**Two sum:**

class Solution(object):

          :type nums: List[int]

        :type target: int

    def twoSum(self, nums, target):

      :rtype: List[int]

        num\_map = {}  # Dictionary to store number and its index

        for i, num in enumerate(nums):

            complement = target - num

            if complement in num\_map:

                return [num\_map[complement], i]

            num\_map;

**Add two numbers:**

# 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\_head = ListNode()

        current = dummy\_head

        carry = 0

        while l1 or l2 or carry:

            val1 = l1.val if l1 else 0

            val2 = l2.val if l2 else 0

# Calculate sum and new carry

            total\_sum = val1 + val2 + carry

            carry = total\_sum // 10

            digit = total\_sum % 10

            # Create new node for the result list

            current.next = ListNode(digit)

            current = current.next

            # to the next nodes in l1 and l2

            if l1:

                l1 = l1.next

            if l2:

                l2 = l2.next

**Longest Substring Without repeating character:**

class Solution(object):

    def lengthOfLongestSubstring(self, s):

   :type s: str

        :rtype: int:

          if not s:

            return 0

        # `char\_set` will store characters currently in the sliding window.

        char\_set = set()

                # `max\_length` will store the maximum length found so far.

        max\_length = 0

                # `left` is the left pointer of the sliding window.

        left = 0

           # `right` is the right pointer of the sliding window, iterating through the string.

        for right in range(len(s)):

            # If the character at the `right` pointer is already in our set,

            # it means we have a duplicate. We need to shrink the window

            # from the `left` until the duplicate is removed.

            while s[right] in char\_set:

                char\_set.remove(s[left])

                left += 1

                        # Once `s[right]` is not in the set (or after removing duplicates),

            # add `s[right]` to the set.

            char\_set.add(s[right])

                        # The current length of the substring without repeating characters

            # is `right - left + 1`. Update `max\_length` if this is larger.

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

                    return max\_length

[**Median of Two Sorted Arrays**](https://leetcode.com/problems/median-of-two-sorted-arrays/)

class Solution(object):

    def findMedianSortedArrays(self, nums1, nums2):

              :type nums1: List[int]

        :type nums2: List[int]

        :rtype: float

        if len(nums1) > len(nums2):

            nums1, nums2 = nums2, nums1

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

        low, high = 0, m

        half\_len = (m + n + 1) // 2

                while low <= high:

            partitionX = (low + high) // 2

            partitionY = half\_len - partitionX

            max\_left\_X = nums1[partitionX - 1] if partitionX != 0 else float('-inf')

            min\_right\_X = nums1[partitionX] if partitionX != m else float('inf')

                        max\_left\_Y = nums2[partitionY - 1] if partitionY != 0 else float('-inf')

            min\_right\_Y = nums2[partitionY] if partitionY != n else float('inf')

                # the median is the average of the two middle elements.

                # These are the largest in the left half and the smallest in the right half.

                else:

                    return float(max(max\_left\_X, max\_left\_Y) + min(min\_right\_X, min\_right\_Y)) / 2.0

                       # If the partition is not correct, adjust `low` or `high` for binary search:

            # If `max\_left\_X` is greater than `min\_right\_Y`, it means `partitionX` is too far to the right.

            elif max\_left\_X > min\_right\_Y:

return 0.0

[Longest Palindromic Substring](https://leetcode.com/problems/longest-palindromic-substring/)

class Solution(object):

    def longestPalindrome(self, s):

               :type s: str

        :rtype: str

               if not s:

            return

# Initialize the longest palindrome found so far.

        # At minimum, a single character is a palindrome.

        longest\_palindromic\_substring = s[0]

        def expand\_and\_find\_palindrome(left, right):

            # Expand outwards as long as characters match and pointers are within bounds.

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

                left -= 1

                right += 1

        # So, the valid palindrome starts at `left + 1` and ends at `right - 1`.

        for i in range(len(s)):

            # Case 1: Odd length palindromes (e.g., "aba", center is s[i])

            odd\_length\_palindrome = expand\_and\_find\_palindrome(i, i)

            if len(odd\_length\_palindrome) > len(longest\_palindromic\_substring):

                longest\_palindromic\_substring = odd\_length\_palindrome

                        # Case 2: Even length palindromes (e.g., "abba", center is between s[i] and s[i+1])

            even\_length\_palindrome = expand\_and\_find\_palindrome(i, i + 1)

            if len(even\_length\_palindrome) > len(longest\_palindromic\_substring):

                longest\_palindromic\_substring = even\_length\_palindrome

                return longest\_palindromic\_substring

[Reverse Integer](https://leetcode.com/problems/reverse-integer/)

class Solution(object):

    def reverse(self, x):

               :type x: int

        :rtype: int

               INT\_MAX = 2\*\*31 - 1

        INT\_MIN = -2\*\*31

        negative = x < 0

        abs\_x = abs(x)

        reversed\_num = 0

        while abs\_x > 0:

            digit = abs\_x % 10

                     return 0

                if not negative and digit > INT\_MAX % 10: # for positive INT\_MAX, last digit is 7, so if current digit is > 7, it overflows

                    return 0

            reversed\_num = reversed\_num \* 10 + digit

            abs\_x //= 10

        if negative:

                return 0

            return -reversed\_num

        else:

                        # (This check is partially redundant with the loop's checks but good for clarity).

            if reversed\_num > INT\_MAX:

                    return 0

[String to Integer (atoi)](https://leetcode.com/problems/string-to-integer-atoi/)

class Solution(object):

    def myAtoi(self, s):

              :type s: str

        :rtype: int

               i = 0

        n = len(s)

        # 1. Read and ignore leading whitespace

        while i < n and s[i] == ' ':

            i += 1

                # If all characters are whitespace or string is empty, return 0

        if i == n:

            return 0

                    # 2. Check for sign

        sign = 1

        if s[i] == '+':

            i += 1

        elif s[i] == '-':

            sign = -1

            i += 1

                    # 3. Read digits and convert

        result = 0

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

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

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

            digit = int(s[i])

                        # 4. Handle overflow \*before\* adding the current digit.

            # We accumulate 'result' as an absolute (positive) value first.

                        # This happens if result > abs(INT\_MIN) // 10 OR (result == abs(INT\_MIN) // 10 AND digit > 8)

            else: # sign == -1

                if result > abs(INT\_MIN) // 10 or (result == abs(INT\_MIN) // 10 and digit > 8):

                    return INT\_MIN

                        result = result \* 10 + digit

            i += 1

                    # 5. Apply the determined sign

        final\_result = result \* sign

                # The in-loop overflow checks should already handle cases that exceed INT\_MAX or INT\_MIN.

        # This final clamp is a safeguard but should ideally not be hit for values that \*would\* overflow.

        # For values that fit exactly, like INT\_MAX or INT\_MIN, this passes them through.

                # Example: s = "2147483647", result becomes 2147483647, sign is 1. final\_result = 2147483647.

        # Example: s = "-2147483648", result becomes 2147483648, sign is -1. final\_result = -2147483648.

        return final\_result

Palindrome number

class Solution(object):

    def isPalindrome(self, x):

        """

        :type x: int

        :rtype: bool

        """

        # Negative numbers cannot be palindromes (e.g., -121 reads 121- backwards)

        # Numbers ending in 0 (except for 0 itself) cannot be palindromes

        # (e.g., 10 reads 01 backwards, 200 reads 002 backwards)

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

            return False

        # If x is 0, it's a palindrome

        if x == 0:

            return True

        reversed\_half = 0

        # We only need to reverse the second half of the number.

        # We stop when reversed\_half becomes greater than or equal to x.

        while x > reversed\_half:

            digit = x % 10

            reversed\_half = reversed\_half \* 10 + digit

            x //= 10

        # When the number has an even number of digits, x and reversed\_half will be equal.

        # Example: x = 1221 -> x becomes 12, reversed\_half becomes 12.

[Regular Expression Matching](https://leetcode.com/problems/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 = [[False] \* (n + 1) for \_ in range(m + 1)]

        dp[0][0] = True

        # Deals with patterns like a\*, a\*b\*, a\*b\*c\* etc.

        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]:

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

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

                    # Zero occurrence of the character before '\*'

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

                    # One or more occurrence if preceding character matches current s char

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

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

                else:

                    dp[i][j] = False

        return dp[m][n]

[Integer to Roman](https://leetcode.com/problems/integer-to-roman/)

class Solution(object):

    def intToRoman(self, num):

        """

        :type num: int

        :rtype: str

        """

        # Define the Roman numeral symbols and their corresponding integer values.

        # It's crucial to list special subtractive cases (like 900, 400, 90, 40, 9, 4)

        # before their larger components (like 1000, 500, 100, 50, 10, 5) to ensure

        # the greedy approach works correctly.

        roman\_map = [

            (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 = [] # Use a list to build the string efficiently

                # Iterate through the Roman numeral mappings

        for value, symbol in roman\_map:

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

            # append the symbol to the result and subtract the value from num.

            while num >= value:

                result.append(symbol)

                num -= value

                        # Join the list of symbols to form the final Roman numeral string.

        return "".join(result)