**[22. Generate Parentheses](https://leetcode.com/problems/generate-parentheses/)**

class Solution:

def generateParenthesis(self, n):

res = []

def backtrack(s='', open\_count=0, close\_count=0):

if len(s) == 2 \* n:

res.append(s)

return

if open\_count < n:

backtrack(s + '(', open\_count + 1, close\_count)

if close\_count < open\_count:

backtrack(s + ')', open\_count, close\_count + 1)

backtrack()

return res

[**24. Swap Nodes in Pairs**](https://leetcode.com/problems/swap-nodes-in-pairs/)

class Solution:

def swapPairs(self, head):

dummy = ListNode(0)

dummy.next = head

prev = dummy

while prev.next and prev.next.next:

first = prev.next

second = first.next

# Swap nodes

first.next = second.next

second.next = first

prev.next = second

# Move prev to next pair

prev = first

return dummy.next

[**26. Remove Duplicates from Sorted Array**](https://leetcode.com/problems/remove-duplicates-from-sorted-array/)

class Solution:

def removeDuplicates(self, nums):

if not nums:

return 0

write = 1

for read in range(1, len(nums)):

if nums[read] != nums[read - 1]:

nums[write] = nums[read]

write += 1

return write

[**27. Remove Element**](https://leetcode.com/problems/remove-element/)

# Option 1: Without type hints (simplest for LeetCode)

class Solution:

def removeElement(self, nums, val):

write = 0

for read in range(len(nums)):

if nums[read] != val:

nums[write] = nums[read]

write += 1

return write

[**29. Divide Two Integers**](https://leetcode.com/problems/divide-two-integers/)

class Solution:

def divide(self, dividend, divisor):

# Handle overflow case

if dividend == -2\*\*31 and divisor == -1:

return 2\*\*31 - 1

# Determine the sign of the result

negative = (dividend < 0) != (divisor < 0)

# Work with absolute values

dividend, divisor = abs(dividend), abs(divisor)

quotient = 0

# Bit manipulation to find quotient

for i in range(31, -1, -1):

if (dividend >> i) >= divisor:

dividend -= divisor << i

quotient += 1 << i

if negative:

quotient = -quotient

# Clamp result to 32-bit signed integer range

return max(min(quotient, 2\*\*31 - 1), -2\*\*31)

[**31. Next Permutation**](https://leetcode.com/problems/next-permutation/)

class Solution:

def nextPermutation(self, nums):

n = len(nums)

i = n - 2

# Find the first decreasing element from the end

while i >= 0 and nums[i] >= nums[i + 1]:

i -= 1

if i >= 0:

# Find the element just larger than nums[i]

j = n - 1

while nums[j] <= nums[i]:

j -= 1

# Swap them

nums[i], nums[j] = nums[j], nums[i]

# Reverse the subarray after index i

left, right = i + 1, n - 1

while left < right:

nums[left], nums[right] = nums[right], nums[left]

left += 1

right -= 1

[**35. Search Insert Position**](https://leetcode.com/problems/search-insert-position/)

class Solution:

def searchInsert(self, nums, target):

left, right = 0, len(nums)

while left < right:

mid = (left + right) // 2

if nums[mid] < target:

left = mid + 1

else:

right = mid

return left

[**39. Combination Sum**](https://leetcode.com/problems/combination-sum/)

class Solution:

def combinationSum(self, candidates, target):

res = []

def backtrack(start, target, path):

if target == 0:

res.append(path)

return

for i in range(start, len(candidates)):

if candidates[i] <= target:

backtrack(i, target - candidates[i], path + [candidates[i]])

backtrack(0, target, [])

return res

[**41. First Missing Positive**](https://leetcode.com/problems/first-missing-positive/)

class Solution:

def firstMissingPositive(self, nums):

n = len(nums)

# Step 1: Replace negative numbers and zeros with n+1

for i in range(n):

if nums[i] <= 0 or nums[i] > n:

nums[i] = n + 1

# Step 2: Use the index as a hash key and mark the corresponding index as negative

for i in range(n):

val = abs(nums[i])

if 1 <= val <= n:

nums[val - 1] = -abs(nums[val - 1])

# Step 3: Find the first index with a positive value

for i in range(n):

if nums[i] > 0:

return i + 1

return n + 1

[**45. Jump Game II**](https://leetcode.com/problems/jump-game-ii/)

class Solution:

def jump(self, nums):

jumps, current\_end, farthest = 0, 0, 0

for i in range(len(nums) - 1):

farthest = max(farthest, i + nums[i])

if i == current\_end:

jumps += 1

current\_end = farthest

return jumps