**TWO SUM**

class Solution(object):

def twoSum(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: List[int]

"""

# A dictionary to store the difference and its corresponding index

num\_map = {}

for index, num in enumerate(nums):

# Calculate the difference needed to reach the target

difference = target - num

# If the difference is already in the map, return the indices

if difference in num\_map:

return [num\_map[difference], index]

# Otherwise, store the current number's index in the map

num\_map[num] = index

**ADD TWO NUMBERS**

# Definition for singly-linked list.

# class ListNode(object):

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

# self.val = val

# self.next = next

class Solution(object):

def addTwoNumbers(self, l1, l2):

"""

:type l1: Optional[ListNode]

:type l2: Optional[ListNode]

:rtype: Optional[ListNode]

"""

# Create a dummy node to simplify edge case handling

dummy = ListNode()

current = dummy # This will track the last node in the result

carry = 0 # This keeps track of carry-over in case of sum > 9

# Traverse both lists

while l1 or l2 or carry:

# Get the current values (0 if the list is shorter than the other)

val1 = l1.val if l1 else 0

val2 = l2.val if l2 else 0

# Sum the values along with any carry

total = val1 + val2 + carry

carry = total // 10 # Update carry (1 if total >= 10, otherwise 0)

current.next = ListNode(total % 10) # Create a new node for the result

# Move to the next node in the result list

current = current.next

# Move to the next nodes in the input lists, if available

if l1: l1 = l1.next

if l2: l2 = l2.next

return dummy.next # Return the result list (skip the dummy node)

**LENGTH OF LONGEST SUBSTRING**

class Solution(object):

def lengthOfLongestSubstring(self, s):

char\_index = {}

left = 0

max\_length = 0

for right in range(len(s)):

if s[right] in char\_index and char\_index[s[right]] >= left:

left = char\_index[s[right]] + 1

char\_index[s[right]] = right

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

return max\_length

# Example tests

sol = Solution()

print(sol.lengthOfLongestSubstring("abcabcbb")) # Expected 3 ("abc")

print(sol.lengthOfLongestSubstring("bbbbb")) # Expected 1 ("b")

print(sol.lengthOfLongestSubstring("pwwkew")) # Expected 3 ("wke")

print(sol.lengthOfLongestSubstring("")) # Expected 0

**FIND MEDIAN ARRAYS**

class Solution(object):

def findMedianSortedArrays(self, nums1, nums2):

"""

:type nums1: List[int]

:type nums2: List[int]

:rtype: float

"""

A, B = nums1, nums2

total = len(A) + len(B)

half = total // 2

if len(A) > len(B):

A, B = B, A

l, r = 0, len(A) - 1

while True:

i = (l + r) // 2 # partition for A

j = half - i - 2 # partition for B

Aleft = A[i] if i >= 0 else float("-inf")

Aright = A[i + 1] if (i + 1) < len(A) else float("inf")

Bleft = B[j] if j >= 0 else float("-inf")

Bright = B[j + 1] if (j + 1) < len(B) else float("inf")

if Aleft <= Bright and Bleft <= Aright:

if total % 2:

return min(Aright, Bright)

return (max(Aleft, Bleft) + min(Aright, Bright)) / 2.0

elif Aleft > Bright:

r = i - 1

else:

l = i + 1

# Example usage:

sol = Solution()

print(sol.findMedianSortedArrays([1, 3], [2])) # Output: 2.0

print(sol.findMedianSortedArrays([1, 2], [3, 4])) # Output: 2.5

**LONGEST POLINDROME**

class Solution(object):

def longestPalindrome(self, s):

"""

:type s: str

:rtype: str

"""

if not s:

return ""

start, end = 0, 0

def expandFromCenter(left, right):

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

left -= 1

right += 1

return left + 1, right - 1 # return the bounds of the palindrome

for i in range(len(s)):

# Odd length palindrome

l1, r1 = expandFromCenter(i, i)

# Even length palindrome

l2, r2 = expandFromCenter(i, i + 1)

# Choose the longer one

if r1 - l1 > end - start:

start, end = l1, r1

if r2 - l2 > end - start:

start, end = l2, r2

return s[start:end + 1]

# Example usage:

sol = Solution()

print(sol.longestPalindrome("babad")) # Output: "bab" or "aba"

print(sol.longestPalindrome("cbbd")) # Output: "bb"

**REVERSE INTRGER**

class Solution(object):

def reverse(self, x):

"""

:type x: int

:rtype: int

"""

sign = -1 if x < 0 else 1

x = abs(x)

rev = 0

while x != 0:

pop = x % 10

x //= 10

rev = rev \* 10 + pop

rev \*= sign

# Handle 32-bit signed integer overflow

if rev < -2\*\*31 or rev > 2\*\*31 - 1:

return 0

return rev

# Example usage:

sol = Solution()

print(sol.reverse(123)) # Output: 321

print(sol.reverse(-123)) # Output: -321

print(sol.reverse(120)) # Output: 21

print(sol.reverse(1534236469)) # Output: 0 (overflow)

**STRING TO INTEGER**

class Solution(object):

def myAtoi(self, s):

"""

:type s: str

:rtype: int

"""

i, n, sign, result = 0, len(s), 1, 0

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

# 1. Skip leading whitespace

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

i += 1

# 2. Check for sign

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

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

i += 1

# 3. Convert digits to integer

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

digit = int(s[i])

# 4. Handle 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

# Example usage:

sol = Solution()

print(sol.myAtoi("42")) # Output: 42

print(sol.myAtoi(" -42")) # Output: -42

print(sol.myAtoi("4193 with words")) # Output: 4193

print(sol.myAtoi("words and 987")) # Output: 0

print(sol.myAtoi("-91283472332")) # Output: -2147483648 (clamped)

**POLINDROME NUMBER**

class Solution(object):

def isPalindrome(self, x):

"""

:type x: int

:rtype: bool

"""

s = str(x)

return s == s[::-1]

# Example usage:

sol = Solution()

print(sol.isPalindrome(121)) # True

print(sol.isPalindrome(-121)) # False

print(sol.isPalindrome(10)) # False

**REGULAR EXPRESSION MATCHING**

class Solution(object):

def isMatch(self, s, p):

"""

:type s: str

:type p: str

:rtype: bool

"""

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

# dp[i][j] means s[:i] matches p[:j]

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

dp[0][0] = True

# Handle patterns like a\*, a\*b\*, a\*b\*c\* that can match an empty string

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

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

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

# Build the DP table

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

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

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

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

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

# Two cases:

# 1️⃣ Zero occurrence of preceding element

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

# 2️⃣ One or more occurrences (if preceding matches current)

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

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

return dp[m][n]

# Example usage:

sol = Solution()

print(sol.isMatch("aa", "a")) # False

print(sol.isMatch("aa", "a\*")) # True

print(sol.isMatch("ab", ".\*")) # True

print(sol.isMatch("aab", "c\*a\*b")) # True

print(sol.isMatch("mississippi", "mis\*is\*p\*.")) # False

**CONTAINER WITH MOST WATER**

class Solution(object):

def maxArea(self, height):

"""

:type height: List[int]

:rtype: int

"""

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

max\_area = 0

while left < right:

# Calculate the area formed by the two lines

width = right - left

area = min(height[left], height[right]) \* width

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

# Move the pointer that limits the height

if height[left] < height[right]:

left += 1

else:

right -= 1

return max\_area

# Example usage:

sol = Solution()

print(sol.maxArea([1,8,6,2,5,4,8,3,7])) # Output: 49

print(sol.maxArea([1,1])) # Output: 1

**INTEGER TO ROMAN**

class Solution(object):

def intToRoman(self, num):

"""

:type num: int

:rtype: str

"""

# Mapping of integer values to Roman numerals

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]:

roman += syms[i]

num -= val[i]

return roman

# Example usage:

sol = Solution()

print(sol.intToRoman(3)) # Output: "III"

print(sol.intToRoman(58)) # Output: "LVIII" (50 + 5 + 3)

print(sol.intToRoman(1994)) # Output: "MCMXCIV" (1000 + 900 + 90 + 4)

**ROMAN TO INTEGER**

class Solution(object):

def romanToInt(self, s):

"""

:type s: str

:rtype: int

"""

roman = {

'I': 1,

'V': 5,

'X': 10,

'L': 50,

'C': 100,

'D': 500,

'M': 1000

}

total = 0

prev\_value = 0

for ch in reversed(s):

value = roman[ch]

# If a smaller value precedes a larger one, subtract it

if value < prev\_value:

total -= value

else:

total += value

prev\_value = value

return total

# Example usage:

sol = Solution()

print(sol.romanToInt("III")) # Output: 3

print(sol.romanToInt("LVIII")) # Output: 58

print(sol.romanToInt("MCMXCIV")) # Output: 1994

**LONGEST COMMON PREFIX**

class Solution(object):

def longestCommonPrefix(self, strs):

"""

:type strs: List[str]

:rtype: str

"""

if not strs:

return ""

# Start with the first string as the prefix

prefix = strs[0]

for s in strs[1:]:

# Reduce prefix until it matches the start of string s

while not s.startswith(prefix):

prefix = prefix[:-1]

if not prefix:

return ""

return prefix

# Example usage:

sol = Solution()

print(sol.longestCommonPrefix(["flower","flow","flight"])) # Output: "fl"

print(sol.longestCommonPrefix(["dog","racecar","car"])) # Output: ""

print(sol.longestCommonPrefix(["interspecies","interstellar","interstate"])) # Output: "inters"

**3 SUM CLOSEST**

class Solution(object):

def threeSumClosest(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: int

"""

nums.sort()

n = len(nums)

closest\_sum = float('inf')

for i in range(n - 2):

left, right = i + 1, n - 1

while left < right:

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

# Update closest\_sum if this sum is closer to target

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

closest\_sum = current\_sum

if current\_sum < target:

left += 1

elif current\_sum > target:

right -= 1

else:

# Exact match

return target

return closest\_sum

# Example usage:

sol = Solution()

print(sol.threeSumClosest([-1,2,1,-4], 1)) # Output: 2 (-1 + 2 + 1 = 2)

print(sol.threeSumClosest([0,0,0], 1)) # Output: 0

**LETTER COMBINATONS OF A PHONE**

class Solution(object):

def letterCombinations(self, digits):

"""

:type digits: str

:rtype: List[str]

"""

if not digits:

return []

phone\_map = {

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

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

}

res = []

def backtrack(index, path):

# If the path length equals digits length, append to results

if index == len(digits):

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

return

for char in phone\_map[digits[index]]:

path.append(char)

backtrack(index + 1, path)

path.pop()

backtrack(0, [])

return res

# Example usage:

sol = Solution()

print(sol.letterCombinations("23"))

# Output: ['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf']

print(sol.letterCombinations(""))

# Output: []

**REMOVE NTH NODE FROM END OF LIST**

# Definition for singly-linked list.

# class ListNode(object):

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

# self.val = val

# self.next = next

class Solution(object):

def removeNthFromEnd(self, head, n):

"""

:type head: Optional[ListNode]

:type n: int

:rtype: Optional[ListNode]

"""

dummy = ListNode(0)

dummy.next = head

first = dummy

second = dummy

# Move first pointer n+1 steps ahead

for \_ in range(n + 1):

first = first.next

# Move first to the end, maintaining the gap

while first:

first = first.next

second = second.next

# Remove the nth node from end

second.next = second.next.next

return dummy.next