
        #   - path tracks current subset being built
        result = []
        path = []

        # STEP 2: Define recursive backtrack function
        #   - start: index to begin choosing from
        #   - invariant: all subsets using nums[0:start] are fixed
        def backtrack(start: int):
            # Add current path as a new subset (always valid)
            result.append(path.copy())

            # Try adding each remaining element
            for i in range(start, len(nums)):
                # Choose nums[i]
                path.append(nums[i])
                # Recurse with next index to avoid reuse
                backtrack(i + 1)
                # Unchoose: remove last added to try next option
                path.pop()

        # STEP 3: Start backtracking from index 0
        backtrack(0)

        # STEP 4: Return all collected subsets
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    output1 = sol.subsets([1, 2, 3])
    expected1 = [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    assert sorted(output1) == sorted(expected1), f"Test 1 failed: {output1}"
```

```python
#  Test 2: Edge case - single element
output2 = sol.subsets([0])
expected2 = [[], [0]]
assert output2 == expected2, f"Test 2 failed: {output2}"

#  Test 3: Tricky/negative - empty input
output3 = sol.subsets([])
expected3 = [[]]
assert output3 == expected3, f"Test 3 failed: {output3}"

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `subsets([1, 2])` step by step:

1. **Initialize**:
   `result = []`, `path = []`
   Call `backtrack(0)`.
2. **Inside `backtrack(0)`**:
   - Append `path.copy()` → `result = [[]]`
   - Loop `i` from 0 to 1:
     - **i=0**:
       * `path.append(1)` → `path = [1]`
       * Call `backtrack(1)`
3. **Inside `backtrack(1)`**:
   - Append `[1]` → `result = [[], [1]]`
   - Loop `i` from 1 to 1:
     - **i=1**:

        * `path.append(2)` → `path = [1,2]`
        * Call `backtrack(2)`
4. **Inside `backtrack(2)`**:
    - Append `[1,2]` → `result = [[], [1], [1,2]]`
    - Loop doesn't run (`start=2`, `len=2`)
    - Return
5. **Back to `backtrack(1)`**:
    - `path.pop()` → `path = [1]`
    - Loop ends
6. **Back to `backtrack(0)`**:
    - `path.pop()` → `path = []`
    - **i=1**:
        - `path.append(2)` → `path = [2]`
        - Call `backtrack(2)`
7. **Inside `backtrack(2)` again**:
    - Append `[2]` → `result = [[], [1], [1,2], [2]]`
    - Return
8. **Final result**: `[[], [1], [1,2], [2]]` → matches expected.

Key insight: Every time we enter `backtrack`, we **record the current path** — even if empty — because every partial path is a valid subset.

Complexity Analysis

- **Time Complexity**: `O(2^n * n)`

  There are `2^n` subsets. For each, we spend `O(n)` time copying the path to result (in worst case, path length = n).

- **Space Complexity**: `O(n)`

  The recursion depth is at most `n` (one call per element). The `path` list also uses `O(n)` space. Output space (`O(2^n * n)`) is not counted in auxiliary space.

2. Permutations

Pattern: Backtracking

Problem Statement

Given an array `nums` of distinct integers, return all the possible permutations. You can return the answer in any order.

Sample Input & Output

```
Input: [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
Explanation: All possible orderings of the 3 distinct elements.
```

```
Input: [0,1]
Output: [[0,1],[1,0]]
Explanation: Only two permutations for two elements.
```

```
Input: [5]
Output: [[5]]
Explanation: Single-element array has only one permutation.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - result: collects all valid permutations
        #   - path: current partial permutation being built
        #   - used: boolean list to track which indices are in path
        result = []
        path = []
        used = [False] * len(nums)

        def backtrack():
            # STEP 2: Main loop / recursion
            #   - Base case: if path length == nums length,
            #     we have a complete permutation
            if len(path) == len(nums):
                result.append(path[:])  # append a copy
                return

            # STEP 3: Update state / bookkeeping
            #   - Try every number not yet used
            for i in range(len(nums)):
                if not used[i]:
                    # Choose
                    path.append(nums[i])
                    used[i] = True
                    # Explore
                    backtrack()
                    # Unchoose (backtrack)
                    path.pop()
                    used[i] = False

        backtrack()
        # STEP 4: Return result
        #   - All permutations collected in result
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
```

```python
    sol = Solution()

    #   Test 1: Normal case
    assert sol.permute([1,2,3]) == [
        [1,2,3],[1,3,2],
        [2,1,3],[2,3,1],
        [3,1,2],[3,2,1]
    ], "Normal case failed"

    #   Test 2: Edge case
    assert sol.permute([5]) == [[5]], "Single element failed"

    #   Test 3: Tricky/negative
    assert sol.permute([0,1]) == [[0,1],[1,0]], "Two elements failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `permute([1,2])` step by step:

1. **Initialization**:
   - `result = []`
   - `path = []`
   - `used = [False, False]`
2. **First call to `backtrack()`**:
   - `len(path) = 0  2` → skip base case
   - Loop `i = 0`: `used[0]` is `False`
     - **Choose**: `path = [1]`, `used = [True, False]`
     - **Recurse**: call `backtrack()`

3. **Second call (depth 1)**:
   - `len(path) = 1  2`
   - Loop `i = 0`: `used[0] = True` → skip
   - Loop `i = 1`: `used[1] = False`
     - **Choose**: `path = [1,2]`, `used = [True, True]`
     - **Recurse**: call `backtrack()`
4. **Third call (depth 2)**:
   - `len(path) = 2 == 2` → **base case hit!**
   - Append copy: `result = [[1,2]]`
   - Return to previous call
5. **Backtrack from depth 2**:
   - **Unchoose**: `path = [1]`, `used = [True, False]`
   - Loop ends → return to depth 0
6. **Back at depth 0, continue loop**:
   - `i = 1`: `used[1] = False`
     - **Choose**: `path = [2]`, `used = [False, True]`
     - **Recurse**: call `backtrack()`
7. **New depth 1**:
   - `i = 0`: not used → add `1` → `path = [2,1]`
   - Base case → append `[2,1]` to `result`
   - Backtrack → `path = [2]` → then `path = []`
8. **Final result**: `[[1,2], [2,1]]`

Key idea: **build partial solution**, **recurse**, then **undo** (backtrack) to try alternatives.

Complexity Analysis

- **Time Complexity**: `O(N × N!)`

  There are `N!` permutations. Each permutation takes `O(N)` time to copy into the result list. The backtracking explores `N!` leaves and the work per node is proportional to depth, but the dominant cost is copying full permutations.

- **Space Complexity**: `O(N)`

  The recursion depth is `N` (one call per element in path). The `path` and `used` arrays each use `O(N)` space. The output list is not counted toward auxiliary space complexity per standard LeetCode conventions.

3. Letter Combinations of a Phone Number

Pattern: Backtracking (Recursive Enumeration)

Problem Statement

Given a string containing digits from `2` to `9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**. A mapping of digits to letters (just like on the telephone buttons) is given below. Note that `1` does not map to any letters.

```
2 -> "abc"
3 -> "def"
4 -> "ghi"
5 -> "jkl"
6 -> "mno"
7 -> "pqrs"
8 -> "tuv"
9 -> "wxyz"
```

Sample Input & Output

```
Input: "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
Explanation: "2" maps to "abc", "3" to "def". All pairwise combos are formed.
```

```
Input: ""
Output: []
Explanation: Empty input → no combinations.
```

```
Input: "2"
Output: ["a","b","c"]
Explanation: Single digit → just its letters.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        # STEP 1: Initialize structures
        #   - Map each digit to its letters (constant lookup)
        #   - Use backtracking to build combinations
        if not digits:
            return []

        digit_to_letters = {
            '2': 'abc', '3': 'def', '4': 'ghi',
            '5': 'jkl', '6': 'mno', '7': 'pqrs',
            '8': 'tuv', '9': 'wxyz'
        }
        result = []

        # STEP 2: Main loop / recursion
        #   - Recursively build combination one digit at a time
        #   - Base case: path length == digits length → add to result
        def backtrack(index: int, path: str):
            # Base case: processed all digits
            if index == len(digits):
```

```
                result.append(path)
                return

            # Get letters for current digit
            letters = digit_to_letters[digits[index]]

            # STEP 3: Update state / bookkeeping
            #   - Try each letter, recurse to next digit
            #   - No explicit "undo" needed because path is immutable
            for letter in letters:
                backtrack(index + 1, path + letter)

        backtrack(0, "")
        # STEP 4: Return result
        #   - Handles empty input early; otherwise returns built list
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.letterCombinations("23") == [
        "ad","ae","af","bd","be","bf","cd","ce","cf"
    ], "Test 1 failed"

    #  Test 2: Edge case - empty input
    assert sol.letterCombinations("") == [], "Test 2 failed"

    #  Test 3: Tricky/negative - single digit
    assert set(sol.letterCombinations("2")) == {"a", "b", "c"}, \
        "Test 3 failed"

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `letterCombinations("23")` step by step.

1. **Initial Call**:
   - `digits = "23"` → not empty, so proceed.
   - `digit_to_letters` built.
   - `result = []`
   - Call `backtrack(0, "")`
2. **First `backtrack(0, "")`**:
   - `index = 0`, `path = ""`
   - `digits[0] = '2'` → `letters = "abc"`
   - Loop over `'a'`, `'b'`, `'c'`:

   → **For `'a'`**: call `backtrack(1, "a")`
3. **Inside `backtrack(1, "a")`**:
   - `index = 1`, `path = "a"`
   - `digits[1] = '3'` → `letters = "def"`
   - Loop over `'d'`, `'e'`, `'f'`:

     → `'d'`: call `backtrack(2, "ad")`

     – `index == len(digits)` $(2 == 2)$ → append `"ad"` to `result`
     → `'e'`: append `"ae"`
     → `'f'`: append `"af"`

   → `result = ["ad", "ae", "af"]`
4. **Back to first loop, next letter `'b'`**:
   - Call `backtrack(1, "b")` → same as above → adds `"bd","be","bf"`
5. **Then `'c'`**:

- Adds `"cd","ce","cf"`

6. **Final `result`**:
   `["ad","ae","af","bd","be","bf","cd","ce","cf"]`

No mutation of shared state → clean recursion with immutable strings.

Complexity Analysis

- **Time Complexity**: `O(4^N * N)`

  Each digit maps to up to 4 letters (e.g., '7' or '9'). For `N` digits, we generate up to `4^N` combinations. Each combination takes `O(N)` time to build (string concatenation). So total = `O(N * 4^N)`.

- **Space Complexity**: `O(N)`

  The recursion depth is `N` (one call per digit). The output list is **not** counted in space complexity (as per standard LeetCode conventions), but the call stack uses `O(N)` space. Intermediate strings are created anew each time (immutable), but max depth is `N`.

4. Generate Parentheses

Pattern: Backtracking

Problem Statement

Given `n` pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Sample Input & Output

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
Explanation: All 5 valid combinations of 3 matched pairs.
```

```
Input: n = 1
Output: ["()"]
Explanation: Only one valid pair.
```

```
Input: n = 0
Output: [""]
Explanation: Zero pairs → empty string is valid.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        # STEP 1: Initialize result list and start backtracking
        #   - We'll build strings recursively, tracking open/close counts

        result = []

        def backtrack(current: str, open_count: int, close_count: int):
            # STEP 2: Base case - valid combination found
            #   - When string length is 2*n, add to result
            if len(current) == 2 * n:
                result.append(current)
                return

            # STEP 3: Add '(' if we haven't used all n opens
            #   - Ensures we never exceed n open parentheses
            if open_count < n:
                backtrack(current + "(", open_count + 1, close_count)

            # STEP 4: Add ')' only if it won't break validity
            #   - We can close only if more opens than closes so far
```

```
            if close_count < open_count:
                backtrack(current + ")", open_count, close_count + 1)

        # Start with empty string and zero counts
        backtrack("", 0, 0)
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    output1 = sol.generateParenthesis(3)
    expected1 = ["((()))","(()())","(())()","()(())","()()()"]
    assert sorted(output1) == sorted(expected1), f"Test 1 failed: {output1}"
    print("  Test 1 passed:", output1)

    #  Test 2: Edge case (n = 1)
    output2 = sol.generateParenthesis(1)
    expected2 = ["()"]
    assert output2 == expected2, f"Test 2 failed: {output2}"
    print("  Test 2 passed:", output2)

    #  Test 3: Tricky/negative (n = 0)
    output3 = sol.generateParenthesis(0)
    expected3 = [""]
    assert output3 == expected3, f"Test 3 failed: {output3}"
    print("  Test 3 passed:", output3)
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `generateParenthesis(2)` step by step to see how backtracking builds valid strings.

Initial Call: `backtrack("", 0, 0)`
- `current = ""`, `open = 0`, `close = 0`

- Length 4 → continue
- `open < 2` → call `backtrack("(", 1, 0)`

Call 1: `backtrack("(", 1, 0)`
- Length = 1 4
- `open < 2` → call `backtrack("((", 2, 0)`
- Also, `close (0) < open (1)` → will later call `backtrack("()", 1, 1)`

Call 2: `backtrack("((", 2, 0)`
- Length = 2 4
- `open == 2` → can't add more '('
- `close (0) < open (2)` → call `backtrack("(()", 2, 1)`

Call 3: `backtrack("(()", 2, 1)`
- Length = 3 4
- Can't add '(' (max used)
- `close (1) < open (2)` → call `backtrack("(())", 2, 2)`

Call 4: `backtrack("(())", 2, 2)`
- Length = 4 → add to result: `["(())"]` → return

Now backtrack to **Call 1** and explore second branch:
Call 5: `backtrack("()", 1, 1)`
- Length = 2
- `open < 2` → call `backtrack("()(", 2, 1)`
- `close == open` → can't add ')' yet

Call 6: `backtrack("()(", 2, 1)`
- Length = 3
- Can't add '('
- `close (1) < open (2)` → call `backtrack("()()", 2, 2)`

Call 7: `backtrack("()()", 2, 2)`
- Length = 4 → add to result: `["(())", "()()"]`

Final result: `["(())", "()()"]` → matches expected.

Key idea: **Only add ')' when it won't create more closes than opens** — this maintains validity at every step.

Complexity Analysis

- **Time Complexity**: `O(4^n / sqrt(n))`

  This is the *n-th Catalan number*, which counts valid parenthesis combinations. Each valid string has length `2n`, and we explore only valid paths — but the growth is exponential.

- **Space Complexity**: `O(4^n / sqrt(n))`

  Dominated by the output list storing all valid combinations. The recursion depth is `O(n)` (max call stack), but output size dwarfs it.

5. N-Queens

Pattern: Backtracking

Problem Statement

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the* ***n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Sample Input & Output

```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],
         ["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above.
```

```
Input: n = 1
Output: [["Q"]]
Explanation: Only one queen on a 1x1 board - trivially valid.
```

```
Input: n = 2
Output: []
Explanation: No valid placement exists for 2 queens on a 2x2 board.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        # STEP 1: Initialize structures
        #   - board: current placement ('.' initially)
        #   - cols, diag1, diag2: track attacked columns & diagonals
        board = [['.' for _ in range(n)] for _ in range(n)]
        cols = set()          # occupied columns
        diag1 = set()         # row - col (constant for \ diagonals)
        diag2 = set()         # row + col (constant for / diagonals)
        results = []

        def backtrack(row: int):
            # STEP 2: Main loop / recursion
            #   - Base case: placed queens in all rows
            if row == n:
                results.append([''.join(r) for r in board])
                return

            # Try placing queen in each column of current row
            for col in range(n):
                # Skip if under attack
                if (col in cols or
                    (row - col) in diag1 or
                    (row + col) in diag2):
                    continue
```

```
                # STEP 3: Update state / bookkeeping
                board[row][col] = 'Q'
                cols.add(col)
                diag1.add(row - col)
                diag2.add(row + col)

                backtrack(row + 1)

                # Undo changes (backtrack)
                board[row][col] = '.'
                cols.remove(col)
                diag1.remove(row - col)
                diag2.remove(row + col)

        backtrack(0)
        return results

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    res1 = sol.solveNQueens(4)
    expected1 = [
        [".Q..", "...Q", "Q...", "..Q."],
        ["..Q.", "Q...", "...Q", ".Q.."]
    ]
    assert res1 == expected1, f"Test 1 failed: got {res1}"
    print(" Test 1 passed: n=4")

    #  Test 2: Edge case
    res2 = sol.solveNQueens(1)
    expected2 = [["Q"]]
    assert res2 == expected2, f"Test 2 failed: got {res2}"
    print(" Test 2 passed: n=1")

    #  Test 3: Tricky/negative
    res3 = sol.solveNQueens(2)
    expected3 = []
    assert res3 == expected3, f"Test 3 failed: got {res3}"
    print(" Test 3 passed: n=2")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `n = 2` to see **why no solution exists**, step by step.

1. **Initialize**:
   - `board = [['.', '.'], ['.', '.']]`
   - `cols`, `diag1`, `diag2` are empty sets.
   - `results = []`
2. **Call `backtrack(0)`** (start at row 0):
   - Loop over `col = 0` and `col = 1`.
3. **Try `col = 0` in row 0**:
   - Not in any attack set → safe.
   - Update:
     - `board[0][0] = 'Q'` → `[['Q', '.'], ['.', '.']]`
     - `cols = {0}`, `diag1 = {0}`, `diag2 = {0}`
   - Call `backtrack(1)`.
4. **In `backtrack(1)` (row 1)**:
   - Try `col = 0`:
     - `0 in cols` → **attack!** skip.
   - Try `col = 1`:
     - Check diagonals:
       - `row - col = 1 - 1 = 0` → in `diag1` → **attack!** skip.
   - No valid column → return without adding to results.
5. **Backtrack**: undo changes from step 3:

- `board[0][0] = '.'`
- Remove 0 from all sets → back to initial state.

6. **Try `col = 1` in row 0**:
   - Safe → update:
     - `board[0][1] = 'Q'` → `[['.', 'Q'], ['.', '.']]`
     - `cols = {1}`, `diag1 = {-1}`, `diag2 = {1}`
   - Call `backtrack(1)`.
7. **In `backtrack(1)` again**:
   - `col = 0`:
     - `row + col = 1 + 0 = 1` → in `diag2` → **attack!**
   - `col = 1`: in `cols` → **attack!**
   - No valid placement → return.
8. **All options exhausted** → `results` remains empty → return `[]`.

Final output: `[]` — correctly reflects impossibility for `n=2`.

Complexity Analysis

- **Time Complexity**: `O(N!)`

  At row 0, we have `N` choices; row 1 has `N-1`; row 2 `N-2`, etc.
  Though pruning reduces actual work, worst-case explores `N!` placements.
  Each placement check is `O(1)` due to hash sets.

- **Space Complexity**: $O(N^2)$
  - `board` uses N^2 space.
  - Recursion depth is `N` (one per row).
  - Sets `cols`, `diag1`, `diag2` each store `N` integers.
    Dominated by board storage → $O(N^2)$.

6. Next Permutation

Pattern: Arrays & In-Place Manipulation

Problem Statement

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such an arrangement is not possible, the numbers must be rearranged as the lowest possible order (i.e., sorted in ascending order).
The replacement must be **in-place** and use only constant extra memory.

Sample Input & Output

```
Input: [1,2,3]
Output: [1,3,2]
Explanation: The next lexicographical permutation after [1,2,3] is [1,3,2].
```

```
Input: [3,2,1]
Output: [1,2,3]
Explanation: This is the highest permutation; reset to lowest (sorted).
```

```
Input: [1,1,5]
Output: [1,5,1]
Explanation: Duplicate values handled correctly; next valid permutation.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        # STEP 1: Initialize structures
        #   - Find the first index 'i' from right where nums[i] < nums[i+1]
        #   - This identifies the "pivot" where we can increase the prefix
        n = len(nums)
        i = n - 2
        while i >= 0 and nums[i] >= nums[i + 1]:
            i -= 1

        # STEP 2: Main loop / recursion
        #   - If such 'i' exists, find smallest 'j' > i with nums[j] > nums[i]
        #   - Swap to get next higher prefix
        if i >= 0:
            j = n - 1
            while nums[j] <= nums[i]:
                j -= 1
            nums[i], nums[j] = nums[j], nums[i]

        # STEP 3: Update state / bookkeeping
        #   - Reverse suffix after 'i' to get smallest lexicographic order
        #   - If no pivot (i = -1), this reverses entire array → sorted order
        left, right = i + 1, n - 1
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

        # STEP 4: Return result
        #   - In-place modification; nothing to return

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    nums1 = [1, 2, 3]
    sol.nextPermutation(nums1)
    print(f"Test 1: {nums1}")  # Expected: [1, 3, 2]
```

```
    #  Test 2: Edge case (descending → reset)
    nums2 = [3, 2, 1]
    sol.nextPermutation(nums2)
    print(f"Test 2: {nums2}")  # Expected: [1, 2, 3]

    #  Test 3: Tricky/negative (duplicates)
    nums3 = [1, 1, 5]
    sol.nextPermutation(nums3)
    print(f"Test 3: {nums3}")  # Expected: [1, 5, 1]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `nums = [1, 2, 3]`

1. **Initialize `n = 3`, `i = 1`** (`n - 2`).
   - Check `nums[1] = 2` vs `nums[2] = 3` → `2 < 3` → stop.
   - So `i = 1` (pivot found).
2. **Find `j` from right where `nums[j] > nums[i]`**:
   - Start `j = 2`, `nums[2] = 3 > 2` → found.
   - Swap `nums[1]` and `nums[2]` → `nums = [1, 3, 2]`.
3. **Reverse suffix after `i = 1`**:
   - Suffix is `[2]` (only one element) → reversal does nothing.
   - Final: `[1, 3, 2]`.

Now **Test 2**: `nums = [3, 2, 1]`

1. Start `i = 1`: `nums[1] = 2 >= 1` → continue.
2. `i = 0`: `nums[0] = 3 >= 2` → continue.

3. `i = -1`: loop ends → no pivot.
4. Since `i = -1`, reverse entire array:
    - Swap `nums[0]` and `nums[2]` → `[1, 2, 3]`.

Test 3: `nums = [1, 1, 5]`

1. `i = 1`: `nums[1] = 1 < 5` → pivot at `i = 1`.
2. Find `j`: `j = 2`, `5 > 1` → swap → `[1, 5, 1]`.
3. Reverse suffix after `i = 1` → only `[1]` → unchanged.
4. Result: `[1, 5, 1]`.

Complexity Analysis

- **Time Complexity**: `O(n)`

    At most three linear passes: (1) find pivot, (2) find successor, (3) reverse suffix. Each is O(n), so total O(n).

- **Space Complexity**: `O(1)`

    Only a few integer variables used; all operations are in-place on input list.