1. Container With Most Water

def max\_area(height):

"""

Finds the maximum area of water a container can hold.

Args:

height: A list of integers representing the height of walls.

Returns:

The maximum area of water the container can hold.

"""

left, right, max\_water = 0, len(height) - 1, 0

while left < right:

width = right - left

current\_height = min(height[left], height[right])

current\_area = width \* current\_height

max\_water = max(max\_water, current\_area)

if height[left] < height[right]:

left += 1

else:

right -= 1

return max\_water

# Example usage

height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

max\_water\_area = max\_area(height)

print(f"Maximum area of water: {max\_water\_area}")

1. Integer to Roman

def int\_to\_roman(num):

"""

Converts an integer to a Roman numeral.

Args:

num: An integer between 1 and 3999.

Returns:

The Roman numeral representation of the integer.

"""

# Define a dictionary mapping integer values to their corresponding Roman numerals

roman\_numerals = {

1000: "M",

900: "CM",

500: "D",

400: "CD",

100: "C",

90: "XC",

50: "L",

40: "XL",

10: "X",

9: "IX",

5: "V",

4: "IV",

1: "I"

}

result = ""

# Iterate through the dictionary in descending order

for value, numeral in sorted(roman\_numerals.items(), reverse=True):

# While the current value is less than or equal to the number,

# append the corresponding numeral to the result and subtract the value from the number

while num >= value:

result += numeral

num -= value

return result

# Example usage

num = 1994

roman\_numeral = int\_to\_roman(num)

print(f"{num} in Roman numeral: {roman\_numeral}")

1. Roman to Integer

def roman\_to\_int(s):

"""

Converts a Roman numeral to an integer.

Args:

s: A valid Roman numeral string.

Returns:

The integer equivalent of the Roman numeral.

"""

# Define a dictionary mapping Roman numerals to their corresponding integer values

roman\_values = {

"I": 1,

"V": 5,

"X": 10,

"L": 50,

"C": 100,

"D": 500,

"M": 1000

}

result = 0

prev\_value = 0

# Iterate through the Roman numeral string

for char in s:

# Get the integer value of the current character

current\_value = roman\_values[char]

# Handle subtractive notation (IV, IX, etc.)

if current\_value > prev\_value:

result += current\_value - prev\_value

prev\_value = 0

else:

# Add the current value to the result

result += current\_value

prev\_value = current\_value

return result

# Example usage

roman\_numeral = "MCMXCIV"

integer\_value = roman\_to\_int(roman\_numeral)

print(f"{roman\_numeral} in integer: {integer\_value}")

4. Longest Common Prefix

def longest\_common\_prefix(strs):

"""

Finds the longest common prefix string amongst an array of strings.

Args:

strs: A list of strings.

Returns:

The longest common prefix string.

"""

if not strs:

return ""

shortest\_str = min(strs, key=len) # Find the shortest string in the list

# Iterate through the shortest string and compare characters with other strings

for i in range(len(shortest\_str)):

for other\_str in strs:

if other\_str[i] != shortest\_str[i]:

return shortest\_str[:i] # Return the prefix up to the point of difference

return shortest\_str # All strings have the same prefix as the shortest string

# Example usage

strs = ["flower", "flow", "flight"]

longest\_prefix = longest\_common\_prefix(strs)

print(f"Longest common prefix: {longest\_prefix}")

5.3Sum

def three\_sum(nums):

"""

Finds all unique triplets in an array that sum to zero.

Args:

nums: A list of integers.

Returns:

A list of lists containing the unique triplets.

"""

# Sort the input array

nums.sort()

result = []

for i in range(len(nums) - 2):

# Skip duplicate elements (i.e., if the current element is the same as the previous one, continue)

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

continue

left, right = i + 1, len(nums) - 1

while left < right:

# Calculate the sum of the current triplet

current\_sum = nums[i] + nums[left] + nums[right]

# If the sum is zero, append the triplet to the result and move the pointers

if current\_sum == 0:

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

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

left += 1 # Skip duplicate elements in the left pointer

left += 1

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

right -= 1 # Skip duplicate elements in the right pointer

right -= 1

# Move the pointers based on the sum

elif current\_sum < 0:

left += 1

else:

right -= 1

return result

# Example usage

nums = [-1, 0, 1, 2, -1, -4]

triplets = three\_sum(nums)

print(f"Triplets that sum to zero: {triplets}")

6. 3Sum Closest

def three\_sum\_closest(nums, target):

"""

Finds the sum of three integers in an array that is closest to a given target.

Args:

nums: A list of integers.

target: The target sum.

Returns:

The sum of the three integers that is closest to the target.

"""

# Sort the input array

nums.sort()

closest\_sum = float('inf')

for i in range(len(nums) - 2):

# Skip duplicate elements (i.e., if the current element is the same as the previous one, continue)

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

continue

left, right = i + 1, len(nums) - 1

while left < right:

# Calculate the current sum

current\_sum = nums[i] + nums[left] + nums[right]

# Update the closest sum if the difference is smaller

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

closest\_sum = current\_sum

# Move the pointers based on the difference between current sum and target

if current\_sum < target:

left += 1

else:

right -= 1

return closest\_sum

# Example usage

nums = [-1, 2, 1, -4]

target = 1

closest\_sum = three\_sum\_closest(nums, target)

print(f"Sum closest to target: {closest\_sum}")

7. Letter Combinations of a Phone Number

def letter\_combinations(digits):

"""

Finds all possible letter combinations of a phone number.

Args:

digits: A string containing digits from 2-9.

Returns:

A list of strings representing all possible letter combinations.

"""

# Define a dictionary mapping digits to their corresponding letters

digit\_to\_letters = {

"2": "abc",

"3": "def",

"4": "ghi",

"5": "jkl",

"6": "mno",

"7": "pqrs",

"8": "tuv",

"9": "wxyz"

}

if not digits:

return [] # Return empty list if no digits provided

# Initialize an empty list to store the combinations

combinations = []

def backtrack(current\_combination, index):

"""

Recursive backtracking function to generate all combinations.

Args:

current\_combination: The current combination of letters being built.

index: The current index in the digits string.

"""

if index == len(digits):

combinations.append(current\_combination)

return

# Get the letters corresponding to the current digit

letters = digit\_to\_letters[digits[index]]

for letter in letters:

# Add the current letter to the combination and recursively explore possibilities for remaining digits

backtrack(current\_combination + letter, index + 1)

# Initiate backtracking starting from an empty string and index 0

backtrack("", 0)

return combinations

# Example usage

digits = "23"

combinations = letter\_combinations(digits)

print(f"Letter combinations for {digits}: {combinations}")

8. 4Sum

def four\_sum(nums, target):

"""

Finds all unique quadruplets in an array that sum to a given target.

Args:

nums: A list of integers.

target: The target sum.

Returns:

A list of lists containing the unique quadruplets.

"""

# Sort the input array to enable efficient two-pointer approach

nums.sort()

result = []

# Iterate through the array using a loop for the first element (a)

for i in range(len(nums) - 3):

# Skip duplicate elements (i.e., if the current element is the same as the previous one, continue)

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

continue

# Use two-pointer approach to find the remaining elements (b, c, d)

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

# Skip duplicate elements (j.e., if the current element is the same as the previous one, continue)

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

continue

left, right = j + 1, len(nums) - 1

while left < right:

# Calculate the current sum

current\_sum = nums[i] + nums[j] + nums[left] + nums[right]

# If the sum is equal to the target, append the quadruplet to the result and move the pointers

if current\_sum == target:

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

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

left += 1 # Skip duplicate elements in the left pointer

left += 1

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

right -= 1 # Skip duplicate elements in the right pointer

right -= 1

# Move the pointers based on the difference between current sum and target

elif current\_sum < target:

left += 1

else:

right -= 1

return result

# Example usage

nums = [1, 0, -1, 0, -2, 2]

target = 0

quadruplets = four\_sum(nums, target)

print(f"Quadruplets that sum to target: {quadruplets}")

9. Remove Nth Node From End of List

class ListNode:

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

self.val = val

self.next = next

class Solution:

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

"""

Removes the nth node from the end of a linked list and returns its head.

Args:

head: The head of the linked list.

n: The number of nodes to remove from the end.

Returns:

The head of the modified linked list.

"""

# Create a dummy node to simplify the edge case of removing the head node

dummy = ListNode(0)

dummy.next = head

# Find the node before the nth node from the end (slow pointer)

# Use a fast pointer that moves n steps ahead

fast = slow = dummy

for \_ in range(n + 1):

fast = fast.next

# Traverse the list until fast reaches the end

while fast:

slow = slow.next

fast = fast.next

# Remove the nth node by linking the slow node to its next next node

slow.next = slow.next.next

return dummy.next

# Example usage

head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

n = 2

result = Solution().removeNthFromEnd(head, n)

while result:

print(result.val, end=" -> ")

result = result.next

print("None")

10.  Valid Parentheses

def is\_valid(s):

"""

Checks if a string containing parentheses is valid.

Args:

s: A string containing parentheses.

Returns:

True if the string is valid, False otherwise.

"""

# Define a dictionary mapping opening brackets to their closing counterparts

bracket\_map = {

"(": ")",

"{": "}",

"[": "]"

}

stack = []

# Iterate through the string

for char in s:

# If the character is an opening bracket, push it onto the stack

if char in bracket\_map:

stack.append(char)

# If the character is a closing bracket, check if it matches the top element on the stack

elif char in bracket\_map.values():

if not stack or bracket\_map[stack.pop()] != char:

return False

# The string is valid if the stack is empty after processing all characters

return not stack

# Example usage

s1 = "()"

s2 = "()[]{}"

s3 = "(]"

print(f"Is '{s1}' valid? {is\_valid(s1)}")

print(f"Is '{s2}' valid? {is\_valid(s2)}")

print(f"Is '{s3}' valid? {is\_valid(s3)}")