**Add two numbers:**

# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution:

def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:

dummy = ListNode() # dummy head

current = dummy

carry = 0

while l1 or l2 or carry:

val1 = l1.val if l1 else 0

val2 = l2.val if l2 else 0

# sum of two digits + carry

total = val1 + val2 + carry

carry = total // 10

digit = total % 10

# create new node

current.next = ListNode(digit)

current = current.next

# move forward

if l1: l1 = l1.next

if l2: l2 = l2.next

return dummy.next

**longest substring:**

class Solution:

def lengthOfLongestSubstring(self, s: str) -> int:

char\_set = set()

left = 0

max\_len = 0

for right in range(len(s)):

# If duplicate, move left pointer until it's unique

while s[right] in char\_set:

char\_set.remove(s[left])

left += 1

# Add current char

char\_set.add(s[right])

# Update max length

max\_len = max(max\_len, right - left + 1)

return max\_len

**sorted arrays:**

class Solution:

def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:

# Ensure nums1 is the smaller array

if len(nums1) > len(nums2):

nums1, nums2 = nums2, nums1

m, n = len(nums1), len(nums2)

total = m + n

half = total // 2

left, right = 0, m

while left <= right:

i = (left + right) // 2 # partition in nums1

j = half - i # partition in nums2

# Edge values (handle out-of-bounds by using infinities)

left1 = nums1[i-1] if i > 0 else float("-inf")

right1 = nums1[i] if i < m else float("inf")

left2 = nums2[j-1] if j > 0 else float("-inf")

right2 = nums2[j] if j < n else float("inf")

# Correct partition found

if left1 <= right2 and left2 <= right1:

# Odd total length → median is min(right side)

if total % 2:

return float(min(right1, right2))

# Even total length → average of max(left) and min(right)

return (max(left1, left2) + min(right1, right2)) / 2

# Move binary search

elif left1 > right2:

right = i - 1

else:

left = i + 1

**palindromic substring:**

class Solution:

def longestPalindrome(self, s: str) -> str:

def expand\_around\_center(left: int, right: int) -> str:

while left >= 0 and right < len(s) and s[left] == s[right]:

left -= 1

right += 1

return s[left+1:right] # substring between mismatched boundaries

longest = ""

for i in range(len(s)):

# Odd length palindrome

odd = expand\_around\_center(i, i)

# Even length palindrome

even = expand\_around\_center(i, i+1)

# Update longest

if len(odd) > len(longest):

longest = odd

if len(even) > len(longest):

longest = even

return longest

**zigzag conversion:**

class Solution:

def convert(self, s: str, numRows: int) -> str:

# Edge cases

if numRows == 1 or numRows >= len(s):

return s

rows = [''] \* numRows

cur\_row = 0

going\_down = False

for ch in s:

rows[cur\_row] += ch

# flip direction at the top or bottom row

if cur\_row == 0 or cur\_row == numRows - 1:

going\_down = not going\_down

cur\_row += 1 if going\_down else -1

return ''.join(rows)

**reverse integer:**

class Solution:

def reverse(self, x: int) -> int:

INT\_MAX = 2\*\*31 - 1 # 2147483647

INT\_MIN = -2\*\*31 # -2147483648

sign = -1 if x < 0 else 1

x = abs(x)

rev = 0

while x != 0:

pop = x % 10

x //= 10

# Check for overflow before multiplying

if rev > (INT\_MAX - pop) // 10:

return 0

rev = rev \* 10 + pop

return sign \* rev

**string to integer:**

class Solution:

def myAtoi(self, s: str) -> int:

INT\_MAX = 2\*\*31 - 1

INT\_MIN = -2\*\*31

i = 0

n = len(s)

# Step 1: Skip leading whitespace

while i < n and s[i] == ' ':

i += 1

# Step 2: Handle sign

sign = 1

if i < n and (s[i] == '+' or s[i] == '-'):

sign = -1 if s[i] == '-' else 1

i += 1

# Step 3: Convert digits to integer

result = 0

while i < n and s[i].isdigit():

digit = int(s[i])

# Check for overflow

if result > (INT\_MAX - digit) // 10:

return INT\_MAX if sign == 1 else INT\_MIN

result = result \* 10 + digit

i += 1

return sign \* result

**palindrome number:**

class Solution:

def isPalindrome(self, x: int) -> bool:

# Negative numbers and numbers ending with 0 (but not 0 itself) are not palindromes

if x < 0 or (x % 10 == 0 and x != 0):

return False

reversed\_half = 0

while x > reversed\_half:

reversed\_half = reversed\_half \* 10 + x % 10

x //= 10

# For even length, x == reversed\_half

# For odd length, reversed\_half // 10 removes the middle digit

return x == reversed\_half or x == reversed\_half // 10

**regular expression matching:**

class Solution:

def isMatch(self, s: str, p: str) -> bool:

m, n = len(s), len(p)

# dp[i][j] = whether s[:i] matches p[:j]

dp = [[False] \* (n + 1) for \_ in range(m + 1)]

dp[0][0] = True

# Patterns like a\*, a\*b\*, etc. can match empty string

for j in range(2, n + 1):

if p[j - 1] == '\*':

dp[0][j] = dp[0][j - 2]

for i in range(1, m + 1):

for j in range(1, n + 1):

if p[j - 1] == '\*':

# Two cases:

# 1) treat "x\*" as zero occurrences: drop x\* (j-2)

dp[i][j] = dp[i][j - 2]

# 2) if preceding pattern char matches current s char, consume one s char

# and keep pattern (because \* can match multiple)

if p[j - 2] == '.' or p[j - 2] == s[i - 1]:

dp[i][j] = dp[i][j] or dp[i - 1][j]

else:

# Direct match or '.' wildcard

if p[j - 1] == '.' or p[j - 1] == s[i - 1]:

dp[i][j] = dp[i - 1][j - 1]

return dp[m][n]

**cointainer with most water:**

class Solution:

def maxArea(self, height: List[int]) -> int:

left, right = 0, len(height) - 1

max\_area = 0

while left < right:

# Calculate the area

width = right - left

h = min(height[left], height[right])

area = width \* h

max\_area = max(max\_area, area)

# Move the pointer pointing to the shorter line

if height[left] < height[right]:

left += 1

else:

right -= 1

return max\_area

**integer to roman:**

class Solution:

def intToRoman(self, num: int) -> str:

# Mapping of values and symbols

val = [

1000, 900, 500, 400,

100, 90, 50, 40,

10, 9, 5, 4,

1

]

syms = [

"M", "CM", "D", "CD",

"C", "XC", "L", "XL",

"X", "IX", "V", "IV",

"I"

]

roman = ""

for i in range(len(val)):

while num >= val[i]:

num -= val[i]

roman += syms[i]

return roman

**longest common prefix:**

class Solution:

def longestCommonPrefix(self, strs: List[str]) -> str:

if not strs:

return ""

# Start with the first word as prefix

prefix = strs[0]

# Compare prefix with each string

for s in strs[1:]:

# Shrink the prefix until it matches the start of s

while not s.startswith(prefix):

prefix = prefix[:-1]

if not prefix:

return ""

return prefix

**3sum:**

class Solution:

def threeSum(self, nums: List[int]) -> List[List[int]]:

res = []

n = len(nums)

if n < 3:

return res

nums.sort()

for i in range(n - 2):

# Skip duplicate first elements

if i > 0 and nums[i] == nums[i - 1]:

continue

target = -nums[i]

l, r = i + 1, n - 1

while l < r:

s = nums[l] + nums[r]

if s == target:

res.append([nums[i], nums[l], nums[r]])

# Skip duplicates for l and r

while l < r and nums[l] == nums[l + 1]:

l += 1

while l < r and nums[r] == nums[r - 1]:

r -= 1

l += 1

r -= 1

elif s < target:

l += 1

else:

r -= 1

return res

**3sum closet:**

class Solution:

def threeSumClosest(self, nums: List[int], target: int) -> int:

nums.sort()

n = len(nums)

# Initialize closest\_sum with sum of first three (valid since n >= 3 on LeetCode)

closest\_sum = nums[0] + nums[1] + nums[2]

for i in range(n - 2):

l, r = i + 1, n - 1

while l < r:

cur\_sum = nums[i] + nums[l] + nums[r]

# If this is closer to target, update

if abs(cur\_sum - target) < abs(closest\_sum - target):

closest\_sum = cur\_sum

# Move pointers depending on comparison

if cur\_sum < target:

l += 1

elif cur\_sum > target:

r -= 1

else:

# Exact match

return target

return closest\_sum

**letter combination of a phone number:**

class Solution:

def letterCombinations(self, digits: str) -> List[str]:

if not digits:

return []

mapping = {

"2": "abc", "3": "def", "4": "ghi", "5": "jkl",

"6": "mno", "7": "pqrs", "8": "tuv", "9": "wxyz"

}

res = []

def backtrack(index: int, path: list):

# If the current combination is complete

if index == len(digits):

res.append("".join(path))

return

possible\_chars = mapping[digits[index]]

for ch in possible\_chars:

path.append(ch)

backtrack(index + 1, path)

path.pop()

backtrack(0, [])

return res

**4sum:**

class Solution:

def fourSum(self, nums: List[int], target: int) -> List[List[int]]:

nums.sort()

n = len(nums)

result = []

for i in range(n - 3):

# Skip duplicates for i

if i > 0 and nums[i] == nums[i - 1]:

continue

for j in range(i + 1, n - 2):

# Skip duplicates for j

if j > i + 1 and nums[j] == nums[j - 1]:

continue

left, right = j + 1, n - 1

while left < right:

total = nums[i] + nums[j] + nums[left] + nums[right]

if total == target:

result.append([nums[i], nums[j], nums[left], nums[right]])

# Skip duplicates for left

while left < right and nums[left] == nums[left + 1]:

left += 1

# Skip duplicates for right

while left < right and nums[right] == nums[right - 1]:

right -= 1

left += 1

right -= 1

elif total < target:

left += 1

else:

right -= 1

return result

**remove nth node from end of list:**

# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution:

def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

dummy = ListNode(0, head)

first = dummy

second = dummy

# Move first pointer n+1 steps ahead

for \_ in range(n + 1):

first = first.next

# Move both until first reaches the end

while first:

first = first.next

second = second.next

# Skip the nth node

second.next = second.next.next

return dummy.next

**valid parentheses:**

class Solution:

def isValid(self, s: str) -> bool:

stack = []

mapping = {')': '(', ']': '[', '}': '{'}

for char in s:

if char in mapping: # closing bracket

top\_element = stack.pop() if stack else '#'

if mapping[char] != top\_element:

return False

else: # opening bracket

stack.append(char)

return not stack

**merge two sorted lists:**

# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution:

def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

# Create a dummy node to simplify edge cases

dummy = ListNode(0)

current = dummy

while list1 and list2:

if list1.val < list2.val:

current.next = list1

list1 = list1.next

else:

current.next = list2

list2 = list2.next

current = current.next

# Attach remaining nodes

if list1:

current.next = list1

elif list2:

current.next = list2

return dummy.next

**generate parentheses:**

class Solution:

def generateParenthesis(self, n: int) -> List[str]:

result = []

def backtrack(current, open\_count, close\_count):

# If the current string is complete

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

result.append(current)

return

# Add '(' if we still have some left

if open\_count < n:

backtrack(current + "(", open\_count + 1, close\_count)

# Add ')' if it won't break validity

if close\_count < open\_count:

backtrack(current + ")", open\_count, close\_count + 1)

backtrack("", 0, 0)

return result