1. TWO SUM

class Solution(object):

    def twoSum(self, nums, target):

        """

        :type nums: List[int]

        :type target: int

        :rtype: List[int]

        """

        num\_map = {}  # value -> index

        for i, num in enumerate(nums):

            complement = target - num

            if complement in num\_map:

                return [num\_map[complement], i]

            num\_map[num] = i

        # Should never reach here if a valid solution always exists

        return []

1. ADD OF 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]

"""

dummy = ListNode() # Dummy node to simplify list construction

current = dummy

carry = 0

while l1 or l2 or carry:

# Get current values (0 if list has ended)

val1 = l1.val if l1 else 0

val2 = l2.val if l2 else 0

# Compute sum and carry

total = val1 + val2 + carry

carry = total // 10

current.next = ListNode(total % 10)

# Move pointers forward

current = current.next

if l1:

l1 = l1.next

if l2:

l2 = l2.next

return dummy.next

1. LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS

class Solution(object):

def lengthOfLongestSubstring(self, s):

"""

:type s: str

:rtype: int

"""

char\_index = {} # stores last seen index of each character

left = 0 # left pointer of the window

max\_length = 0

for right, char in enumerate(s):

# if character is repeated, move the left pointer

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

left = char\_index[char] + 1

# update last seen index of the current character

char\_index[char] = right

# update the max length

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

return max\_length

1. MEDIAN OF TWO SORTED ARRAYS

class Solution(object):

def findMedianSortedArrays(self, nums1, nums2):

"""

:type nums1: List[int]

:type nums2: List[int]

:rtype: float

"""

# Ensure nums1 is the smaller array to minimize binary search range

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

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')

if left1 <= right2 and left2 <= right1:

# Found correct partition

if total % 2 == 0:

return (max(left1, left2) + min(right1, right2)) / 2.0

else:

return float(min(right1, right2))

elif left1 > right2:

# Move partition left in nums1

right = i - 1

else:

# Move partition right in nums1

left = i + 1

1. LONGEST PALINDROMIC SUBSTRING

class Solution(object):

def longestPalindrome(self, s):

"""

:type s: str

:rtype: str

"""

if not s or len(s) == 1:

return s

start, end = 0, 0

def expandAroundCenter(left, right):

# expand while the characters match and are in bounds

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

left -= 1

right += 1

# return the length of palindrome found

return right - left - 1

for i in range(len(s)):

# check for odd-length palindrome (single center)

len1 = expandAroundCenter(i, i)

# check for even-length palindrome (two centers)

len2 = expandAroundCenter(i, i + 1)

max\_len = max(len1, len2)

if max\_len > end - start:

start = i - (max\_len - 1) // 2

end = i + max\_len // 2

return s[start:end + 1]

1. ZIGZAG CONVERSION

class Solution(object):

def convert(self, s, numRows):

"""

:type s: str

:type numRows: int

:rtype: str

"""

# Edge case: if only one row, zigzag = original string

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

return s

# Create a list of strings for each row

rows = [''] \* numRows

current\_row = 0

going\_down = False

# Traverse the string, placing each character in the correct row

for char in s:

rows[current\_row] += char

# Change direction at top or bottom

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

going\_down = not going\_down

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

# Join all rows to get the final converted string

return ''.join(rows)

1. REVERSE INTEGER

class Solution(object):

def reverse(self, x):

"""

:type x: int

:rtype: int

"""

sign = -1 if x < 0 else 1

x = abs(x)

# Reverse digits using string manipulation

reversed\_x = int(str(x)[::-1]) \* sign

# Check for 32-bit signed integer overflow

if reversed\_x < -2\*\*31 or reversed\_x > 2\*\*31 - 1:

return 0

return reversed\_x

1. STRING TO INTEGER(atoi)

class Solution(object):

def myAtoi(self, s):

"""

:type s: str

:rtype: int

"""

s = s.strip() # Remove leading and trailing spaces

if not s:

return 0

sign = 1

i = 0

result = 0

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

INT\_MIN = -2\*\*31

# Check the sign

if s[0] in ['-', '+']:

sign = -1 if s[0] == '-' else 1

i += 1

# Convert characters to integer until non-digit is found

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

digit = int(s[i])

# Check for overflow before adding digit

if (result > INT\_MAX // 10) or (result == INT\_MAX // 10 and digit > INT\_MAX % 10):

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

result = result \* 10 + digit

i += 1

return result \* sign

1. PALINDROME NUMBER

class Solution(object):

def isPalindrome(self, x):

"""

:type x: int

:rtype: bool

"""

# Negative numbers and numbers ending with 0 (except 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 numbers: x == reversed\_half

# For odd-length numbers: x == reversed\_half // 10 (ignore middle digit)

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

1. 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] = does s[:i] match p[:j]

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

dp[0][0] = True # empty string matches empty pattern

# 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]

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]:

# Characters match directly or via '.'

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

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

# '\*' can represent zero or more of the preceding char

dp[i][j] = dp[i][j - 2] # zero occurrence

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

dp[i][j] |= dp[i - 1][j] # one or more occurrences

return dp[m][n]