1. **[Two Sum](https://leetcode.com/problems/two-sum/)**:

class Solution(object):

    def twoSum(self, nums, target):

        """

        :type nums: List[int]

        :type target: int

        :rtype: List[int]

        """

        n = len(nums)

        for i in range(n):

            for j in range(i + 1, n): # Start j from i + 1 to avoid duplicate pairs and self-pairing

                if nums[i] + nums[j] == target:

                    return [i, j]

        # This line should ideally not be reached if a solution is guaranteed

        return []

1. [**Add Two Numbers**](https://leetcode.com/problems/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 head node for the result linked list.

        # This simplifies handling the head of the list.

        dummy\_head = ListNode(0)

        # `current` pointer will traverse and build the new list.

        current = dummy\_head

        # `carry` variable to store any carry-over from adding digits.

        carry = 0

        # Loop continues as long as there are digits in either list

        # or there is a carry-over remaining.

        while l1 is not None or l2 is not None or carry != 0:

            # Get the value of the current digit from l1 (or 0 if l1 is exhausted).

            val1 = l1.val if l1 is not None else 0

            # Get the value of the current digit from l2 (or 0 if l2 is exhausted).

            val2 = l2.val if l2 is not None else 0

            # Calculate the sum of the current digits plus the carry.

            total\_sum = val1 + val2 + carry

            # Determine the new carry-over.

            carry = total\_sum // 10

            # Determine the digit to be placed in the current node.

            digit = total\_sum % 10

            # Create a new node with the calculated digit and append it to the result list.

            current.next = ListNode(digit)

            # Move the `current` pointer to the newly added node.

            current = current.next

            # Move l1 and l2 pointers to their next nodes if they exist.

            if l1 is not None:

                l1 = l1.next

            if l2 is not None:

                l2 = l2.next

        # The result linked list starts from `dummy\_head.next`

        # as `dummy\_head` was just a placeholder.

        return dummy\_head.next

1. [**Longest Substring Without Repeating Characters**](https://leetcode.com/problems/longest-substring-without-repeating-characters/):

class Solution(object):

    def lengthOfLongestSubstring(self, s):

        """

        :type s: str

        :rtype: int

        """

        char\_set = set()

        max\_length = 0

        left = 0

        for right in range(len(s)):

            while s[right] in char\_set:

                char\_set.remove(s[left])

                left += 1

            char\_set.add(s[right])

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

        return max\_length

1. [**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

        """

        merged = sorted(nums1 + nums2)

        total\_len = len(merged)

        if total\_len % 2 == 1:

            # Odd length: median is the middle element

            return float(merged[total\_len // 2])

        else:

            # Even length: median is the average of the two middle elements

            mid1 = merged[total\_len // 2 - 1]

            mid2 = merged[total\_len // 2]

            return (float(mid1) + float(mid2)) / 2.0

1. [**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 ""

        longest\_palindrome\_str = ""

        max\_length = 0

        n = len(s)

        # Iterate over all possible starting points

        for i in range(n):

            # Iterate over all possible ending points (from current start to end of string)

            for j in range(i, n):

                substring = s[i : j + 1]

                # Check if the current substring is a palindrome

                if substring == substring[::-1]:

                    # If it's a palindrome and longer than the current longest, update

                    if len(substring) > max\_length:

                        max\_length = len(substring)

                        longest\_palindrome\_str = substring

        return longest\_palindrome\_str

1. [**Zigzag Conversion**](https://leetcode.com/problems/zigzag-conversion/):

class Solution(object):

    def convert(self, s, numRows):

        """

        :type s: str

        :type numRows: int

        :rtype: str

        """

        # Edge case: If there's only one row, or the string is empty,

        # no zigzag conversion is needed.

        if numRows == 1 or not s:

            return s

        # Create a list of strings, one for each row.

        # Each string will accumulate characters for its respective row.

        rows = [""] \* numRows

        current\_row = 0

        # direction determines whether we are moving down (1) or up (-1)

        # We start by moving down.

        direction = 1

        # Iterate through each character in the input string

        for char in s:

            # Append the current character to the string of the current\_row

            rows[current\_row] += char

            # Check if we need to change direction:

            # If we hit the bottom row (numRows - 1), we must change direction to go up.

            if current\_row == numRows - 1:

                direction = -1

            # If we hit the top row (0), we must change direction to go down.

            elif current\_row == 0:

                direction = 1

            # Move to the next row based on the current direction

            current\_row += direction

        # After processing all characters, join all row strings together

        # to form the final converted zigzag string.

        return "".join(rows)

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

class Solution(object):

    def reverse(self, x):

        """

        :type x: int

        :rtype: int

        """

        # Define the 32-bit signed integer range

        # MAX\_INT = 2^31 - 1 = 2147483647

        # MIN\_INT = -2^31 = -2147483648

        MAX\_INT = 2\*\*31 - 1

        MIN\_INT = -2\*\*31

        # Determine the sign

        is\_negative = x < 0

        # Get the absolute value for reversal

        abs\_x = abs(x)

        # Convert to string, reverse, and convert back to int

        reversed\_abs\_x\_str = str(abs\_x)[::-1]

        reversed\_num = int(reversed\_abs\_x\_str)

        # Apply the original sign

        if is\_negative:

            reversed\_num \*= -1

        # Check for overflow

        if not (MIN\_INT <= reversed\_num <= MAX\_INT):

            return 0

        return reversed\_num

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

class Solution(object):

    def myAtoi(self, s):

        """

        :type s: str

        :rtype: int

        """

        # Define the 32-bit signed integer range

        MAX\_INT = 2\*\*31 - 1  # 2147483647

        MIN\_INT = -2\*\*31     # -2147483648

        i = 0

        n = len(s)

        # 1. Read and ignore leading whitespace

        while i < n and s[i] == ' ':

            i += 1

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

        if i == n:

            return 0

        # 2. Check for sign

        sign = 1 # Default to positive

        if s[i] == '-':

            sign = -1

            i += 1

        elif s[i] == '+':

            sign = 1

            i += 1

        result = 0

        # 3. Read digits and 4. Convert to integer

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

            digit = int(s[i])

            # 5. Handle overflow

            # We accumulate 'result' as a positive magnitude and apply sign at the end.

            # Therefore, we check against MAX\_INT for positive results and abs(MIN\_INT) for negative.

            if sign == 1: # For a potentially positive final number

                # Check if result \* 10 + digit would exceed MAX\_INT

                if result > MAX\_INT // 10 or (result == MAX\_INT // 10 and digit > MAX\_INT % 10):

                    return MAX\_INT

            else: # For a potentially negative final number (check against abs(MIN\_INT))

                # Check if result \* 10 + digit would exceed the magnitude of MIN\_INT (which is 2^31)

                # Note: abs(MIN\_INT) = 2147483648

                if result > abs(MIN\_INT) // 10 or (result == abs(MIN\_INT) // 10 and digit > abs(MIN\_INT) % 10):

                    return MIN\_INT

            result = result \* 10 + digit

            i += 1

        # Apply the final sign

        return result \* sign

[**9. Palindrome Number**](https://leetcode.com/problems/palindrome-number/) :

class Solution(object):

    def isPalindrome(self, x):

        """

        :type x: int

        :rtype: bool

        """

        # Negative numbers are not palindromes (e.g., -121 is not 121-)

        # Numbers ending in 0 (but not 0 itself) cannot be palindromes (e.g., 10, 120)

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

            return False

        reverted\_number = 0

        while x > reverted\_number:

            reverted\_number = reverted\_number \* 10 + x % 10

            x //= 10

        # When the number has an odd number of digits, the middle digit

        # is stored in `x` but doesn't affect the palindrome check.

        # For example, if x = 12321, at the end of the loop, x becomes 12, and reverted\_number becomes 123.

        # We can ignore the middle digit by dividing reverted\_number by 10.

        return x == reverted\_number or x == reverted\_number // 10

[**10. 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 = len(s)**

**n = len(p)**

**# dp[i][j] will be True if s[:i] matches p[:j]**

**# We use m+1 and n+1 for easier 0-based string/pattern access (s[i-1], p[j-1])**

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

**# Base case: Empty string matches empty pattern**

**dp[0][0] = True**

**# Initialize for empty string s:**

**# s="" can match patterns like "a\*", "a\*b\*", etc.**

**# This means p[j-1] must be '\*' and p[:j-2] must match ""**

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

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

**# If the current pattern char is '\*', it can match zero of the preceding element.**

**# So, if p[:j-2] matched an empty string, p[:j] (which is p[:j-2] + x\*) can also match an empty string.**

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

**# No other pattern can match an empty string s, so dp[0][j] remains False by default**

**# Fill the DP table**

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

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

**# If current pattern character is a literal character or '.'**

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

**# '.' matches any single character. So if s[i-1] is matched by p[j-1],**

**# the rest of the strings must have matched.**

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

**elif p[j - 1] == s[i - 1]:**

**# If current characters are equal, they match.**

**# The rest of the strings must have matched.**

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

**# If current pattern character is '\*'**

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

**# Case 1: '\*' matches zero occurrences of the preceding element (p[j-2])**

**# In this case, we effectively ignore p[j-2] and p[j-1] ("x\*").**

**# The match depends on whether s[:i] matches p[:j-2].**

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

**# Case 2: '\*' matches one or more occurrences of the preceding element (p[j-2])**

**# For this to be possible, s[i-1] must match p[j-2]**

**# (i.e., s[i-1] == p[j-2] or p[j-2] == '.').**

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

**# If s[i-1] matches p[j-2], then '\*' can consume s[i-1].**

**# The remaining s[:i-1] then needs to match p[:j] (as '\*' can still match more).**

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

**# If characters don't match and p[j-1] is not '\*', dp[i][j] remains False (default value)**

**return dp[m][n]**