Set Matrix Zeros

```python
def set_zeros(matrix):
    rows, cols = len(matrix), len(matrix[0])
    zero_rows, zero_cols = set(), set()

    # Find all rows and columns that contain zeros
    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 0:
                zero_rows.add(i)
                zero_cols.add(j)

    # Set entire rows to zero
    for row in zero_rows:
        for j in range(cols):
            matrix[row][j] = 0

    # Set entire columns to zero
    for col in zero_cols:
        for i in range(rows):
            matrix[i][col] = 0

    return matrix

matrix = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 8, 9]
]

result = set_zeros(matrix)
for row in result:
    print(row)
```

Pascal's Triangle

```python
def generate_pascal_triangle(num_rows):
    triangle = []
    for row_num in range(num_rows):
        row = [None for _ in range(row_num + 1)]
        row[0], row[-1] = 1, 1  # First and last elements are always 1
        for j in range(1, len(row) - 1):
            row[j] = triangle[row_num - 1][j - 1] + triangle[row_num - 1][j]
        triangle.append(row)
    return triangle

def print_pascal_triangle(triangle):
```

```python
    for row in triangle:
        print(" ".join(map(str, row)))

# Example usage:
num_rows = 5
triangle = generate_pascal_triangle(num_rows)
print_pascal_triangle(triangle)
```

Next Permutation

```python
def next_permutation(nums):
    # Step 1: Find the largest index k such that nums[k] < nums[k+1]
    k = len(nums) - 2
    while k >= 0 and nums[k] >= nums[k + 1]:
        k -= 1
    # Step 2: Find the largest index l greater than k such that nums[k] <
nums[l]
    if k >= 0:
        l = len(nums) - 1
        while l > k and nums[l] <= nums[k]:
            l -= 1
        # Swap nums[k] and nums[l]
        nums[k], nums[l] = nums[l], nums[k]

    # Step 3: Reverse the sequence from nums[k+1] onwards
    left, right = k + 1, len(nums) - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1

nums = [1, 2, 3]
next_permutation(nums)
print(nums)  # Output: [1, 3, 2]

nums = [3, 2, 1]
next_permutation(nums)
print(nums)  # Output: [1, 2, 3]

nums = [1, 1, 5]
next_permutation(nums)
print(nums)  # Output: [1, 5, 1]

nums = [1]
next_permutation(nums)
print(nums)  # Output: [1]
```

Kadane's Algorithm

```python
def kadane(nums):
    if not nums:
        return 0

    current_max = nums[0]
    global_max = nums[0]

    for i in range(1, len(nums)):
        current_max = max(nums[i], current_max + nums[i])
        if current_max > global_max:
            global_max = current_max

    return global_max

# Example usage:
nums1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("Maximum contiguous sum:", kadane(nums1))  # Output: 6 (corresponding
to [4, -1, 2,   1])

nums2 = [-1, -2, -3, -4]
print("Maximum contiguous sum:", kadane(nums2))  # Output: -1 (corresponding
to [-1])

nums3 = [1, 2, 3, 4, 5]
print("Maximum contiguous sum:", kadane(nums3))  # Output: 15 (entire array
itself)
```

Sort an array of 0's, 1's and 2's

```python
def sort_colors(nums):
    # Initialize pointers for the Dutch National Flag algorithm
    low, mid, high = 0, 0, len(nums) - 1

    # Traverse the array
    while mid <= high:
        if nums[mid] == 0:
            # Swap nums[low] and nums[mid]
            nums[low], nums[mid] = nums[mid], nums[low]
            # Move both low and mid pointers to the right
            low += 1
            mid += 1
        elif nums[mid] == 1:
            # No swap needed, just move mid pointer
            mid += 1
        else:  # nums[mid] == 2
            # Swap nums[mid] and nums[high]
```

```python
            nums[mid], nums[high] = nums[high], nums[mid]
            # Move high pointer to the left
            high -= 1

    return nums

# Example usage:
nums = [2, 0, 2, 1, 1, 0]
print(sort_colors(nums))  # Output: [0, 0, 1, 1, 2, 2]
```

Stock Buy and Sell

```python
def max_profit(prices):
    if not prices or len(prices) < 2:
        return 0

    min_price = prices[0]  # Initialize minimum price to the first element
    max_profit = 0  # Initialize maximum profit to 0

    for price in prices:
        if price < min_price:
            min_price = price  # Update the minimum price if a lower price
is found
        else:
            # Calculate current profit
            current_profit = price - min_price
            # Update max profit if current profit is greater
            if current_profit > max_profit:
                max_profit = current_profit

    return max_profit

# Example usage:
prices = [7, 1, 5, 3, 6, 4]
print(max_profit(prices))  # Output: 5 (Buy at 1 and sell at 6)

prices = [7, 6, 4, 3, 1]
print(max_profit(prices))  # Output: 0 (No transaction possible)

prices = [2, 4, 1]
print(max_profit(prices))  # Output: 2 (Buy at 2 and sell at 4)
```

Rotate Matrix

```python
def rotate(matrix):
```

```python
    n = len(matrix)

    # Step 1: Transpose the matrix
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # Step 2: Reverse each row
    for i in range(n):
        matrix[i].reverse()

    return matrix

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

rotated_matrix = rotate(matrix)
for row in rotated_matrix:
    print(row)
```

Merge Overlapping Subintervals

```python
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort intervals based on the start time
    intervals.sort(key=lambda x: x[0])

    merged = []
    for interval in intervals:
        # If the list of merged intervals is empty or if the current
        # interval does not overlap with the previous, simply append it.
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            # There is overlap, so merge the current and previous intervals
            merged[-1][1] = max(merged[-1][1], interval[1])

    return merged

# Example usage:
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
```

```python
print(merge_intervals(intervals))  # Output: [[1, 6], [8, 10], [15, 18]]

intervals = [[1, 4], [4, 5]]
print(merge_intervals(intervals))  # Output: [[1, 5]]
```

Merge two sorted arrays without extra space

```python
def merge(nums1, m, nums2, n):
    # Initialize pointers for nums1, nums2 and the end of nums1
    i = m - 1
    j = n - 1
    k = m + n - 1

    # Merge nums1 and nums2 from the end
    while i >= 0 and j >= 0:
        if nums1[i] > nums2[j]:
            nums1[k] = nums1[i]
            i -= 1
        else:
            nums1[k] = nums2[j]
            j -= 1
        k -= 1

    # If there are remaining elements in nums2, append them to nums1
    while j >= 0:
        nums1[k] = nums2[j]
        j -= 1
        k -= 1

# Example usage:
nums1 = [1, 2, 3, 0, 0, 0]
m = 3
nums2 = [2, 5, 6]
n = 3

merge(nums1, m, nums2, n)
print(nums1)  # Output: [1, 2, 2, 3, 5, 6]
```

Find the duplicate in an array of N+1 integers

```python
def find_duplicate(nums):
    # Step 1: Find the intersection point of the two pointers
    slow = nums[0]
    fast = nums[nums[0]]

    while slow != fast:
```

```python
        slow = nums[slow]
        fast = nums[nums[fast]]

    # Step 2: Find the duplicate number
    slow = 0
    while slow != fast:
        slow = nums[slow]
        fast = nums[fast]

    return slow

# Example usage:
nums = [1, 3, 4, 2, 2]
print(find_duplicate(nums))  # Output: 2

nums = [3, 1, 3, 4, 2]
print(find_duplicate(nums))  # Output: 3
```

Repeat and Missing Number

```python
def find_repeat_missing(nums):
    n = len(nums)

    # Calculate the sum and sum of squares for 1 to N
    sum_of_n = n * (n + 1) // 2
    sum_of_sq_n = n * (n + 1) * (2 * n + 1) // 6

    # Calculate the sum and sum of squares for nums array
    sum_actual = sum(nums)
    sum_sq_actual = sum(num*num for num in nums)

    # Difference between sum_of_n and sum_actual gives missing_number
    missing_number = sum_of_n - sum_actual

    # Difference between sum_of_sq_n and sum_sq_actual gives repeat_number^2
    repeat_number_sq = sum_of_sq_n - sum_sq_actual

    # Find repeat_number from the square
    repeat_number = int((repeat_number_sq + missing_number * missing_number)
/ missing_number) // 2

    return repeat_number, missing_number

# Example usage:
nums = [4, 3, 2, 7, 8, 2, 1, 5]
repeat, missing = find_repeat_missing(nums)
```

```
  print("Repeat number:", repeat)   # Output: 2
  print("Missing number:", missing) # Output: 6
```

Inversion of Array (Pre-req: Merge Sort)

```
  def merge_count_split_inv(arr, temp_arr, left,   mid, right):
      i = left    # Starting index for left subarray
    j = mid + 1  # Starting index for right   subarray
      k = left    # Starting index to be sorted
      inv_count = 0

    # Conditions are checked to ensure that i   doesn't exceed mid and j
doesn't exceed right
      while i <= mid and j <= right:
          if arr[i] <= arr[j]:
              temp_arr[k] = arr[i]
              i += 1
          else:
            # There are mid - i inversions, because all the remaining elements
in   the left subarray
            # (arr[i], arr[i+1], ..., arr[mid])   are greater than arr[j]
              temp_arr[k] = arr[j]
              inv_count += (mid-i + 1)
              j += 1
          k += 1

    # Copy the remaining elements of left   subarray, if any
      while i <= mid:
          temp_arr[k] = arr[i]
          i += 1
          k += 1

    # Copy the remaining elements of right   subarray, if any
      while j <= right:
          temp_arr[k] = arr[j]
          j += 1
          k += 1

      # Copy the sorted subarray into Original array
      for i in range(left, right + 1):
          arr[i] = temp_arr[i]

      return inv_count

def merge_sort_and_count(arr, temp_arr, left,   right):
      inv_count = 0
      if left < right:
```

```python
        mid = (left + right)//2

        inv_count += merge_sort_and_count(arr,   temp_arr, left, mid)
        inv_count += merge_sort_and_count(arr,   temp_arr, mid + 1, right)
        inv_count += merge_count_split_inv(arr,   temp_arr, left, mid, right)

    return inv_count

def count_inversions(arr):
    n = len(arr)
    temp_arr = [0]*n
  return merge_sort_and_count(arr, temp_arr, 0,   n-1)

# Example usage:
arr = [1, 20, 6, 4, 5]
print("Number of inversions:", count_inversions  (arr))  # Output: 5
```

Search in a 2 D matrix

```python
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False

    rows = len(matrix)
    cols = len(matrix[0])

    # Binary search to find the correct row
    low, high = 0, rows - 1
    while low <= high:
        mid = (low + high) // 2
      if matrix[mid][0] <= target <= matrix[mid]  [cols - 1]:
          # Perform binary search in the current   row
            left, right = 0, cols - 1
            while left <= right:
                col_mid = (left + right) // 2
                if matrix[mid][col_mid] == target:
                    return True
                elif matrix[mid][col_mid] < target:
                    left = col_mid + 1
                else:
                    right = col_mid - 1
            return False
        elif target < matrix[mid][0]:
            high = mid - 1
        else:
            low = mid + 1
```

```python
    return False

# Example usage:
matrix = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
]
target = 3
print(search_matrix(matrix, target))  # Output:   True

target = 13
print(search_matrix(matrix, target))  # Output:   False
```

Pow(x, n)

```python
def myPow(x, n):
    if n == 0:
        return 1.0
    elif n < 0:
        return 1.0 / myPow(x, -n)
    else:
        half = myPow(x, n // 2)
        if n % 2 == 0:
            return half * half
        else:
            return half * half * x

# Example usage:
x = 2.0
n = 10
print(myPow(x, n))  # Output: 1024.0

x = 2.1
n = 3
print(myPow(x, n))  # Output: 9.261000000000001

x = 2.0
n = -2
print(myPow(x, n))  # Output: 0.25
```

Majority Element (>n/2 times)

```python
def majorityElement(nums):
    candidate = None
    count = 0
```

```
    # First pass: Find the candidate
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)

    # Second pass: Verify the candidate
    count = 0
    for num in nums:
        if num == candidate:
            count += 1
  #The candidate is guaranteed to be the majority element by the problem
statement
    return candidate

# Example usage:
nums = [3, 2, 3]
print(majorityElement(nums))  # Output: 3

nums = [2, 2, 1, 1, 1, 2, 2]
print(majorityElement(nums))  # Output: 2
```

Majority Element (n/3 times)

```
def majorityElement(nums):
    if not nums:
        return []

    candidate1, candidate2 = None, None
    count1, count2 = 0, 0

    # Step 1: Find potential candidates
    for num in nums:
        if num == candidate1:
            count1 += 1
        elif num == candidate2:
            count2 += 1
        elif count1 == 0:
            candidate1 = num
            count1 = 1
        elif count2 == 0:
            candidate2 = num
            count2 = 1
        else:
            count1 -= 1
            count2 -= 1
```

```
    # Step 2: Verify candidates
    count1, count2 = 0, 0
    for num in nums:
        if num == candidate1:
            count1 += 1
        elif num == candidate2:
            count2 += 1

  # Step 3: Determine if candidates are majority   elements
    result = []
    if count1 > len(nums) // 3:
        result.append(candidate1)
    if count2 > len(nums) // 3:
        result.append(candidate2)

    return result

# Example usage:
nums = [3, 2, 3]
print(majorityElement(nums))  # Output: [3]

nums = [1, 1, 1, 3, 3, 2, 2, 2]
print(majorityElement(nums))  # Output: [1, 2]
```

Grid Unique Paths

```
def uniquePaths(m, n):
    # Initialize a 2D DP table with all zeros
    dp = [[0] * n for _ in range(m)]

    # Base case: There is exactly one way to be at the starting point
    dp[0][0] = 1

    # Fill the DP table
    for i in range(m):
        for j in range(n):
            if i > 0:
                dp[i][j] += dp[i-1][j]  # Add paths from above
            if j > 0:
                dp[i][j] += dp[i][j-1]  # Add paths from left

    # The result is the number of unique paths to reach bottom-right corner
    return dp[m-1][n-1]

# Example usage:
m = 3
```

```python
n = 7
print(uniquePaths(m, n))  # Output: 28

m = 3
n = 2
print(uniquePaths(m, n))  # Output: 3
```

Reverse Pairs (Leetcode)

```python
def reversePairs(nums):
    def merge_count_split_inv(nums, temp_arr, left, right):
        if left >= right:
            return 0

        mid = (left + right) // 2
        count = merge_count_split_inv(nums, temp_arr, left, mid) +
merge_count_split_inv(nums, temp_arr, mid + 1, right)

        # Counting the pairs
        j = mid + 1
        for i in range(left, mid + 1):
            while j <= right and nums[i] > 2 * nums[j]:
                j += 1
            count += (j - (mid + 1))

        # Merge two sorted subarrays
        i, j, k = left, mid + 1, left
        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp_arr[k] = nums[i]
                i += 1
            else:
                temp_arr[k] = nums[j]
                j += 1
            k += 1

        while i <= mid:
            temp_arr[k] = nums[i]
            i += 1
            k += 1

        while j <= right:
            temp_arr[k] = nums[j]
            j += 1
            k += 1

        for i in range(left, right + 1):
```

```
            nums[i] = temp_arr[i]

        return count

    if not nums:
        return 0

    temp_arr = [0] * len(nums)
    return merge_count_split_inv(nums, temp_arr, 0, len(nums) - 1)

# Example usage:
nums = [1, 3, 2, 3, 1]
print(reversePairs(nums))  # Output: 2

nums = [2, 4, 3, 5, 1]
print(reversePairs(nums))  # Output: 3
```

2Sum Problem

```
def twoSum(nums, target):
    num_map = {}

    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map:
            return [num_map[complement], i]
        num_map[num] = i

    return []  # If no solution found

# Example usage:
nums = [2, 7, 11, 15]
target = 9
print(twoSum(nums, target))  # Output: [0, 1] (indices of elements that sum up to 9)

nums = [3, 2, 4]
target = 6
print(twoSum(nums, target))  # Output: [1, 2] (indices of elements that sum up to 6)
```

4-Sum Problem

```
def fourSum(nums, target):
    nums.sort()
    n = len(nums)
```

```
    result = []

    for i in range(n - 3):
        # Skip duplicates
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        for j in range(i + 1, n - 2):
            # Skip duplicates
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue

            left, right = j + 1, n - 1

            while left < right:
                current_sum = nums[i] + nums[j] + nums[left] + nums[right]

                if current_sum == target:
                    result.append([nums[i], nums[j], nums[left],
nums[right]])
                    left += 1
                    right -= 1
                    # Skip duplicates
                    while left < right and nums[left] == nums[left - 1]:
                        left += 1
                    while left < right and nums[right] == nums[right + 1]:
                        right -= 1
                elif current_sum < target:
                    left += 1
                else:
                    right -= 1

    return result

# Example usage:
nums = [1, 0, -1, 0, -2, 2]
target = 0
print(fourSum(nums, target))  # Output: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1,
0, 0, 1]]
```

Longest Consecutive Sequence

```
def longestConsecutive(nums):
    if not nums:
        return 0

    num_set = set(nums)
```

```
    max_length = 0

    for num in num_set:
        if num - 1 not in num_set:  # Check if num is the start of a sequence
            current_num = num
            current_length = 1

            while current_num + 1 in num_set:
                current_num += 1
                current_length += 1

            max_length = max(max_length, current_length)

    return max_length

# Example usage:
nums = [100, 4, 200, 1, 3, 2]
print(longestConsecutive(nums))  # Output: 4 (The longest consecutive sequence is [1, 2, 3, 4])
```

Largest Subarray with K sum

```
def maxSubArrayLen(nums, k):
    prefix_sum_map = {}
    prefix_sum_map[0] = -1  # Initialize with a prefix sum of 0 at index -1
    max_len = 0
    current_sum = 0

    for i in range(len(nums)):
        current_sum += nums[i]
        target_sum = current_sum - k

        if target_sum in prefix_sum_map:
            max_len = max(max_len, i - prefix_sum_map[target_sum])

        if current_sum not in prefix_sum_map:
            prefix_sum_map[current_sum] = i

    return max_len

# Example usage:
nums = [1, -1, 5, -2, 3]
k = 3
print(maxSubArrayLen(nums, k))  # Output: 4 (The subarray [1, -1, 5, -2] has sum 3 and is the longest)
```

Count number of subarrays with given xor K

```python
def countSubarraysWithXOR(nums, K):
    prefix_xor_map = {}
    prefix_xor_map[0] = 1  # Initialize with prefix XOR 0 with frequency 1
    current_xor = 0
    count = 0

    for num in nums:
        current_xor ^= num

        desired_xor = current_xor ^ K

        if desired_xor in prefix_xor_map:
            count += prefix_xor_map[desired_xor]

        if current_xor in prefix_xor_map:
            prefix_xor_map[current_xor] += 1
        else:
            prefix_xor_map[current_xor] = 1

    return count

# Example usage:
nums = [4, 2, 2, 6, 4]
K = 6
print(countSubarraysWithXOR(nums, K))  # Output: 4 (The subarrays with XOR 6 are [4, 2], [4, 2, 2, 6], [2, 2, 6, 4], [6])
```

Longest Substring without repeat

```python
def lengthOfLongestSubstring(s):
    char_map = {}
    max_len = 0
    start = 0

    for end in range(len(s)):
        char = s[end]

        if char in char_map and char_map[char] >= start:
            start = char_map[char] + 1

        char_map[char] = end
        max_len = max(max_len, end - start + 1)

    return max_len

# Example usage:
```

```
s = "abcabcbb"
print(lengthOfLongestSubstring(s))  # Output: 3 ("abc" or "bca" or "cab" are the longest substrings without repeating characters)
```

Reverse a LinkedList

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseLinkedList(head):
    if not head or not head.next:
        return head

    prev = None
    curr = head

    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    return prev

# Function to print the linked list
def printLinkedList(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage:
# Create a linked list: 1 -> 2 -> 3 -> 4 -> None
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
print("Original linked list:")
printLinkedList(head)

# Reverse the linked list
reversed_head = reverseLinkedList(head)
print("\nReversed linked list:")
printLinkedList(reversed_head)
```

Find the middle of LinkedList

```python
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

# Function to print the linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Example usage:
# Creating a sample linked list: 1 -> 2 -> 3 -> 4 -> 5
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)

print("Linked List:")
print_linked_list(head)

middle_node = find_middle(head)
print(f"The middle element of the linked list is: {middle_node.value}")
```

Merge two sorted Linked List (use method used in mergeSort)

```python
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

def merge_sorted_lists(head1, head2):
    # Base cases
    if not head1:
        return head2
    if not head2:
```

```python
        return head1

    # Determine the smaller head between head1 and head2
    if head1.value <= head2.value:
        merged_head = head1
        merged_head.next = merge_sorted_lists(head1.next, head2)
    else:
        merged_head = head2
        merged_head.next = merge_sorted_lists(head1, head2.next)

    return merged_head

# Function to print the linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Example usage:
# Creating two sorted linked lists: 1 -> 3 -> 5 and 2 -> 4 -> 6 -> 7
head1 = ListNode(1)
head1.next = ListNode(3)
head1.next.next = ListNode(5)

head2 = ListNode(2)
head2.next = ListNode(4)
head2.next.next = ListNode(6)
head2.next.next.next = ListNode(7)

print("First sorted linked list:")
print_linked_list(head1)
print("Second sorted linked list:")
print_linked_list(head2)

merged_head = merge_sorted_lists(head1, head2)
print("Merged sorted linked list:")
print_linked_list(merged_head)
```

Remove N-th node from back of LinkedList

```python
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next
```

```python
def remove_nth_from_end(head, n):
    # Dummy node to handle edge cases
    dummy = ListNode(0)
    dummy.next = head
    slow = fast = dummy

    # Move the fast pointer n steps ahead
    for _ in range(n):
        fast = fast.next

    # Move both pointers until fast reaches the end
    while fast.next:
        slow = slow.next
        fast = fast.next

    # Remove the N-th node from the end
    slow.next = slow.next.next
    return dummy.next

# Function to print the linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Example usage:
# Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)

print("Original linked list:")
print_linked_list(head)

n = 2
head = remove_nth_from_end(head, n)
print(f"Linked list after removing {n}-th node   from end:")
print_linked_list(head)
```

Add two numbers as LinkedList

```python
class ListNode:
    def __init__(self, val=0, next=None):
```

```python
        self.val = val
        self.next = next

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    dummy_head = ListNode(0)
    current = dummy_head
    carry = 0

    while l1 or l2:
        x = l1.val if l1 else 0
        y = l2.val if l2 else 0
        total = carry + x + y
        carry = total // 10
        current.next = ListNode(total % 10)
        current = current.next
        if l1: l1 = l1.next
        if l2: l2 = l2.next

    if carry > 0:
        current.next = ListNode(carry)

    return dummy_head.next

# Helper function to print the linked list
def printList(node: ListNode):
    while node:
        print(node.val, end=" -> " if node.next else "\n")
        node = node.next

# Helper function to create a linked list from a list of integers
def createList(lst):
    dummy_root = ListNode(0)
    ptr = dummy_root
    for number in lst:
        ptr.next = ListNode(number)
        ptr = ptr.next
    return dummy_root.next

# Example usage
l1 = createList([2, 4, 3])
l2 = createList([5, 6, 4])

print("List 1:")
printList(l1)

print("List 2:")
printList(l2)
```

```python
result = addTwoNumbers(l1, l2)

print("Result:")
printList(result)
```

Delete a given Node when a node is given.(0(1) solution)

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def deleteNode(node: ListNode):
    if node is None or node.next is None:
        return  # Cannot delete the node if it is None or the last node

    # Copy the value from the next node
    node.val = node.next.val
    # Link the current node to the node after the next node
    node.next = node.next.next

# Helper function to print the linked list
  def printList(node: ListNode):
      while node:
          print(node.val, end=" -> " if node.next else "\n")
          node = node.next

  # Helper function to create a linked list from a list of integers
  def createList(lst):
      dummy_root = ListNode(0)
      ptr = dummy_root
      for number in lst:
          ptr.next = ListNode(number)
          ptr = ptr.next
      return dummy_root.next

  # Example usage
  lst = [4, 5, 1, 9]
  head = createList(lst)

  print("Original List:")
  printList(head)

  # Assume we want to delete the node with value 5
  # To do this, we need a reference to the node with value 5
  node_to_delete = head.next  # Node with value 5
  deleteNode(node_to_delete)
```

```
print("List after deleting node with value 5:")
printList(head)
```

Find intersection point of Y LinkedList

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    if not headA or not headB:
        return None

    pointerA, pointerB = headA, headB

# Traverse both lists. Once a pointer reaches the end, move it to the start
of the other list.
    while pointerA != pointerB:
        pointerA = pointerA.next if pointerA else headB
        pointerB = pointerB.next if pointerB else headA

    return pointerA

# Helper function to print the linked list
def printList(node: ListNode):
    while node:
        print(node.val, end=" -> " if node.next else "\n")
        node = node.next

# Helper function to create a linked list from a list of integers
def createList(lst):
    dummy_root = ListNode(0)
    ptr = dummy_root
    for number in lst:
        ptr.next = ListNode(number)
        ptr = ptr.next
    return dummy_root.next

# Example usage
# Create two linked lists that intersect
# List A: 4 -> 1 -> 8 -> 4 -> 5
# List B: 5 -> 6 -> 1 -> 8 -> 4 -> 5
headA = createList([4, 1])
headB = createList([5, 6, 1])
intersection = createList([8, 4, 5])
```

```python
# Attach the intersection part to both lists
lastA = headA
while lastA.next:
    lastA = lastA.next
lastA.next = intersection

lastB = headB
while lastB.next:
    lastB = lastB.next
lastB.next = intersection

# Find the intersection
intersection_node = getIntersectionNode(headA, headB)

# Print the result
print("List A:")
printList(headA)

print("List B:")
printList(headB)

if intersection_node:
    print(f"Intersection at node with value: {intersection_node.val}")
else:
    print("No intersection")
```

Detect a cycle in Linked List

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def hasCycle(head: ListNode) -> bool:
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next
```

```python
    return True

# Helper function to create a linked list from a list of integers
def createList(lst, pos=-1):
    dummy_root = ListNode(0)
    ptr = dummy_root
    cycle_entry = None
    cycle_node = None
    for index, number in enumerate(lst):
        ptr.next = ListNode(number)
        ptr = ptr.next
        if index == pos:
            cycle_entry = ptr
    if pos != -1 and cycle_entry:
        ptr.next = cycle_entry
    return dummy_root.next

# Example usage
# List: 3 -> 2 -> 0 -> -4 (with a cycle that connects the last node back to
the   second node)
head = createList([3, 2, 0, -4], 1)

if hasCycle(head):
    print("Cycle detected in the linked list.")
else:
    print("No cycle detected in the linked list.")
```

Reverse a LinkedList in groups of size k.

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseKGroup(head: ListNode, k: int) -> ListNode:
    if head is None or k == 1:
        return head

    dummy = ListNode(0)
    dummy.next = head
    current = dummy
    nex = dummy
    pre = dummy

    count = 0
    while current.next is not None:
        current = current.next
```

```python
        count += 1

    while count >= k:
        current = pre.next
        nex = current.next
        for _ in range(1, k):
            current.next = nex.next
            nex.next = pre.next
            pre.next = nex
            nex = current.next
        pre = current
        count -= k

    return dummy.next

# Helper function to print the linked list
def printList(node: ListNode):
    while node:
        print(node.val, end=" -> " if node.next else "\n")
        node = node.next

# Helper function to create a linked list from a list of integers
def createList(lst):
    dummy_root = ListNode(0)
    ptr = dummy_root
    for number in lst:
        ptr.next = ListNode(number)
        ptr = ptr.next
    return dummy_root.next

# Example usage
# List: 1 -> 2 -> 3 -> 4 -> 5 and k = 3
head = createList([1, 2, 3, 4, 5])
k = 3

print("Original List:")
printList(head)

reversed_head = reverseKGroup(head, k)

print(f"List after reversing in groups of {k}:")
printList(reversed_head)
```

Check if a LinkedList is palindrome or not.

```python
class ListNode:
    def __init__(self, val=0, next=None):
```

```python
        self.val = val
        self.next = next

def isPalindrome(head: ListNode) -> bool:
    if head is None or head.next is None:
        return True

    # Step 1: Find the middle of the linked list
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse the second half of the linked list
    prev = None
    current = slow
    while current:
        next_temp = current.next
        current.next = prev
        prev = current
        current = next_temp

    # Step 3: Compare the two halves
    first_half = head
    second_half = prev
    while second_half:
        if first_half.val != second_half.val:
            return False
        first_half = first_half.next
        second_half = second_half.next

    return True

# Helper function to print the linked list
def printList(node: ListNode):
    while node:
        print(node.val, end=" -> " if node.next else "\n")
        node = node.next

# Helper function to create a linked list from a list of integers
def createList(lst):
    dummy_root = ListNode(0)
    ptr = dummy_root
    for number in lst:
        ptr.next = ListNode(number)
        ptr = ptr.next
    return dummy_root.next
```

```python
# Example usage
# List: 1 -> 2 -> 2 -> 1
head = createList([1, 2, 2, 1])

print("Original List:")
printList(head)

if isPalindrome(head):
    print("The linked list is a palindrome.")
else:
    print("The linked list is not a palindrome.")
```

Find the starting point of the Loop of LinkedList

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def detectCycle(head: ListNode) -> ListNode:
    if not head or not head.next:
        return None

    slow = head
    fast = head

    # Phase 1: Detect cycle
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            break

    if not fast or not fast.next:
        return None

    # Phase 2: Find the starting point of the cycle
    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return slow

# Helper function to create a linked list from a list of integers
def createList(lst, pos=-1):
```

```python
    dummy_root = ListNode(0)
    ptr = dummy_root
    cycle_entry = None
    cycle_node = None
    for index, number in enumerate(lst):
        ptr.next = ListNode(number)
        ptr = ptr.next
        if index == pos:
            cycle_entry = ptr
    if pos != -1 and cycle_entry:
        ptr.next = cycle_entry
    return dummy_root.next

# Helper function to print the linked list (useful for debugging, but won't
work   with cycles)
def printList(node: ListNode):
    seen = set()
    while node and node not in seen:
        print(node.val, end=" -> " if node.next else "\n")
        seen.add(node)
        node = node.next
    if node:
        print(f"Cycle detected starting at node with value: {node.val}")

# Example usage
# List: 3 -> 2 -> 0 -> -4 (with a cycle that connects the last node back to
the   second node)
head = createList([3, 2, 0, -4], 1)

print("Original List (with cycle):")
printList(head)

cycle_node = detectCycle(head)
if cycle_node:
    print(f"The cycle starts at node with value: {cycle_node.val}")
else:
    print("No cycle detected in the linked list.")
```

Flattening of a LinkedList

```python
class ListNode:
    def __init__(self, val=0, next=None, child=None):
        self.val = val
        self.next = next
        self.child = child

def flatten(head: ListNode) -> ListNode:
```

```python
    if not head:
        return head

    # Pointer to the current node
    curr = head

    while curr:
        if curr.child:
            # Save the next pointer
            next = curr.next

            # Recursively flatten the child list
            child = flatten(curr.child)

            # Insert the flattened child list between curr and next
            curr.next = child

            # Find the tail of the flattened child list
            while child.next:
                child = child.next

            # Reconnect the tail to the next node
            child.next = next

            # Remove the child pointer
            curr.child = None

        # Move to the next node
        curr = curr.next
    return head

# Helper function to create a linked list from a nested list
def createList(nested_list):
    if not nested_list:
        return None

    head = ListNode(nested_list[0])
    curr = head
    for i in range(1, len(nested_list)):
        if isinstance(nested_list[i], list):
            curr.child = createList(nested_list[i])
        else:
            curr.next = ListNode(nested_list[i])
            curr = curr.next

    return head

# Helper function to print the flattened linked list
```

```python
def printList(head: ListNode):
    while head:
        print(head.val, end=" -> " if head.next else "\n")
        head = head.next

# Example usage
# List: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> None
#       |    |
#       7 -> 8 -> 9 -> 10 -> None
#            |
#            11 -> 12 -> None
nested_list = [1, [7], 2, [8, [11, 12], 9, 10], 3, 4, 5, 6]
head = createList(nested_list)

print("Original List:")
printList(head)

flattened_head = flatten(head)

print("Flattened List:")
printList(flattened_head)
```

Rotate a LinkedList

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def rotateRight(head: ListNode, k: int) -> ListNode:
    if not head or not head.next or k == 0:
        return head

    # Step 1: Compute the length of the list and get the last node
    length = 1
    current = head
    while current.next:
        current = current.next
        length += 1

    # Step 2: Connect the last node to the head, making it a circular linked
list
    current.next = head

    # Step 3: Find the point to break the circle
    k = k % length
    steps_to_new_head = length - k
```

```python
    # Step 4: Move to the new head and break the circle
    new_tail = head
    for _ in range(steps_to_new_head - 1):
        new_tail = new_tail.next
    new_head = new_tail.next
    new_tail.next = None

    return new_head

# Helper function to create a linked list from a list of integers
def createList(lst):
    dummy_root = ListNode(0)
    ptr = dummy_root
    for number in lst:
        ptr.next = ListNode(number)
        ptr = ptr.next
    return dummy_root.next

# Helper function to print the linked list
def printList(node: ListNode):
    while node:
        print(node.val, end=" -> " if node.next else "\n")
        node = node.next

# Example usage
# List: 1 -> 2 -> 3 -> 4 -> 5
head = createList([1, 2, 3, 4, 5])
k = 2

print("Original List:")
printList(head)

rotated_head = rotateRight(head, k)

print(f"List after rotating by {k} positions:")
printList(rotated_head)
```

Clone a Linked List with random and next pointer

```python
class Node:
    def __init__(self, val=0, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random

def cloneList(head: Node) -> Node:
```

```
    if not head:
        return None

    # Step 1: Create new nodes and interweave them with the original list
    current = head
    while current:
        new_node = Node(current.val, current.next)
        current.next = new_node
        current = new_node.next

    # Step 2: Set the random pointers for the new nodes
    current = head
    while current:
        if current.random:
            current.next.random = current.random.next
        current = current.next.next

    # Step 3: Separate the original list and the copied list
    current = head
    new_head = head.next
    while current:
        copy = current.next
        current.next = copy.next
        if copy.next:
            copy.next = copy.next.next
        current = current.next

    return new_head

# Helper function to create a linked list from a list of tuples (val,
random_index)
def createList(values_with_random):
    if not values_with_random:
        return None

    # Create nodes and a map to access nodes by index
    nodes = [Node(val) for val, _ in values_with_random]
    for i, (val, random_index) in enumerate(values_with_random):
        if i < len(values_with_random) - 1:
            nodes[i].next = nodes[i + 1]
        if random_index is not None:
            nodes[i].random = nodes[random_index]

    return nodes[0]

# Helper function to print the linked list
def printList(head: Node):
    current = head
```

```
    while current:
        random_val = current.random.val if current.random else None
        print(f"Node(val={current.val}, random={random_val})", end=" -> " if
current.  next else "\n")
        current = current.next

# Example usage
# List: 1 -> 2 -> 3 with random pointers:
# 1.random -> 3, 2.random -> 1, 3.random -> 2
values_with_random = [(1, 2), (2, 0), (3, 1)]
head = createList(values_with_random)

print("Original List:")
printList(head)

cloned_head = cloneList(head)

print("Cloned List:")
printList(cloned_head)
```

3 sum

```
def threeSum(nums):
    nums.sort()
    result = []

    for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i - 1]:
            continue  # skip duplicate numbers for the first number in the
triplet

        left, right = i + 1, len(nums) - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            if total < 0:
                left += 1
            elif total > 0:
                right -= 1
            else:
                result.append([nums[i], nums[left], nums[right]])
                while left < right and nums[left] == nums[left + 1]:
                  left += 1  # skip duplicate numbers for the second number
in the triplet
                while left < right and nums[right] == nums[right - 1]:
                  right -= 1  # skip duplicate numbers for the third number
in the triplet
                left += 1
```

```python
                right -= 1

    return result

# Example usage
nums = [-1, 0, 1, 2, -1, -4]
print("Input array:", nums)
result = threeSum(nums)
print("Unique triplets that sum to zero:")
for triplet in result:
    print(triplet)
```

Trapping Rainwater

```python
def trap(height):
    if not height:
        return 0

    left, right = 0, len(height) - 1
    left_max, right_max = height[left], height[right]
    water_trapped = 0

    while left < right:
        if left_max < right_max:
            left += 1
            left_max = max(left_max, height[left])
            water_trapped += max(0, left_max - height[left])
        else:
            right -= 1
            right_max = max(right_max, height[right])
            water_trapped += max(0, right_max - height[right])

    return water_trapped

# Example usage
height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
print("Height array:", height)
print("Water trapped:", trap(height))
```

Remove Duplicate from Sorted array

```python
def removeDuplicates(nums):
    if not nums:
        return 0

    # Pointer for the place to insert the next unique element
```

```python
    insert_pos = 1

    for i in range(1, len(nums)):
        if nums[i] != nums[i - 1]:
            nums[insert_pos] = nums[i]
            insert_pos += 1

    return insert_pos

# Example usage
nums = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]
print("Original array:", nums)
new_length = removeDuplicates(nums)
print("Array after removing duplicates:", nums[:new_length])
print("New length:", new_length)
```

Max consecutive ones

```python
def findMaxConsecutiveOnes(nums):
    max_count = 0
    current_count = 0

    for num in nums:
        if num == 1:
            current_count += 1
        else:
            max_count = max(max_count, current_count)
            current_count = 0

    # Check the last sequence of 1s
    max_count = max(max_count, current_count)

    return max_count

# Example usage
nums = [1, 1, 0, 1, 1, 1]
print("Input array:", nums)
print("Maximum consecutive ones:", findMaxConsecutiveOnes(nums))
```

N meetings in one room

```python
def max_meetings(start_times, end_times):
    # Create a list of meetings with start and end times
    meetings = list(zip(start_times, end_times))

    # Sort the meetings by their end times
```

```
    meetings.sort(key=lambda x: x[1])

    # Initialize variables
    last_end_time = 0
    selected_meetings = []

    # Iterate through the sorted meetings
    for start, end in meetings:
        if start >= last_end_time:
     #If the meeting starts after or when the last meeting ends, select it
            selected_meetings.append((start, end))
            last_end_time = end

    return selected_meetings

# Example usage:
start_times = [1, 3, 0, 5, 8, 5]
end_times = [2, 4, 6, 7, 9, 9]

selected_meetings = max_meetings(start_times, end_times)
print("The maximum number of non-overlapping meetings is:",
len(selected_meetings))
print("The selected meetings are:", selected_meetings)
```

Minimum number of platforms required for a railway

```
def find_min_platforms(arrivals, departures):
    # Sort the arrival and departure times
    arrivals.sort()
    departures.sort()

    # Initialize pointers for arrival and departure times
    arrival_index = 0
    departure_index = 0
    platform_count = 0
    max_platforms = 0

    # Iterate over arrivals and departures
    while arrival_index < len(arrivals) and departure_index <
len(departures):
        if arrivals[arrival_index] < departures[departure_index]:
            # A train arrives before the previous one departs, need an extra
platform
            platform_count += 1
            arrival_index += 1
            # Update the maximum number of platforms needed
            max_platforms = max(max_platforms, platform_count)
```

```
        else:
            # A train departs, release a platform
            platform_count -= 1
            departure_index += 1

    return max_platforms

# Example usage:
arrivals = [900, 940, 950, 1100, 1500, 1800]
departures = [910, 1200, 1120, 1130, 1900, 2000]

min_platforms_needed = find_min_platforms(arrivals, departures)
print("Minimum number of platforms required:", min_platforms_needed)
```

Minimum number of platforms required for a railway

```
def find_min_platforms(arrivals, departures):
    # Sort the arrival and departure times
    arrivals.sort()
    departures.sort()

    # Initialize pointers for arrival and departure times
    arrival_index = 0
    departure_index = 0
    platform_count = 0
    max_platforms = 0

    # Iterate over arrivals and departures
    while arrival_index < len(arrivals) and departure_index <
len(departures):
        if arrivals[arrival_index] < departures[departure_index]:
   # A train arrives before the previous one departs, need an extra platform
            platform_count += 1
            arrival_index += 1
            # Update the maximum number of platforms needed
            max_platforms = max(max_platforms, platform_count)
        else:
            # A train departs, release a platform
            platform_count -= 1
            departure_index += 1

    return max_platforms

# Example usage:
arrivals = [900, 940, 950, 1100, 1500, 1800]
departures = [910, 1200, 1120, 1130, 1900, 2000]
```

```
min_platforms_needed = find_min_platforms(arrivals, departures)
print("Minimum number of platforms required:", min_platforms_needed)
```

Job sequencing Problem

```
def job_sequencing(jobs):
    # Sort jobs by profit in descending order
    jobs.sort(key=lambda x: x[2], reverse=True)

    # Find maximum deadline
    max_deadline = max(job[1] for job in jobs)

    # Initialize result array and total profit
    result = [-1] * max_deadline
    total_profit = 0

    # Iterate through each job
    for job in jobs:
        # Find the latest available slot before the deadline
        for j in range(job[1] - 1, -1, -1):
            if result[j] == -1:
                result[j] = job[0]  # Assign job ID to this slot
                total_profit += job[2]  # Add profit of this job to total
profit
                break

    # Return the list of job IDs scheduled and total profit
    return [job_id for job_id in result if job_id != -1], total_profit

# Example usage:
jobs = [(1, 4, 20), (2, 1, 10), (3, 1, 40), (4, 1, 30)]
scheduled_jobs, total_profit = job_sequencing(jobs)

print("Scheduled jobs (in order of execution):", scheduled_jobs)
print("Total profit:", total_profit)
```

Fractional Knapsack Problem

```
def fractional_knapsack(items, capacity):
    # Calculate value-to-weight ratios and sort items by ratio in descending
order
    items.sort(key=lambda x: x[1] / x[0], reverse=True)

    total_value = 0.0
    knapsack = [0.0] * len(items)  # To store the fraction of each item
taken
```

```python
    for i, (value, weight) in enumerate(items):
        if capacity >= weight:
            # Take the whole item
            knapsack[i] = 1.0
            total_value += value
            capacity -= weight
        else:
            # Take a fraction of the item
            fraction = capacity / weight
            knapsack[i] = fraction
            total_value += value * fraction
            capacity = 0
            break  # Knapsack is full

    return knapsack, total_value

# Example usage:
items = [(60, 10), (100, 20), (120, 30)]
knapsack_capacity = 50

selected_items, max_value = fractional_knapsack(items, knapsack_capacity)

print("Selected fractions of each item:", selected_items)
print("Maximum value in the knapsack:", max_value)
```

Greedy algorithm to find minimum number of coins

```python
def min_coins(amount, denominations):
    # Sort denominations in descending order
    denominations.sort(reverse=True)

    # Initialize variables
    num_coins = 0
    coin_count = {}

    # Iterate through each denomination
    for coin in denominations:
        if amount <= 0:
            break

        # Calculate how many coins of this denomination can be used
        count = amount // coin
        if count > 0:
            coin_count[coin] = count
            num_coins += count
            amount -= count * coin
```

```
    return num_coins, coin_count

# Example usage:
amount = 87
denominations = [1, 5, 10, 25]

min_coins_needed, coin_count = min_coins(amount, denominations)

print(f"Minimum number of coins needed for amount {amount}:
{min_coins_needed}")
print("Coins used:")
for coin, count in coin_count.items():
    print(f"{count} coin(s) of denomination {coin}")
```

Activity Selection (it is the same as N meeting in one room)

```
def max_activities(activities):
    # Sort activities by their finish times
    activities.sort(key=lambda x: x[1])

    # Initialize variables
    selected_activities = []
    last_end_time = 0

    # Iterate through sorted activities
    for activity in activities:
        start, end = activity
        if start >= last_end_time:
            selected_activities.append(activity)
            last_end_time = end

    return selected_activities

# Example usage:
activities = [(1, 2), (3, 4), (0, 6), (5, 7), (8, 9), (5, 9)]
selected_activities = max_activities(activities)

print("Maximum number of non-overlapping activities:",
len(selected_activities))
print("Selected activities:", selected_activities)
```

Subset Sums

```
def subset_sum_recursive(nums, target):
    # Base cases
```

```python
    if target == 0:
        return True
    if not nums and target != 0:
        return False

    # Recursive cases
    # Include the last element and check the remaining subset
    include_last = subset_sum_recursive(nums[:-1], target - nums[-1])
    # Exclude the last element and check the remaining subset
    exclude_last = subset_sum_recursive(nums[:-1], target)

    # Return True if either of the above cases is True
    return include_last or exclude_last

# Example usage:
nums = [3, 34, 4, 12, 5, 2]
target = 9
print(subset_sum_recursive(nums, target))  # Output: True (because 4 + 5 =
9)
```

Subset-II

```python
def subsets_with_duplicates(nums):
    # Sort the input to handle duplicates
    nums.sort()
    results = []

    def backtrack(start, subset):
        results.append(subset[:])
        for i in range(start, len(nums)):
            # Skip duplicates to ensure uniqueness in subsets
            if i > start and nums[i] == nums[i-1]:
                continue
            subset.append(nums[i])
            backtrack(i + 1, subset)
            subset.pop()

    backtrack(0, [])
    return results

# Example usage:
nums = [1, 2, 2]
print(subsets_with_duplicates(nums))
```

Combination sum-1

```python
def combination_sum(nums, target):
    results = []

    def backtrack(start, target, path):
        if target == 0:
            results.append(path)
            return
        if target < 0:
            return

        for i in range(start, len(nums)):
            # Include the current element in the path
            backtrack(i, target - nums[i], path + [nums[i]])

    backtrack(0, target, [])
    return results

# Example usage:
nums = [2, 3, 6, 7]
target = 7
print(combination_sum(nums, target))  # Output: [[2, 2, 3], [7]]
```

Combination sum-2

```python
def combination_sum2(nums, target):
    nums.sort()  # Sort the input to handle duplicates
    results = []

    def backtrack(start, target, path):
        if target == 0:
            results.append(path)
            return
        if target < 0:
            return

        for i in range(start, len(nums)):
            # Skip duplicates
            if i > start and nums[i] == nums[i-1]:
                continue
            # Include the current element in the path
            backtrack(i + 1, target - nums[i], path + [nums[i]])

    backtrack(0, target, [])
    return results

# Example usage:
nums = [10, 1, 2, 7, 6, 1, 5]
```

```
  target = 8
  print(combination_sum2(nums, target))  # Output: [[1, 1, 6], [1, 2, 5], [1,
7], [2, 6]]
```

Palindrome Partitioning

```
def partition(s):
    def is_palindrome(s):
        return s == s[::-1]

    def backtrack(start, path):
        if start == len(s):
            results.append(path[:])
            return

        for i in range(start, len(s)):
            substring = s[start:i+1]
            if is_palindrome(substring):
                path.append(substring)
                backtrack(i + 1, path)
                path.pop()

    results = []
    backtrack(0, [])
    return results

# Example usage:
s = "aab"
print(partition(s))  # Output: [['a', 'a', 'b'], ['aa', 'b']]
```

K-th permutation Sequence

```
import math

def getPermutation(n, k):
    nums = [str(i) for i in range(1, n + 1)]
    factorial = [1] * (n + 1)
    for i in range(2, n + 1):
        factorial[i] = factorial[i - 1] * i

    result = []
    k -= 1  # Convert k to 0-based index

    for i in range(n, 0, -1):
        index = k // factorial[i - 1]
        k %= factorial[i - 1]
```

```python
        result.append(nums[index])
        nums.pop(index)

    return ''.join(result)

# Example usage:
n = 4
k = 9
print(getPermutation(n, k))  # Output: "2314"
```

Print all permutations of a string/array

```python
def permutations(s):
    result = []
    backtrack(list(s), 0, result)
    return result

def backtrack(s, start, result):
    if start == len(s):
        result.append("".join(s))
    else:
        for i in range(start, len(s)):
            # Swap characters
            s[start], s[i] = s[i], s[start]
            # Recursively permute the rest of the string
            backtrack(s, start + 1, result)
            # Backtrack: restore the original order
            s[start], s[i] = s[i], s[start]

# Example usage:
input_string = "abc"
print("Permutations of", input_string, "are:", permutations(input_string))
```

N queens Problem

```python
def solve_n_queens(n):
    result = []
    board = [['.'] * n for _ in range(n)]
    backtrack(board, 0, result)
    return result

def backtrack(board, row, result):
    n = len(board)
    if row == n:
        result.append(["".join(row) for row in board])
        return
```

```python
    for col in range(n):
        if is_safe(board, row, col):
            board[row][col] = 'Q'
            backtrack(board, row + 1, result)
            board[row][col] = '.'

def is_safe(board, row, col):
    n = len(board)
    # Check if there's a queen in the same column
    for i in range(row):
        if board[i][col] == 'Q':
            return False

    # Check upper left diagonal
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j] == 'Q':
            return False

    # Check upper right diagonal
    for i, j in zip(range(row-1, -1, -1), range(col+1, n)):
        if board[i][j] == 'Q':
            return False

    return True

# Example usage:
n = 4
solutions = solve_n_queens(n)
print(f"Number of solutions for {n}-Queens problem:", len(solutions))
for idx, solution in enumerate(solutions):
    print(f"Solution {idx + 1}:")
    for row in solution:
        print(row)
    print()
```

Sudoko Solver

```python
def solve_sudoku(board):
    if not board:
        return False
    return solve(board)

def solve(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
```

```python
                if board[i][j] == '.':
                    for num in '123456789':
                        if is_valid(board, i, j, num):
                            board[i][j] = num
                            if solve(board):
                                return True
                            board[i][j] = '.'
                    return False
        return True

def is_valid(board, row, col, num):
    # Check row
    for j in range(9):
        if board[row][j] == num:
            return False

    # Check column
    for i in range(9):
        if board[i][col] == num:
            return False

    # Check 3x3 box
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False

    return True

# Example usage:
board = [
    ['5', '3', '.', '.', '7', '.', '.', '.', '.'],
    ['6', '.', '.', '1', '9', '5', '.', '.', '.'],
    ['.', '9', '8', '.', '.', '.', '.', '6', '.'],
    ['8', '.', '.', '.', '6', '.', '.', '.', '3'],
    ['4', '.', '.', '8', '.', '3', '.', '.', '1'],
    ['7', '.', '.', '.', '2', '.', '.', '.', '6'],
    ['.', '6', '.', '.', '.', '.', '2', '8', '.'],
    ['.', '.', '.', '4', '1', '9', '.', '.', '5'],
    ['.', '.', '.', '.', '8', '.', '.', '7', '9']
]

print("Sudoku board before solving:")
for row in board:
    print(row)

solve_sudoku(board)
```

```python
print("\nSudoku board after solving:")
for row in board:
    print(row)
```

M Coloring Problem

```python
def graph_coloring(graph, m):
    n = len(graph)
    colors = [-1] * n  # Initialize colors for all vertices as -1

    if not graph_coloring_util(graph, m, colors, 0):
        print(f"No solution exists with {m} colors.")
        return False

    print(f"Solution exists with {m} colors. Vertex colors are:")
    for i in range(n):
        print(f"Vertex {i}: Color {colors[i]}")

    return True

def graph_coloring_util(graph, m, colors, vertex):
    n = len(graph)
    if vertex == n:
        return True

    for color in range(m):
        if is_safe(graph, colors, vertex, color):
            colors[vertex] = color
            if graph_coloring_util(graph, m, colors, vertex + 1):
                return True
            colors[vertex] = -1

    return False

def is_safe(graph, colors, vertex, color):
    for i in range(len(graph)):
        if graph[vertex][i] == 1 and colors[i] == color:
            return False
    return True

# Example usage:
graph = [
    [0, 1, 1, 1],
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 0]
```

```
]
m = 3

graph_coloring(graph, m)
```

Rat in a Maze

```
def solve_maze(maze):
    if not maze:
        return []

    n = len(maze)
    if n == 0:
        return []

    solution = [[0] * n for _ in range(n)]  # Initialize solution matrix
    if solve_maze_util(maze, 0, 0, solution, n):
        print("Solution exists. Path taken by the rat:")
        print_solution(solution)
    else:
        print("No solution exists.")

def solve_maze_util(maze, x, y, solution, n):
    # Base case: reached the bottom-right corner
    if x == n - 1 and y == n - 1:
        solution[x][y] = 1
        return True

    # Check if x, y is valid
    if is_safe(maze, x, y, n):
        # Mark x, y as part of the solution path
        solution[x][y] = 1

        # Move right
        if solve_maze_util(maze, x + 1, y, solution, n):
            return True

        # Move down
        if solve_maze_util(maze, x, y + 1, solution, n):
            return True

        # If no move leads to a solution, backtrack
        solution[x][y] = 0
        return False

    return False
```

```python
def is_safe(maze, x, y, n):
    # Check if x, y is within maze bounds and is not blocked
    if 0 <= x < n and 0 <= y < n and maze[x][y] == 1:
        return True
    return False

def print_solution(solution):
    n = len(solution)
    for i in range(n):
        for j in range(n):
            print(solution[i][j], end=" ")
        print()

# Example usage:
maze = [
    [1, 0, 0, 0],
    [1, 1, 0, 1],
    [0, 1, 0, 0],
    [1, 1, 1, 1]
]

solve_maze(maze)
```

Word Break (print all ways)

```python
def word_break(s, word_dict):
    memo = {}
    return word_break_util(s, word_dict, memo)

def word_break_util(s, word_dict, memo):
    if s in memo:
        return memo[s]

    result = []
    for word in word_dict:
        if s.startswith(word):
            if len(word) == len(s):
                result.append(word)
            else:
                rest_of_string = s[len(word):]
                sub_breaks = word_break_util(rest_of_string, word_dict,
memo)
                for sub_break in sub_breaks:
                    result.append(word + " " + sub_break)

    memo[s] = result
    return result
```

```
# Example usage:
string = "catsanddog"
dictionary = ["cat", "cats", "and", "sand", "dog"]

ways_to_break = word_break(string, dictionary)
if ways_to_break:
    print(f"Possible ways to break '{string}':")
    for way in ways_to_break:
        print(way)
else:
    print(f"No possible ways to break '{string}' with the given 
dictionary.")
```

The N-th root of an integer

```
def nth_root(target, N):
    if target == 0:
        return 0  # Special case: 0-th root of 0 is 0
    if N == 1:
        return target  # The 1st root of any number is the number itself

    left, right = 0, target
    while left <= right:
        mid = (left + right) // 2
        mid_pow = mid ** N
        if mid_pow == target:
            return mid
        elif mid_pow < target:
            left = mid + 1
        else:
            right = mid - 1
    return right  # Return the largest integer x such that x^N <= target

# Example usage:
target = 27
N = 3
result = nth_root(target, N)
print(f"The {N}-th root of {target} is {result}")
```

Matrix Median

```
def count_less_equal(matrix, mid):
    count = 0
    rows, cols = len(matrix), len(matrix[0])
    for row in matrix:
```

```python
        # Count how many elements in the current row are <= mid
        count += sum(1 for num in row if num <= mid)
    return count

def find_median(matrix):
    rows, cols = len(matrix), len(matrix[0])
    min_element = float('inf')
    max_element = float('-inf')

    # Finding the min and max elements in the matrix
    for row in matrix:
        min_element = min(min_element, row[0])
        max_element = max(max_element, row[-1])

    desired_position = (rows * cols + 1) // 2

    while min_element < max_element:
        mid = min_element + (max_element - min_element) // 2
        count = count_less_equal(matrix, mid)

        if count < desired_position:
            min_element = mid + 1
        else:
            max_element = mid

    return min_element

# Example usage:
matrix = [
    [1, 3, 5],
    [2, 6, 9],
    [3, 6, 9]
]

median = find_median(matrix)
print(f"The median of the matrix is: {median}")
```

Find the element that appears once in a sorted array, and the rest element appears twice (Binary search)

```python
def find_single_element(nums):
    left, right = 0, len(nums) - 1

    while left <= right:
        if left == right:
            return nums[left]
```

```python
        mid = (left + right) // 2

        if mid % 2 == 0:  # mid is even
            if nums[mid] == nums[mid + 1]:
                left = mid + 2
            else:
                right = mid
        else:  # mid is odd
            if nums[mid] == nums[mid - 1]:
                left = mid + 1
            else:
                right = mid - 1

    return -1  # Not found (though not expected in the problem description)

# Example usage:
nums = [1, 1, 2, 2, 3, 3, 4, 4, 5]  # Example array with one unique element
unique_element = find_single_element(nums)
print(f"The unique element in the array is: {unique_element}")
```

Search element in a sorted and rotated array/ find pivot where it is rotated

```python
def find_pivot(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[right]:  # Pivot is in the right half
            left = mid + 1
        else:  # Pivot is in the left half
            right = mid
    return left  # Pivot index

# Example usage:
nums = [4, 5, 6, 7, 0, 1, 2]
pivot = find_pivot(nums)
print(f"The pivot index is: {pivot}")
```

Median of 2 sorted arrays

```python
def find_median_sorted_arrays(nums1, nums2):
    # Ensure nums1 is the smaller array
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    low, high = 0, x
```

```
    while low <= high:
        partitionX = (low + high) // 2
        partitionY = (x + y + 1) // 2 - partitionX

        maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        minX = float('inf') if partitionX == x else nums1[partitionX]

        maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
        minY = float('inf') if partitionY == y else nums2[partitionY]

        if maxX <= minY and maxY <= minX:
            # We have partitioned correctly
            if (x + y) % 2 == 0:
                return (max(maxX, maxY) + min(minX, minY)) / 2
            else:
                return max(maxX, maxY)
        elif maxX > minY:  # Move towards left in nums1
            high = partitionX - 1
        else:  # Move towards right in nums1
            low = partitionX + 1

    raise ValueError("Input arrays are not sorted.")

# Example usage:
nums1 = [1, 3]
nums2 = [2]
median = find_median_sorted_arrays(nums1, nums2)
print(f"The median of the two sorted arrays is: {median}")
```

K-th element of two sorted arrays

```
def find_kth_element(nums1, nums2, k):
    # Ensure nums1 is the smaller array
    if len(nums1) > len(nums2):
        return find_kth_element(nums2, nums1, k)

    # Base cases
    if not nums1:
        return nums2[k - 1]
    if k == 1:
        return min(nums1[0], nums2[0])

    # Partition sizes
    idx1 = min(len(nums1), k // 2)
    idx2 = k - idx1
```

```
    # Compare and eliminate
    if nums1[idx1 - 1] < nums2[idx2 - 1]:
        return find_kth_element(nums1[idx1:], nums2, k - idx1)
    else:
        return find_kth_element(nums1, nums2[idx2:], k - idx2)

# Example usage:
nums1 = [1, 3, 7, 10, 12]
nums2 = [2, 4, 6, 8, 9]
k = 5
kth_element = find_kth_element(nums1, nums2, k)
print(f"The {k}-th smallest element in the combined sorted arrays is:
{kth_element}")
```

Allocate Minimum Number of Pages

```
def is_valid(arr, n, m, max_pages):
    student_count = 1
    current_sum = 0

    for pages in arr:
        if current_sum + pages > max_pages:
            student_count += 1
            current_sum = pages
            if student_count > m:
                return False
        else:
            current_sum += pages

    return True

def find_min_pages(arr, n, m):
    if m > n:
        return -1

    low, high = max(arr), sum(arr)
    result = high

    while low <= high:
        mid = (low + high) // 2
        if is_valid(arr, n, m, mid):
            result = mid
            high = mid - 1
        else:
            low = mid + 1

    return result
```

```
# Example usage:
arr = [12, 34, 67, 90]
n = len(arr)
m = 2
min_pages = find_min_pages(arr, n, m)
print(f"The minimum number of pages to be allocated so that no student reads
more than this is: {min_pages}")
```

Aggressive Cows

```
def can_place_cows(stalls, n, k, min_dist):
    count = 1  # Place the first cow in the first stall
    last_position = stalls[0]

    for i in range(1, n):
        if stalls[i] - last_position >= min_dist:
            count += 1
            last_position = stalls[i]
            if count == k:
                return True
    return False

def find_max_min_distance(stalls, n, k):
    stalls.sort()

    low = 1  # Minimum possible distance
    high = stalls[-1] - stalls[0]  # Maximum possible distance
    result = 0

    while low <= high:
        mid = (low + high) // 2
        if can_place_cows(stalls, n, k, mid):
            result = mid
            low = mid + 1
        else:
            high = mid - 1
    return result

# Example usage:
stalls = [1, 2, 8, 4, 9]
n = len(stalls)
k = 3
max_min_distance = find_max_min_distance(stalls, n, k)
print(f"The largest minimum distance between any two cows is:
{max_min_distance}")
```

Max heap, Min Heap Implementation (Only for interviews)

Min-Heap

```
  class MinHeap:
      def __init__(self):
          self.heap = []

      def parent(self, i):
          return (i - 1) // 2

      def left_child(self, i):
          return 2 * i + 1

      def right_child(self, i):
          return 2 * i + 2

      def insert(self, key):
          self.heap.append(key)
          self.heapify_up(len(self.heap) - 1)

      def heapify_up(self, i):
          while i != 0 and self.heap[self.parent(i)] > self.heap[i]:
            self.heap[i], self.heap[self.parent(i)] =
self.heap[self.parent(i)],   self.heap[i]
              i = self.parent(i)

      def extract_min(self):
          if not self.heap:
              return None
          root = self.heap[0]
          self.heap[0] = self.heap.pop()
          self.heapify_down(0)
          return root

      def heapify_down(self, i):
          smallest = i
          left = self.left_child(i)
          right = self.right_child(i)

          if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
              smallest = left
          if right < len(self.heap) and self.heap[right] <
self.heap[smallest]:
              smallest = right
          if smallest != i:
              self.heap[i],
```

Max-Heap

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, key):
        self.heap.append(key)
        self.heapify_up(len(self.heap) - 1)

    def heapify_up(self, i):
        while i != 0 and self.heap[self.parent(i)] < self.heap[i]:
            self.heap[i], self.heap[self.parent(i)] =
self.heap[self.parent(i)], self.heap[i]
            i = self.parent(i)

    def extract_max(self):
        if not self.heap:
            return None
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return root

    def heapify_down(self, i):
        largest = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.heap[left] > self.heap[largest]:
            largest = left
        if right < len(self.heap) and self.heap[right] > self.heap[largest]:
            largest = right
        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest],
self.heap[i]
            self.heapify_down(largest)

    def get_max(self):
        return self.heap[0] if self.heap else None
```

```python
# Min-Heap example
min_heap = MinHeap()
min_heap.insert(3)
min_heap.insert(1)
min_heap.insert(6)
min_heap.insert(5)
min_heap.insert(2)
min_heap.insert(4)

print("Min-Heap:", min_heap.heap)
print("Extracted Min:", min_heap.extract_min())
print("Min-Heap after extraction:", min_heap.heap)

# Max-Heap example
max_heap = MaxHeap()
max_heap.insert(3)
max_heap.insert(1)
max_heap.insert(6)
max_heap.insert(5)
max_heap.insert(2)
max_heap.insert(4)

print("Max-Heap:", max_heap.heap)
print("Extracted Max:", max_heap.extract_max())
print("Max-Heap after extraction:", max_heap.heap)
```

Kth Largest Element

```python
import heapq

class KthLargest:
    def __init__(self, k, nums):
        self.min_heap = nums
        self.k = k
        heapq.heapify(self.min_heap)
        while len(self.min_heap) > k:
            heapq.heappop(self.min_heap)

    def add(self, val):
        if len(self.min_heap) < self.k:
            heapq.heappush(self.min_heap, val)
        elif val > self.min_heap[0]:
            heapq.heapreplace(self.min_heap, val)
        return self.min_heap[0]

k = 3
nums = [4, 5, 8, 2]
```

```
kth_largest = KthLargest(k, nums)
print(kth_largest.add(3))  # returns 4
print(kth_largest.add(5))  # returns 5
print(kth_largest.add(10)) # returns 5
print(kth_largest.add(9))  # returns 8
print(kth_largest.add(4))  # returns 8
```

Maximum Sum Combination

```
import heapq

def k_max_sum_combinations(A, B, K):
    A.sort(reverse=True)
    B.sort(reverse=True)

    N = len(A)

    max_heap = []
    visited = set()

    heapq.heappush(max_heap, (-(A[0] + B[0]), 0, 0))
    visited.add((0, 0))

    result = []

    while K > 0 and max_heap:
        current_sum, i, j = heapq.heappop(max_heap)
        result.append(-current_sum)
        K -= 1

        if i + 1 < N and (i + 1, j) not in visited:
            heapq.heappush(max_heap, (-(A[i + 1] + B[j]), i + 1, j))
            visited.add((i + 1, j))

        if j + 1 < N and (i, j + 1) not in visited:
            heapq.heappush(max_heap, (-(A[i] + B[j + 1]), i, j + 1))
            visited.add((i, j + 1))

    return result

# Example usage
A = [1, 4, 2, 3]
B = [2, 5, 1, 6]
K = 4
print(k_max_sum_combinations(A, B, K))  # Output: [10, 9, 9, 8]
```

Find Median from Data Stream

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.max_heap = []  # max-heap for the left half
        self.min_heap = []  # min-heap for the right half

    def addNum(self, num):
    # Add to max-heap (use negative values to simulate max-heap using heapq)
        heapq.heappush(self.max_heap, -num)

    # Balance the heaps: move the largest element from max-heap to min-heap
        heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))

   # Ensure the max-heap can have at most one more element than the min-heap
        if len(self.min_heap) > len(self.max_heap):
            heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))

    def findMedian(self):
        if len(self.max_heap) > len(self.min_heap):
            return -self.max_heap[0]
        else:
            return (-self.max_heap[0] + self.min_heap[0]) / 2.0

# Example usage
median_finder = MedianFinder()
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    median_finder.addNum(num)
    print(f"Added {num}, current median: {median_finder.findMedian()}")
```

Merge K sorted arrays

```python
import heapq

def merge_k_sorted_arrays(arrays):
    min_heap = []
    result = []

    # Initialize the heap with the first element of each array
    for i, array in enumerate(arrays):
        if array:
            heapq.heappush(min_heap, (array[0], i, 0))

    while min_heap:
```

```
        val, list_idx, element_idx = heapq.heappop(min_heap)
        result.append(val)

        # If there is another element in the same array, add it to the heap
        if element_idx + 1 < len(arrays[list_idx]):
            next_val = arrays[list_idx][element_idx + 1]
            heapq.heappush(min_heap, (next_val, list_idx, element_idx + 1))

    return result

# Example usage
arrays = [
    [1, 4, 5],
    [1, 3, 4],
    [2, 6]
]

print(merge_k_sorted_arrays(arrays))  # Output: [1, 1, 2, 3, 4, 4, 5, 6]
```

K most frequent elements

```
import heapq
from collections import defaultdict

def topKFrequent(nums, k):
    freq_map = defaultdict(int)

    # Count frequencies
    for num in nums:
        freq_map[num] += 1

    min_heap = []

    # Push elements into min-heap
    for num, freq in freq_map.items():
        heapq.heappush(min_heap, (freq, num))
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    # Extract top K frequent elements from min-heap
    result = []
    while min_heap:
        result.append(heapq.heappop(min_heap)[1])

    return result[::-1]

# Example usage
```

```
nums = [1, 1, 1, 2, 2, 3]
k = 2
print(topKFrequent(nums, k))  # Output: [1, 2]
```

Implement Stack using Arrays

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack size:", stack.size())
print("Peek:", stack.peek())
print("Pop:", stack.pop())
print("Peek after pop:", stack.peek())
print("Stack size after pop:", stack.size())
```

Implement Queue using Arrays

```
class Queue:
```

```python
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = -1
        self.size = 0

    def enqueue(self, item):
        if self.is_full():
            raise IndexError("Queue is full")
        self.rear = (self.rear + 1) % self.capacity
        self.queue[self.rear] = item
        self.size += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        item = self.queue[self.front]
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return item

    def front(self):
        if self.is_empty():
            return None
        return self.queue[self.front]

    def is_empty(self):
        return self.size == 0

    def is_full(self):
        return self.size == self.capacity

    def queue_size(self):
        return self.size

# Example usage:
queue = Queue(5)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Queue size:", queue.queue_size())
print("Front:", queue.front())
print("Dequeue:", queue.dequeue())
print("Front after dequeue:", queue.front())
print("Queue size after dequeue:", queue.queue_size())
```

Implement Stack using Queue (using single queue)

```python
from collections import deque

class Stack:
    def __init__(self):
        self.queue = deque()

    def push(self, item):
        # Add the new element to the queue
        self.queue.append(item)
        # Rotate the queue so that the newly added item is at the front
        for _ in range(len(self.queue) - 1):
            self.queue.append(self.queue.popleft())

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.queue.popleft()

    def peek(self):
        if self.is_empty():
            return None
        return self.queue[0]

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack size:", stack.size())
print("Peek:", stack.peek())
print("Pop:", stack.pop())
print("Peek after pop:", stack.peek())
print("Stack size after pop:", stack.size())
```

Implement Queue using Stack (0(1) amortized method)

```python
class Queue:
    def __init__(self):
        self.stack1 = []
```

```python
        self.stack2 = []

    def enqueue(self, item):
        self.stack1.append(item)

    def dequeue(self):
        if not self.stack2:
            if not self.stack1:
                raise IndexError("Queue is empty")
            while self.stack1:
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop()

    def peek(self):
        if not self.stack2:
            if not self.stack1:
                raise IndexError("Queue is empty")
            while self.stack1:
                self.stack2.append(self.stack1.pop())
        return self.stack2[-1]

    def is_empty(self):
        return not self.stack1 and not self.stack2

    def queue_size(self):
        return len(self.stack1) + len(self.stack2)

# Example usage:
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Queue size:", queue.queue_size())
print("Peek:", queue.peek())
print("Dequeue:", queue.dequeue())
print("Peek after dequeue:", queue.peek())
print("Queue size after dequeue:", queue.queue_size())
```

Check for balanced parentheses

```python
def balanced_parentheses(expression):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in expression:
        if char in mapping.values():
```

```python
            stack.append(char)
        elif char in mapping.keys():
            if stack and stack[-1] == mapping[char]:
                stack.pop()
            else:
                return False
    return len(stack) == 0

# Example usage:
print(balanced_parentheses("((({})))"))  # True
print(balanced_parentheses("({[]}[])"))  # True
print(balanced_parentheses("(()"))       # False
print(balanced_parentheses("({)}"))      # False
```

Next Greater Element

```python
def next_greater_element(nums):
    stack = []
    result = [-1] * len(nums)  # Initialize result array with -1

    for i in range(len(nums)):
        while stack and nums[stack[-1]] < nums[i]:
            result[stack.pop()] = nums[i]
        stack.append(i)

    return result

# Example usage:
nums = [4, 2, 6, 8, 1, 5]
print("Original array:", nums)
print("Next Greater Elements:", next_greater_element(nums))
```

Sort a Stack

```python
def sort_stack(stack):
    sorted_stack = []

    while stack:
        temp = stack.pop()

        while sorted_stack and sorted_stack[-1] > temp:
            stack.append(sorted_stack.pop())

        sorted_stack.append(temp)

    while sorted_stack:
```

```
        stack.append(sorted_stack.pop())

    return stack

# Example usage:
stack = [3, 1, 4, 2, 5]
print("Original stack:", stack)
sorted_stack = sort_stack(stack[:]) #Make a copy to preserve original stack
print("Sorted stack:", sorted_stack)
```

Next Smaller Element

```
def next_smaller_element(nums):
    stack = []
    result = [-1] * len(nums)  # Initialize result array with -1

    for i in range(len(nums)):
        while stack and nums[stack[-1]] > nums[i]:
            result[stack.pop()] = nums[i]
        stack.append(i)

    return result

# Example usage:
nums = [4, 2, 6, 8, 1, 5]
print("Original array:", nums)
print("Next Smaller Elements:", next_smaller_element(nums))
```

LRU cache (IMPORTANT)

```
from collections import deque

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = []  # To store most recently used items (stack)
        self.queue = deque()  # To store least recently used items (queue)

    def get(self, key):
        if key in self.stack:
            self.stack.remove(key)
            self.stack.append(key)
            return key
        return -1

    def put(self, key):
```

```
        if key in self.stack:
            self.stack.remove(key)
        elif len(self.stack) == self.capacity:
            evicted = self.queue.popleft()
            self.stack.remove(evicted)
        self.stack.append(key)
        self.queue.append(key)

# Example usage:
cache = LRUCache(3)
cache.put(1)
cache.put(2)
cache.put(3)
print(cache.stack)  # [1, 2, 3]
cache.put(4)
print(cache.stack)  # [2, 3, 4]
cache.get(2)
print(cache.stack)  # [3, 4, 2]
```

LFU cache

```
from collections import defaultdict, OrderedDict

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.key_val = {}  # Stores key-value pairs
        self.key_freq = defaultdict(int)  # Stores frequency of each key
        self.freq_keys = defaultdict(OrderedDict)  # Stores keys by
frequency
        self.min_freq = 0  # Tracks minimum frequency in cache

    def get(self, key):
        if key not in self.key_val:
            return -1

        # Update frequency
        value = self.key_val[key]
        current_freq = self.key_freq[key]
        del self.freq_keys[current_freq][key]

        if not self.freq_keys[current_freq] and current_freq ==
self.min_freq:
            self.min_freq += 1

        self.key_freq[key] += 1
        self.freq_keys[current_freq + 1][key] = True
```

```
        return value

    def put(self, key, value):
        if self.capacity <= 0:
            return

        if key in self.key_val:
            self.key_val[key] = value
            self.get(key)
            return

        if len(self.key_val) >= self.capacity:
            del_key, _ = self.freq_keys[self.min_freq].popitem(last=False)
            del self.key_val[del_key]
            del self.key_freq[del_key]

        self.key_val[key] = value
        self.key_freq[key] = 1
        self.freq_keys[1][key] = True
        self.min_freq = 1

# Example usage:
cache = LFUCache(2)
cache.put(1, 1)
cache.put(2, 2)
print(cache.get(1))  # Output: 1
cache.put(3, 3)
print(cache.get(2))  # Output: -1 (not found)
print(cache.get(3))  # Output: 3
cache.put(4, 4)
print(cache.get(1))  # Output: -1 (not found)
print(cache.get(3))  # Output: 3
print(cache.get(4))  # Output: 4
```

Largest rectangle in a histogram

```
def largest_rectangle_area(heights):
    stack = []
    max_area = 0
    index = 0

    while index < len(heights):
        # If this bar is higher than the bar at stack top, push it to the
stack
        if not stack or heights[index] >= heights[stack[-1]]:
            stack.append(index)
```

```
            index += 1
        else:
            # Pop the top
            top_of_stack = stack.pop()
           # Calculate the area with heights[top_of_stack] as the smallest
(or   minimum height) bar 'h'
            area = (heights[top_of_stack] *
                   ((index - stack[-1] - 1) if stack else index))
            # Update max_area, if needed
            max_area = max(max_area, area)

    # Now pop the remaining bars from stack and calculate area with each
popped bar
    while stack:
        top_of_stack = stack.pop()
        area = (heights[top_of_stack] *
               ((index - stack[-1] - 1) if stack else index))
        max_area = max(max_area, area)

    return max_area

# Example usage:
histogram = [6, 2, 5, 4, 5, 1, 6]
print("Histogram heights:", histogram)
print("Largest rectangle area:", largest_rectangle_area(histogram))
```

Sliding Window maximum

```
from collections import deque

def max_sliding_window(nums, k):
    if not nums:
        return []

    result = []
    deque_window = deque()

    for i in range(len(nums)):
        # Remove elements not within the window size
        if deque_window and deque_window[0] <= i - k:
            deque_window.popleft()

        # Maintain decreasing order in deque
        while deque_window and nums[deque_window[-1]] <= nums[i]:
            deque_window.pop()

        deque_window.append(i)
```

```
        # Start adding maximum values to result after first k elements
        if i >= k - 1:
            result.append(nums[deque_window[0]])

    return result

# Example usage:
nums = [1, 3, -1, -3, 5, 3, 6,
```

Implement Min Stack

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []  # Auxiliary stack to keep track of minimum
values

    def push(self, x):
        self.stack.append(x)
        # Update min_stack with the new minimum value
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)

    def pop(self):
        if self.stack:
            popped = self.stack.pop()
            if popped == self.min_stack[-1]:
                self.min_stack.pop()

    def top(self):
        if self.stack:
            return self.stack[-1]

    def getMin(self):
        if self.min_stack:
            return self.min_stack[-1]

    def is_empty(self):
        return len(self.stack) == 0

# Example usage:
min_stack = MinStack()
min_stack.push(-2)
min_stack.push(0)
min_stack.push(-3)
```

```python
print("Current stack:", min_stack.stack)
print("Minimum element:", min_stack.getMin())  # Output: -3
min_stack.pop()
print("Top element after pop:", min_stack.top())  # Output: 0
print("Minimum element:", min_stack.getMin())  # Output: -2
```

Rotten Orange (Using BFS)

```python
from collections import deque

def oranges_rotting(grid):
    if not grid:
        return -1

    rows, cols = len(grid), len(grid[0])
    fresh_count = 0
    queue = deque()
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Initialize queue with all initial rotten oranges and count fresh
oranges
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                queue.append((r, c, 0))  # (row, col, minutes)
            elif grid[r][c] == 1:
                fresh_count += 1

    if fresh_count == 0:
        return 0  # No fresh oranges to rot

    minutes = 0

    while queue:
        r, c, minutes = queue.popleft()

        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                grid[nr][nc] = 2
                fresh_count -= 1
                queue.append((nr, nc, minutes + 1))

    return minutes if fresh_count == 0 else -1

# Example usage:
grid = [
```

```
    [2, 1, 1],
    [1, 1, 0],
    [0, 1, 1]
]
print("Minutes until all oranges rot:", oranges_rotting(grid))
```

Stock span problem

```
def calculate_span(prices):
    n = len(prices)
    span = [0] * n
    stack = []

    # Traverse through all days
    for i in range(n):
    # Pop elements from stack while stack is not empty and price[i] >=
prices [stack[-1]]
        while stack and prices[i] >= prices[stack[-1]]:
            stack.pop()

        # Calculate span for current day
        span[i] = i - stack[-1] if stack else i + 1

        # Push current day index to stack
        stack.append(i)
    return span

# Example usage:
prices = [100, 80, 60, 70, 60, 75, 85]
print("Stock prices:", prices)
print("Stock spans:", calculate_span(prices))
```

Find the maximum of minimums of every window size

```
def find_max_of_min_in_windows(arr):
    n = len(arr)

  # Step 1: Calculate the Next Smaller Element (NSE) to the left (left) and
right   (right) for each element
    left = [-1] * n
    right = [n] * n
    stack = []

    # Calculate NSE to the left (left array)
    for i in range(n):
        while stack and arr[stack[-1]] >= arr[i]:
```

```
            stack.pop()
        left[i] = stack[-1] if stack else -1
        stack.append(i)

    stack = []
    # Calculate NSE to the right (right array)
    for i in range(n-1, -1, -1):
        while stack and arr[stack[-1]] >= arr[i]:
            stack.pop()
        right[i] = stack[-1] if stack else n
        stack.append(i)

    # Step 2: Calculate the maximum of minimums for every window size
    result = [0] * (n + 1)
    for i in range(n):
        window_size = right[i] - left[i] - 1
        result[window_size] = max(result[window_size], arr[i])

    # Step 3: Fill in any gaps in result array
    for i in range(n-1, 0, -1):
        result[i] = max(result[i], result[i+1])

    return result[1:]

# Example usage:
arr = [10, 20, 30, 50, 10, 70, 30]
print("Input array:", arr)
print("Maximum of minimums of every window size:",
find_max_of_min_in_windows(arr))
```

The Celebrity Problem

```
def find_celebrity(n, M):
    left, right = 0, n - 1

    # Step 1: Reduce the number of potential candidates using two-pointer
technique
    while left < right:
        if M[left][right] == 1:
            left += 1
        else:
            right -= 1

    candidate = left

    # Step 2: Verify if candidate is the celebrity
    for i in range(n):
```

```python
        if i != candidate:
            if M[candidate][i] == 1 or M[i][candidate] == 0:
                return -1  # No celebrity found

    return candidate

# Example usage:
n = 4
M = [
    [0, 0, 1, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 0],
    [0, 0, 1, 0]
]

celebrity = find_celebrity(n, M)
if celebrity != -1:
    print(f"Celebrity is person {celebrity}")
else:
    print("No celebrity found")
```

Reverse Words in a String

```python
def reverse_words(s):
    # Split the string into words
    words = s.split()

    # Reverse the list of words
    reversed_words = words[::-1]

    # Join the reversed words into a single string
    reversed_string = ' '.join(reversed_words)

    return reversed_string

# Example usage:
input_string = "Hello World"
reversed_string = reverse_words(input_string)
print(reversed_string)  # Output: "World Hello"
```

Longest Palindrome in a string

```python
def longest_palindrome(s):
    def expand_around_center(s, left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
```

```
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Odd length palindromes
        pal_odd = expand_around_center(s, i, i)
        if len(pal_odd) > len(longest):
            longest = pal_odd

        # Even length palindromes
        pal_even = expand_around_center(s, i, i + 1)
        if len(pal_even) > len(longest):
            longest = pal_even

    return longest

# Example usage:
input_string = "babad"
result = longest_palindrome(input_string)
print(result)  # Output: "bab" or "aba"
```

Roman Number to Integer and vice versa

```
def roman_to_integer(s):
    roman_dict = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
    result = 0
    prev_value = 0

    for char in s:
        curr_value = roman_dict[char]
        result += curr_value
        if curr_value > prev_value:
            result -= 2 * prev_value
        prev_value = curr_value

    return result

# Example usage:
roman_numeral = "IX"
integer_value = roman_to_integer(roman_numeral)
print(integer_value)  # Output: 9
```

```
def integer_to_roman(num):
    roman_map = [
```

```python
        (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 = []
    for value, symbol in roman_map:
        while num >= value:
            result.append(symbol)
            num -= value
        if num == 0:
            break

    return ''.join(result)

# Example usage:
integer_value = 9
roman_numeral = integer_to_roman(integer_value)
print(roman_numeral)  # Output: "IX"
```

Implement ATOI/STRSTR

```python
def atoi(s):
    s = s.strip()  # Remove leading and trailing whitespace

    if not s:
        return 0

    sign = 1
    result = 0
    i = 0

    # Check for sign
    if s[i] == '-':
        sign = -1
        i += 1
    elif s[i] == '+':
        i += 1

    # Convert digits
    while i < len(s) and s[i].isdigit():
        result = result * 10 + ord(s[i]) - ord('0')
        i += 1

    # Apply sign
    result *= sign
```

```
    # Handle overflow
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
    if result > INT_MAX:
        return INT_MAX
    elif result < INT_MIN:
        return INT_MIN

    return result

# Example usage:
s = "   -42"
result = atoi(s)
print(result)  # Output: -42
```

```
def strstr(haystack, needle):
    if needle == "":
        return 0

    for i in range(len(haystack) - len(needle) + 1):
        if haystack[i:i+len(needle)] == needle:
            return i

    return -1

# Example usage:
haystack = "hello"
needle = "ll"
result = strstr(haystack, needle)
print(result)  # Output: 2 (index where 'll' starts in 'hello')
```

Longest Common Prefix

```
def longest_common_prefix(strs):
      if not strs:
          return ""

      # Sort the list of strings to easily find the common prefix
      strs.sort()
      # Compare the first and the last string (after sorting) to find common
prefix
      first = strs[0]
      last = strs[-1]
      n = min(len(first), len(last))
      i = 0
      while i < n and first[i] == last[i]:
```

```
        i += 1

    return first[:i]

# Example usage:
strings = ["flower", "flow", "flight"]
result = longest_common_prefix(strings)
print(result)  # Output: "fl"
```

Rabin Karp

```
def rabin_karp(text, pattern):
    if not text or not pattern:
        return -1

    # Constants for hash function
    BASE = 257  # A prime number
    MOD = 2**31  # A large prime number to avoid overflow

    n = len(text)
    m = len(pattern)

    # Calculate BASE^(m-1) % MOD for rolling hash
    base_power = 1
    for i in range(m-1):
        base_power = (base_power * BASE) % MOD

    # Compute hashes
    text_hash = 0
    pattern_hash = 0
    for i in range(m):
        text_hash = (text_hash * BASE + ord(text[i])) % MOD
        pattern_hash = (pattern_hash * BASE + ord(pattern[i])) % MOD

    # Slide the pattern over the text
    for i in range(n - m + 1):
        if text_hash == pattern_hash:
            if text[i:i+m] == pattern:
                return i

        # Update rolling hash for the next window
        if i < n - m:
            text_hash = (text_hash - ord(text[i]) * base_power) % MOD
            text_hash = (text_hash * BASE + ord(text[i+m])) % MOD
            # Ensure text_hash is non-negative
            text_hash = (text_hash + MOD) % MOD
```

```python
    return -1

# Example usage:
text = "abracadabra"
pattern = "cad"
index = rabin_karp(text, pattern)
print(index)  # Output: 4 (index where 'cad' starts in 'abracadabra')
```

Z-Function

```python
def compute_z(s):
    n = len(s)
    Z = [0] * n
    l, r, K = 0, 0, 0

    for i in range(1, n):
        if i > r:
            l, r = i, i
            while r < n and s[r] == s[r - l]:
                r += 1
            Z[i] = r - l
            r -= 1
        else:
            K = i - l
            if Z[K] < r - i + 1:
                Z[i] = Z[K]
            else:
                l = i
                while r < n and s[r] == s[r - l]:
                    r += 1
                Z[i] = r - l
                r -= 1

    Z[0] = n
    return Z

# Example usage:
text = "abacaba"
Z = compute_z(text)
print(Z)  # Output: [7, 0, 1, 0, 3, 0, 1]
```

KMP algo / LPS(pi) array

```python
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length = 0
```

```python
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps
```

KMP Search Algorithm

```python
def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)

    lps = compute_lps(pattern)

    i = 0  # index for text
    j = 0  # index for pattern

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == m:
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
```

Minimum characters needed to be inserted in the beginning to make it palindromic

```python
def min_insertions_to_palindrome(s):
    def compute_lps(pattern):
        m = len(pattern)
        lps = [0] * m
        length = 0
        i = 1

        while i < m:
            if pattern[i] == pattern[length]:
```

```
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

    # Create the concatenated string
    rev_s = s[::-1]
    concat = s + '#' + rev_s

    # Compute the LPS array for the concatenated string
    lps = compute_lps(concat)

    # The number of characters to insert is the difference between the string length and the last value in the LPS array
    return len(s) - lps[-1]

# Example usage
s = "AACECAAAA"
print(f"Minimum insertions needed: {min_insertions_to_palindrome(s)}")
```

Check for Anagrams

```
def are_anagrams(s1, s2):
    # If lengths of both strings are not equal, they cannot be anagrams
    if len(s1) != len(s2):
        return False
    # Sort both strings and compare
    return sorted(s1) == sorted(s2)

# Example usage
s1 = "listen"
s2 = "silent"
print(f"Are '{s1}' and '{s2}' anagrams? {are_anagrams(s1, s2)}")
```

Count and say

```
def count_and_say(n):
    if n == 1:
        return "1"

    def next_sequence(s):
        result = []
        i = 0
        while i < len(s):
            count = 1
```

```
            while i + 1 < len(s) and s[i] == s[i + 1]:
                i += 1
                count += 1
            result.append(f"{count}{s[i]}")
            i += 1
        return ''.join(result)

    current_sequence = "1"
    for _ in range(n - 1):
        current_sequence = next_sequence(current_sequence)

    return current_sequence

# Example usage
n = 5
print(f"The {n}th term of the Count and Say sequence is: {count_and_say(n)}")
```

Compare version numbers

```
def compare_version(version1, version2):
    # Split the version strings into lists of integers
    v1 = list(map(int, version1.split('.')))
    v2 = list(map(int, version2.split('.')))

    # Determine the length of the longer version
    length = max(len(v1), len(v2))

    # Extend the shorter version with zeros (as trailing zeroes do not affect the
comparison)
    v1.extend([0] * (length - len(v1)))
    v2.extend([0] * (length - len(v2)))

    # Compare the versions component by component
    for i in range(length):
        if v1[i] > v2[i]:
            return 1
        elif v1[i] < v2[i]:
            return -1

    # If all components are equal, return 0
    return 0

# Example usage
version1 = "1.0.2"
version2 = "1.0.10"
result = compare_version(version1, version2)
if result == 0:
    print(f"Version {version1} is equal to version {version2}.")
elif result == 1:
    print(f"Version {version1} is greater than version {version2}.")
else:
    print(f"Version {version1} is less than version {version2}.")
```

Inorder Traversal

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def inorder_traversal(root):
    result = []
    def traverse(node):
        if node:
            traverse(node.left)
            result.append(node.value)
            traverse(node.right)
    traverse(root)
    return result

# Example usage:
# Creating a binary tree:
#        1
#         \
#          2
#         /
#        3

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

print(inorder_traversal(root))  # Output: [1, 3, 2]
```

Preorder Traversal

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def preorder_traversal(root):
    result = []
    def traverse(node):
        if node:
            result.append(node.value)
            traverse(node.left)
            traverse(node.right)
    traverse(root)
    return result

# Example usage:
# Creating a binary tree:
#        1
```

```
#         \
#          2
#         /
#        3

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

print(preorder_traversal(root))  # Output: [1, 2, 3]
```

Postorder Traversal

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def postorder_traversal(root):
    result = []
    def traverse(node):
        if node:
            traverse(node.left)
            traverse(node.right)
            result.append(node.value)
    traverse(root)
    return result

# Example usage:
# Creating a binary tree:
#        1
#         \
#          2
#         /
#        3

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

print(postorder_traversal(root))  # Output: [3, 2, 1]
```

Morris Inorder Traversal

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

```
def morris_inorder_traversal(root):
    result = []
    current = root

    while current:
        if current.left is None:
            result.append(current.value)
            current = current.right
        else:
            # Find the inorder predecessor of current
            predecessor = current.left
          while predecessor.right and predecessor.  right != current:
                predecessor = predecessor.right

            if predecessor.right is None:
              # Make current the right child of its inorder predecessor
                predecessor.right = current
                current = current.left
            else:
                # Revert the changes made in the tree
                predecessor.right = None
                result.append(current.value)
                current = current.right

    return result

# Example usage:
# Creating a binary tree:
#        1
#         \
#          2
#         /
#        3

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

print(morris_inorder_traversal(root))  # Output: [1, 3, 2]
```

Morris Preorder Traversal

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def morris_preorder_traversal(root):
    result = []
    current = root

    while current:
```

```
        if current.left is None:
            result.append(current.value)
            current = current.right
        else:
            # Find the inorder predecessor of current
          predecessor = current.left
            while predecessor.right and predecessor.  right != current:
                predecessor = predecessor.right

            if predecessor.right is None:
                # Visit the current node before threading
              result.append(current.value)
                # Make current the right child of its   inorder predecessor
                predecessor.right = current
                current = current.left
            else:
                # Revert the changes made in the tree
                predecessor.right = None
                current = current.right

    return result

# Example usage:
# Creating a binary tree:
#        1
#         \
#          2
#         /
#        3

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

print(morris_preorder_traversal(root))  # Output: [1, 2, 3]
```

LeftView Of Binary Tree

```
from collections import deque

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def left_view(root):
    if not root:
        return []

    result = []
    queue = deque([root])
```

```
    while queue:
        level_size = len(queue)
        for i in range(level_size):
            node = queue.popleft()
          # Append the first node encountered at each   level (leftmost node)
            if i == 0:
                result.append(node.value)

            # Add left and right children to the queue
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    return result

# Example usage:
# Creating a binary tree:
#         1
#        / \
#       2   3
#      / \
#     4   5

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print(left_view(root))  # Output: [1, 2, 4]
```

Bottom View of Binary Tree

```
from collections import deque, defaultdict

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def bottom_view(root):
    if not root:
        return []

  # Dictionary to store the last node at each   horizontal distance
    bottom_view_map = defaultdict(TreeNode)
  queue = deque([(root, 0)])  # (node, horizontal   distance)

    while queue:
        node, hd = queue.popleft()
```

```
        # Update the bottom view map with the current   node for its horizontal
distance
          bottom_view_map[hd] = node

        # Add left and right children to the queue with   updated horizontal distances
          if node.left:
              queue.append((node.left, hd - 1))
          if node.right:
              queue.append((node.right, hd + 1))

    # Extract the bottom view nodes from the map, sorted   by horizontal distance
    bottom_view = [bottom_view_map[hd].value for hd
in   sorted(bottom_view_map.keys())]

      return bottom_view

  # Example usage:
  # Creating a binary tree:
  #         1
  #       /   \
  #      2     3
  #     / \   / \
  #    4   5 6   7
  #             /
  #            8

  root = TreeNode(1)
  root.left = TreeNode(2)
  root.right = TreeNode(3)
  root.left.left = TreeNode(4)
  root.left.right = TreeNode(5)
  root.right.left = TreeNode(6)
  root.right.right = TreeNode(7)
  root.right.right.left = TreeNode(8)

  print(bottom_view(root))  # Output: [4, 2, 6, 8, 7]
```

Top View of Binary Tree

```
  from collections import deque, defaultdict

  class TreeNode:
      def __init__(self, value=0, left=None, right=None):
          self.value = value
          self.left = left
          self.right = right

  def top_view(root):
      if not root:
          return []

    # Dictionary to store the first node at each   horizontal distance
      top_view_map = defaultdict(TreeNode)
```

```python
    queue = deque([(root, 0)])  # (node, horizontal   distance)

    while queue:
        node, hd = queue.popleft()
      # Store the first encountered node for each   horizontal distance
        if hd not in top_view_map:
            top_view_map[hd] = node

      # Add left and right children to the queue with   updated horizontal distances
        if node.left:
            queue.append((node.left, hd - 1))
        if node.right:
            queue.append((node.right, hd + 1))

  # Extract the top view nodes from the map, sorted by   horizontal distance
  top_view = [top_view_map[hd].value for hd in sorted  (top_view_map.keys())]

    return top_view

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7
#             /
#            8

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
root.right.right.left = TreeNode(8)

print(top_view(root))  # Output: [4, 2, 1, 3, 7]
```

Preorder inorder postorder in a single traversal

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def triple_traversal(root):
    if not root:
        return [], [], []

    preorder = []
```

```
    inorder = []
    postorder = []

    stack = [(root, False, False)]  # (node,   visited_left, visited_right)

    while stack:
        node, visited_left, visited_right = stack.pop()

        if not visited_left:
            preorder.append(node.value)
            stack.append((node, True, visited_right))
            if node.left:
                stack.append((node.left, False, False))
                continue

        if visited_left and not visited_right:
            inorder.append(node.value)
            stack.append((node, True, True))
            if node.right:
                stack.append((node.right, False, False))
                continue

        if visited_left and visited_right:
            postorder.append(node.value)

    return preorder, inorder, postorder

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

preorder, inorder, postorder = triple_traversal(root)

print("Preorder:", preorder)   # Output: [1, 2, 4, 5, 3, 6, 7]
print("Inorder:", inorder)     # Output: [4, 2, 5, 1, 6, 3, 7]
print("Postorder:", postorder) # Output: [4, 5, 2, 6, 7, 3, 1]
```

Vertical order traversal

```
from collections import defaultdict, deque
```

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def vertical_order_traversal(root):
    if not root:
        return []

    # Dictionary to store nodes at each horizontal   distance
  vertical_order_map = defaultdict(list)
    queue = deque([(root, 0)])  # (node, horizontal   distance)

    while queue:
      node, hd = queue.popleft()
        # Add current node to its horizontal distance   list
        vertical_order_map[hd].append(node.value)

        # Add left and right children to the queue with   updated horizontal
distances
        if node.left:
            queue.append((node.left, hd - 1))
        if node.right:
            queue.append((node.right, hd + 1))

    # Extract the vertical order nodes from the map, sorted by horizontal distance
    vertical_order = [vertical_order_map[hd] for hd in
sorted(vertical_order_map.keys())]

    return vertical_order

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7
#             /
#            8

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
root.right.right.left = TreeNode(8)

print(vertical_order_traversal(root))
# Output: [[4], [2], [1, 5, 6], [3, 8], [7]]
```

Root to Node Path in Binary Tree

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def find_path(root, target):
    # Helper function to perform DFS
    def dfs(node, path):
        if not node:
            return False

        # Append the current node to the path
        path.append(node.value)

        # Check if the current node is the target
        if node.value == target:
            return True

        # Recursively search in the left and right   subtrees
        if dfs(node.left, path) or dfs(node.right, path):
            return True

        # If target not found, backtrack (remove the   current node from path)
        path.pop()
        return False

    # Initialize an empty list to store the path
    path = []

    # Start DFS from the root node
    if dfs(root, path):
        return path
    else:
        return None

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
```

```python
target = 5
print(f"Path to node {target}: {find_path(root, target)}")  # Output: [1, 2, 5]
```

Max width of a Binary Tree

```python
from collections import deque

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def max_width(root):
    if not root:
        return 0

    max_width = 0
    queue = deque([(root, 0)])  # (node, index)

    while queue:
        level_size = len(queue)
        max_width = max(max_width, level_size)

        for _ in range(level_size):
            node, index = queue.popleft()

            if node.left:
                queue.append((node.left, 2 * index))
            if node.right:
                queue.append((node.right, 2 * index + 1))

    return max_width

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \     \
#    4   5     7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(7)

print("Maximum width of the binary tree:", max_width(root))  # Output: 3
```

Level order Traversal / Level order traversal in spiral form

```python
from collections import deque

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
```

```python
        self.value = value
        self.left = left
        self.right = right

def level_order_traversal(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        level_nodes = []

        for _ in range(level_size):
            node = queue.popleft()
            level_nodes.append(node.value)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level_nodes)

    return result

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print("Level Order Traversal:", level_order_traversal(root))
# Output: [[1], [2, 3], [4, 5, 6, 7]]
```

```python
def level_order_spiral(root):
    if not root:
        return []

    result = []
    queue = deque([root])
    level_number = 1  # to track odd or even level

    while queue:
        level_size = len(queue)
        level_nodes = []
```

```python
        for _ in range(level_size):
            node = queue.popleft()

            if level_number % 2 != 0:
                level_nodes.append(node.value)
            else:
                level_nodes.insert(0, node.value)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level_nodes)
        level_number += 1

    return result

# Example usage:
# Using the same binary tree as before

print("Level Order Traversal in Spiral Form:",   level_order_spiral(root))
# Output: [[1], [3, 2], [4, 5, 6, 7]]
```

Height of a Binary Tree

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def tree_height(root):
    if not root:
        return 0

    left_height = tree_height(root.left)
    right_height = tree_height(root.right)

    # Height of the tree is the maximum height of left and right subtrees + 1 (for the root)
    return max(left_height, right_height) + 1

# Example usage:
# Creating a binary tree:
#          1
#        /   \
#       2     3
#      / \   / \
#     4   5 6   7
#              /
#             8

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
```

```
root.right.right.left = TreeNode(8)

print("Height of the binary tree:", tree_height(root))  # Output: 4
```

Diameter of Binary Tree

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def tree_diameter(root):
    # Helper function to calculate height and diameter
    def calculate_height_and_diameter(node):
        if not node:
            return 0, 0

        left_height, left_diameter = calculate_height_and_diameter(node.left)
        right_height, right_diameter = calculate_height_and_diameter(node.right)

        # Height of the current node
        current_height = max(left_height, right_height) + 1

        # Diameter passing through the current node
        current_diameter = max(left_height + right_height, left_diameter, right_diameter)

        return current_height, current_diameter

    # Call the helper function with the root node
    _, diameter = calculate_height_and_diameter(root)
    return diameter

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7
#             /
#            8

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
root.right.right.left = TreeNode(8)

print("Diameter of the binary tree:", tree_diameter(root))  # Output: 6
```

Check if the Binary tree is height-balanced or not

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
```

```python
        self.value = value
        self.left = left
        self.right = right

def is_balanced(root):
    # Helper function to calculate height and check balance
    def check_balance(node):
        if not node:
            return True, 0

        # Check balance and calculate height of left subtree
        is_left_balanced, left_height = check_balance(node.left)
        if not is_left_balanced:
            return False, 0

        # Check balance and calculate height of right subtree
        is_right_balanced, right_height = check_balance(node.right)
        if not is_right_balanced:
            return False, 0

        # Calculate the height of the current node
        current_height = max(left_height, right_height) + 1

        # Check if current node is balanced
        if abs(left_height - right_height) > 1:
            return False, current_height

        return True, current_height

    # Call the helper function with the root node
    balanced, _ = check_balance(root)
    return balanced

# Example usage:
# Creating a height-balanced binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root_balanced = TreeNode(1)
root_balanced.left = TreeNode(2)
root_balanced.right = TreeNode(3)
root_balanced.left.left = TreeNode(4)
root_balanced.left.right = TreeNode(5)
root_balanced.right.left = TreeNode(6)
root_balanced.right.right = TreeNode(7)

print("Is the binary tree balanced?", is_balanced(root_balanced))  # Output: True

# Creating an unbalanced binary tree:
#         1
#       /   \
#      2     3
#     /       \
#    4         5
#             /
#            6

root_unbalanced = TreeNode(1)
root_unbalanced.left = TreeNode(2)
```

```
root_unbalanced.right = TreeNode(3)
root_unbalanced.left.left = TreeNode(4)
root_unbalanced.right.right = TreeNode(5)
root_unbalanced.right.right.left = TreeNode(6)

print("Is the binary tree balanced?", is_balanced(root_unbalanced))  # Output: False
```

LCA in Binary Tree

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def find_lca(root, p, q):
      # Base case: If root is None or if root is one of the nodes   p or q
      if not root or root.value == p or root.value == q:
          return root

      # Recursively find LCA in left and right subtrees
      left_lca = find_lca(root.left, p, q)
      right_lca = find_lca(root.right, p, q)

      # If both left_lca and right_lca are not None, then root is   the LCA
      if left_lca and right_lca:
          return root

      # Otherwise, return whichever is not None (either left_lca   or right_lca)
      return left_lca if left_lca else right_lca

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Example 1: LCA of nodes 4 and 5
p1, q1 = 4, 5
lca1 = find_lca(root, p1, q1)
print(f"LCA of nodes {p1} and {q1} is {lca1.value}")  # Output:   2

# Example 2: LCA of nodes 6 and 7
p2, q2 = 6, 7
lca2 = find_lca(root, p2, q2)
print(f"LCA of nodes {p2} and {q2} is {lca2.value}")  # Output: 3
```

Check if two trees are identical or not

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def are_identical(root1, root2):
    # Base case: If both roots are None, they are identical
    if not root1 and not root2:
        return True

    # If one root is None and the other is not, they are not   identical
    if not root1 or not root2:
        return False

    # Check if the current nodes have the same value
    if root1.value != root2.value:
        return False

    # Recursively check left and right subtrees
    return (are_identical(root1.left, root2.left) and
            are_identical(root1.right, root2.right))

# Example usage:
# Creating two identical binary trees:
# Tree 1:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root1 = TreeNode(1)
root1.left = TreeNode(2)
root1.right = TreeNode(3)
root1.left.left = TreeNode(4)
root1.left.right = TreeNode(5)
root1.right.left = TreeNode(6)
root1.right.right = TreeNode(7)

# Tree 2 (identical to Tree 1):
root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.right = TreeNode(3)
root2.left.left = TreeNode(4)
root2.left.right = TreeNode(5)
root2.right.left = TreeNode(6)
root2.right.right = TreeNode(7)

print("Are the binary trees identical?", are_identical(root1,   root2))  # Output: True

# Creating two different binary trees:
# Tree 3:
#         1
#       /   \
#      2     3
#     / \   /
#    4   5 6

root3 = TreeNode(1)
```

```python
root3.left = TreeNode(2)
root3.right = TreeNode(3)
root3.left.left = TreeNode(4)
root3.left.right = TreeNode(5)
root3.right.left = TreeNode(6)

# Tree 4:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root4 = TreeNode(1)
root4.left = TreeNode(2)
root4.right = TreeNode(3)
root4.left.left = TreeNode(4)
root4.left.right = TreeNode(5)
root4.right.left = TreeNode(6)
root4.right.right = TreeNode(7)

print("Are the binary trees identical?", are_identical(root3, root4))  # Output: False
```

Zig Zag Traversal of Binary Tree

```python
from collections import deque

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def zigzag_traversal(root):
    if not root:
        return []

    result = []
    queue = deque([root])
    level_number = 1  # to track odd or even level

    while queue:
        level_size = len(queue)
        level_nodes = []

        for _ in range(level_size):
            node = queue.popleft()

            # Depending on level_number, append node value to    level_nodes
            if level_number % 2 != 0:
                level_nodes.append(node.value)
            else:
                level_nodes.insert(0, node.value)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level_nodes)
```

```python
        level_number += 1

    return result

# Example usage:
# Creating a binary tree:
#         1
#       /   \
#      2     3
#     / \   / \
#    4   5 6   7

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print("Zigzag traversal of the binary tree:", zigzag_traversal(root))
# Output: [[1], [3, 2], [4, 5, 6, 7]]
```

Boundary Traversal of Binary Tree

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def is_leaf(node):
    return node.left is None and node.right is None

def add_leaves(node, result):
    if not node:
        return
    if is_leaf(node):
        result.append(node.value)
    add_leaves(node.left, result)
    add_leaves(node.right, result)

def boundary_traversal(root):
    if not root:
        return []

    result = []
    if not is_leaf(root):
        result.append(root.value)

    # Traverse left boundary (excluding leaf node)
    left = root.left
    while left:
        if not is_leaf(left):
            result.append(left.value)
        if left.left:
            left = left.left
        else:
            left = left.right
```

```
    # Add leaf nodes
    add_leaves(root, result)

    # Traverse right boundary (excluding leaf node)
    right = root.right
    boundary = []
    while right:
        if not is_leaf(right):
            boundary.append(right.value)
        if right.right:
            right = right.right
        else:
            right = right.left

    # Reverse and append right boundary to result
    result.extend(reversed(boundary))

    return result

# Example usage:
# Creating a binary tree:
#          1
#        /   \
#       2     3
#      / \   / \
#     4   5 6   7
#    / \
#   8   9

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
root.left.left.left = TreeNode(8)
root.left.left.right = TreeNode(9)

print("Boundary traversal of the binary tree:", boundary_traversal(root))
# Output: [1, 2, 4, 8, 9, 5, 6, 7, 3]
```

Maximum path sum

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxPathSum(root):
  # Initialize a global variable to store the maximum sum   found
    max_sum = float('-inf')

    def maxPathSumRecursive(node):
        nonlocal max_sum
        if not node:
            return 0

      # Recursively calculate maximum path sums in left and   right subtrees
```

```
        left_sum = max(0, maxPathSumRecursive(node.left))
        right_sum = max(0, maxPathSumRecursive(node.right))

      # Calculate the maximum path sum that includes the   current node
        max_single = node.val + max(left_sum, right_sum)

        # Update the global maximum sum found so far
        max_sum = max(max_sum, node.val + left_sum + right_sum)

      # Return the maximum path sum starting from the current   node
        return max_single

    # Start the recursive calculation from the root
    maxPathSumRecursive(root)

    return max_sum
```

Construct Binary Tree from inorder and preorder

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def buildTree(preorder, inorder):
    # Edge case
    if not preorder or not inorder:
        return None

    # First element in preorder is the root
    root_val = preorder[0]
    root = TreeNode(root_val)

    # Find the index of root in inorder traversal
    root_index_in_inorder = inorder.index(root_val)

    # Recursively build left and right subtrees
    root.left = buildTree(preorder[1:root_index_in_inorder + 1],
inorder[:root_index_in_inorder])
    root.right = buildTree(preorder[root_index_in_inorder + 1:], inorder[root_index_in_inorder
+ 1:])

    return root

# Example usage:
preorder = [3, 9, 20, 15, 7]
inorder = [9, 3, 15, 20, 7]

root = buildTree(preorder, inorder) # Now 'root' contains the root of the reconstructed binary
tree
```

Construct Binary Tree from Inorder and Postorder

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
```

```
        self.right = right

  def buildTree(inorder, postorder):
      # Edge case
      if not inorder or not postorder:
          return None

      # Last element in postorder is the root
      root_val = postorder[-1]
      root = TreeNode(root_val)

      # Find the index of root in inorder traversal
      root_index_in_inorder = inorder.index(root_val)

      # Recursively build left and right subtrees
      root.left = buildTree(inorder[:root_index_in_inorder], postorder[:root_index_in_inorder])
      root.right = buildTree(inorder[root_index_in_inorder + 1:],
postorder[root_index_in_inorder:-1])

      return root

  # Example usage:
  inorder = [9, 3, 15, 20, 7]
  postorder = [9, 15, 7, 20, 3]

  root = buildTree(inorder, postorder) # Now 'root' contains the root of the reconstructed
binary tree
```

Symmetric Binary Tree

```
  class TreeNode:
      def __init__(self, val=0, left=None, right=None):
          self.val = val
          self.left = left
          self.right = right

  def isSymmetric(root):
      def isMirror(node1, node2):
          if not node1 and not node2:
              return True
          if not node1 or not node2:
              return False
          return (node1.val == node2.val) and isMirror(node1.left, node2.right) and
isMirror(node1.right, node2.left)

      if not root:
          return True
      return isMirror(root.left, root.right)

  # Construct a symmetric binary tree
  root = TreeNode(1)
  root.left = TreeNode(2)
  root.right = TreeNode(2)
  root.left.left = TreeNode(3)
  root.left.right = TreeNode(4)
  root.right.left = TreeNode(4)
  root.right.right = TreeNode(3)

  print(isSymmetric(root))  # Output: True
```

Flatten Binary Tree to LinkedList

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def flatten(root):
    if not root:
        return

    # Flatten the left subtree
    flatten(root.left)

    # Flatten the right subtree
    flatten(root.right)

    # Save the original right subtree
    original_right = root.right

    # Attach flattened left subtree to the right of the root
    root.right = root.left
    root.left = None  # Clear the left pointer

    # Find the end of the new flattened subtree to attach the original right subtree
    current = root
    while current.right:
        current = current.right
    current.right = original_right

# Example usage:
# Construct a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(5)
root.left.left = TreeNode(3)
root.left.right = TreeNode(4)
root.right.right = TreeNode(6)

flatten(root)

# Print the flattened tree (linked list)
current = root
while current:
    print(current.val, end=" ")
    current = current.right


# Construct a binary tree

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(5)
root.left.left = TreeNode(3)
root.left.right = TreeNode(4)
root.right.right = TreeNode(6)

flatten(root)

# Print the flattened tree (linked list)
current = root
```

```
while current:
    print(current.val, end=" ")
    current = current.right


# Output: 1 2 3 4 5 6
```

Check if Binary Tree is the mirror of itself or not

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def isSymmetric(root):
    def isMirror(left, right):
        if not left and not right:
            return True
        if not left or not right:
            return False
      return (left.val == right.val) and isMirror(left.left, right.right) and
isMirror(left.right, right.left)

    if not root:
        return True
    return isMirror(root.left, root.right)

# Example usage:
# Construct a symmetric binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(4)
root.right.left = TreeNode(4)
root.right.right = TreeNode(3)

print(isSymmetric(root))  # Output: True

# Construct a non-symmetric binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.right = TreeNode(3)
root.right.right = TreeNode(3)

print(isSymmetric(root))  # Output: False
```

Check for Children Sum Property

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def isChildrenSumProperty(root):
```

```
    if not root:
        return True

    # Calculate the sum of values of left and right children
    sum_children = 0
    if root.left:
        sum_children += root.left.val
    if root.right:
        sum_children += root.right.val

    # Check if the node's value is equal to the sum of its children
    if root.val != sum_children:
        return False

    # Recursively check for left and right subtrees
    return isChildrenSumProperty(root.left) and isChildrenSumProperty(root.right)

# Example usage:
# Construct a binary tree satisfying Children Sum Property
root = TreeNode(10)
root.left = TreeNode(8)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(1)

print(isChildrenSumProperty(root))  # Output: True

# Construct a binary tree not satisfying Children Sum Property
root = TreeNode(10)
root.left = TreeNode(8)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(11)  # Making it invalid

print(isChildrenSumProperty(root))  # Output: False
```

Populate Next Right pointers of Tree

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None, next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next

def connect(root):
    if not root:
        return

    # Helper function to connect nodes at the same level
    def connect_nodes(node):
        if not node:
            return

        if node.left:
            node.left.next = node.right
            if node.next:
                node.right.next = node.next.left
```

```python
        connect_nodes(node.left)
        connect_nodes(node.right)

    connect_nodes(root)
    return root

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node1 = TreeNode(1)
    node2 = TreeNode(2)
    node3 = TreeNode(3)
    node4 = TreeNode(4)
    node5 = TreeNode(5)
    node6 = TreeNode(6)
    node7 = TreeNode(7)

    # Level 1
    node1.left = node2
    node1.right = node3
    node2.left = node4
    node2.right = node5
    node3.left = node6
    node3.right = node7

    return node1  # Return the root node

root = build_sample_bst()
connected_root = connect(root)

# Print the next pointers to verify
def print_next_pointers(root):
    if not root:
        return

    current_level = root
    while current_level:
        current_node = current_level
        while current_node:
            print(f"Node {current_node.val}: Next ->   {current_node.next.val if
current_node.next else   'None'}")
            current_node = current_node.next
        print("\n")
        current_level = current_level.left

print_next_pointers(connected_root)
```

Search given Key in BST

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def search_bst(root, key):
    if not root or root.val == key:
        return root
```

```
    if key < root.val:
        return search_bst(root.left, key)
    else:
        return search_bst(root.right, key)

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node4 = TreeNode(4)
    node2 = TreeNode(2)
    node7 = TreeNode(7)
    node1 = TreeNode(1)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node9 = TreeNode(9)

    # Level 1
    node4.left = node2
    node4.right = node7
    node2.left = node1
    node2.right = node3
    node7.left = node6
    node7.right = node9

    return node4  # Return the root node

root = build_sample_bst()
key = 6
result_node = search_bst(root, key)

if result_node:
    print(f"Node with value {key} found in the BST.")
else:
    print(f"Node with value {key} not found in the BST.")
```

Construct BST from given keys

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def insert_into_bst(root, key):
    if not root:
        return TreeNode(key)

    if key < root.val:
        root.left = insert_into_bst(root.left, key)
    else:
        root.right = insert_into_bst(root.right, key)

    return root

def construct_bst(keys):
    if not keys:
        return None
```

```
    root = None
    for key in keys:
        root = insert_into_bst(root, key)

    return root

# Example usage:
keys = [7, 4, 2, 1, 3, 5, 9, 8, 11, 10, 12]
root = construct_bst(keys)

# Function to perform an inorder traversal (for verification purposes)
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.val, end=" ")
        inorder_traversal(node.right)

print("Inorder traversal of constructed BST:")
inorder_traversal(root)
```

Construct a BST from a preorder traversal

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def bst_from_preorder(preorder):
    if not preorder:
        return None

    root = TreeNode(preorder[0])
    stack = [root]

    for value in preorder[1:]:
        node = TreeNode(value)

        # Find where this node should be placed
        if value < stack[-1].val:
            stack[-1].left = node
        else:
            parent = None
            while stack and value > stack[-1].val:
                parent = stack.pop()
            parent.right = node

        stack.append(node)

    return root

# Example usage:
preorder = [8, 5, 1, 7, 10, 9, 12]
root = bst_from_preorder(preorder)

# Function to perform an inorder traversal (for verification purposes)
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.val, end=" ")
```

```
        inorder_traversal(node.right)

print("Inorder traversal of constructed BST:")
inorder_traversal(root)
```

Check is a BT is BST or not

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_bst(root):
    def is_bst_util(node, min_val, max_val):
        if not node:
            return True

        if node.val <= min_val or node.val >= max_val:
            return False

        return (is_bst_util(node.left, min_val, node.val) and
                is_bst_util(node.right, node.val, max_val))

    # Call the utility function with initial minimum and maximum values
    return is_bst_util(root, float('-inf'), float('inf'))

# Example usage:
# Constructing a sample BT which is also a BST
def build_sample_bst():
    # Level 0
    node4 = TreeNode(4)
    node2 = TreeNode(2)
    node7 = TreeNode(7)
    node1 = TreeNode(1)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node9 = TreeNode(9)

    # Level 1
    node4.left = node2
    node4.right = node7
    node2.left = node1
    node2.right = node3
    node7.left = node6
    node7.right = node9

    return node4  # Return the root node

root = build_sample_bst()
print("Is the binary tree a BST?", is_bst(root))

# Constructing a sample BT which is not a BST
def build_sample_non_bst():
    # Level 0
    node4 = TreeNode(4)
    node2 = TreeNode(2)
    node5 = TreeNode(5)
    node1 = TreeNode(1)
    node3 = TreeNode(3)
```

```
    node6 = TreeNode(6)
    node7 = TreeNode(7)

    # Level 1
    node4.left = node2
    node4.right = node5
    node2.left = node1
    node2.right = node3
    node5.right = node6
    node6.right = node7

    return node4  # Return the root node

root_non_bst = build_sample_non_bst()
print("Is the binary tree a BST?", is_bst(root_non_bst))
```

Find LCA of two nodes in BST

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find_lca_bst(root, node1, node2):
    if not root:
        return None

    # Ensure node1 is smaller than or equal to node2
    if node1.val > node2.val:
        node1, node2 = node2, node1

    while root:
        if root.val < node1.val:
            root = root.right
        elif root.val > node2.val:
            root = root.left
        else:
            return root

    return None

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node4 = TreeNode(4)
    node2 = TreeNode(2)
    node7 = TreeNode(7)
    node1 = TreeNode(1)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node9 = TreeNode(9)

    # Level 1
    node4.left = node2
    node4.right = node7
    node2.left = node1
    node2.right = node3
    node7.left = node6
```

```python
    node7.right = node9

    return node4  # Return the root node

root = build_sample_bst()

# Finding LCA of nodes 2 and 7
node2 = root.left
node7 = root.right
lca = find_lca_bst(root, node2, node7)
print(f"LCA of {node2.val} and {node7.val} is: {lca.val if lca else None}")

# Finding LCA of nodes 2 and 3
node3 = node2.right
lca = find_lca_bst(root, node2, node3)
print(f"LCA of {node2.val} and {node3.val} is: {lca.val if lca else None}")
```

Find the inorder predecessor/successor of a given Key in BST

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find_predecessor(root, key):
    predecessor = None
    while root:
        if key > root.val:
            predecessor = root
            root = root.right
        else:
            root = root.left
    return predecessor

def find_successor(root, key):
    successor = None
    while root:
        if key < root.val:
            successor = root
            root = root.left
        else:
            root = root.right
    return successor

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node4 = TreeNode(4)
    node2 = TreeNode(2)
    node7 = TreeNode(7)
    node1 = TreeNode(1)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node9 = TreeNode(9)

    # Level 1
    node4.left = node2
    node4.right = node7
```

```python
    node2.left = node1
    node2.right = node3
    node7.left = node6
    node7.right = node9

    return node4  # Return the root node

root = build_sample_bst()
key = 4

predecessor = find_predecessor(root, key)
successor = find_successor(root, key)

print(f"Inorder predecessor of {key} is: {predecessor.val if predecessor else None}")
print(f"Inorder successor of {key} is: {successor.val if successor else None}")
```

Floor in a BST

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find_floor(root, key):
    floor = None
    while root:
        if root.val == key:
            return root
        elif root.val > key:
            root = root.left
        else:
            floor = root
            root = root.right
    return floor

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node8 = TreeNode(8)
    node4 = TreeNode(4)
    node12 = TreeNode(12)
    node2 = TreeNode(2)
    node6 = TreeNode(6)
    node10 = TreeNode(10)
    node14 = TreeNode(14)

    # Level 1
    node8.left = node4
    node8.right = node12
    node4.left = node2
    node4.right = node6
    node12.left = node10
    node12.right = node14

    return node8  # Return the root node

root = build_sample_bst()
key = 5
```

```
floor_node = find_floor(root, key)
print(f"Floor of {key} is: {floor_node.val if floor_node else None}")
```

Ceil in a BST

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find_ceil(root, key):
    ceil = None
    while root:
        if root.val == key:
            return root
        elif root.val < key:
            root = root.right
        else:
            ceil = root
            root = root.left
    return ceil

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node8 = TreeNode(8)
    node4 = TreeNode(4)
    node12 = TreeNode(12)
    node2 = TreeNode(2)
    node6 = TreeNode(6)
    node10 = TreeNode(10)
    node14 = TreeNode(14)

    # Level 1
    node8.left = node4
    node8.right = node12
    node4.left = node2
    node4.right = node6
    node12.left = node10
    node12.right = node14

    return node8  # Return the root node

root = build_sample_bst()
key = 5

ceil_node = find_ceil(root, key)
print(f"Ceil of {key} is: {ceil_node.val if ceil_node else None}")
```

Find K-th smallest element in BST

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
```

```python
        self.right = right

def kth_smallest(root, k):
    # Helper function to perform inorder traversal
    def inorder(node):
        if not node or self.count >= k:
            return
        inorder(node.left)
        self.count += 1
        if self.count == k:
            self.result = node.val
            return
        inorder(node.right)

    self.count = 0
    self.result = None
    inorder(root)
    return self.result

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node5 = TreeNode(5)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node2 = TreeNode(2)
    node4 = TreeNode(4)
    node1 = TreeNode(1)

    # Level 1
    node5.left = node3
    node5.right = node6
    node3.left = node2
    node3.right = node4
    node2.left = node1

    return node5  # Return the root node

root = build_sample_bst()
k = 3
print(f"The {k}-th smallest element in the BST is: {kth_smallest(root, k)}")
```

Find K-th largest element in BST

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def kth_largest(root, k):
    # Helper function to perform reverse inorder traversal
    def reverse_inorder(node):
        if not node or self.count >= k:
            return
        reverse_inorder(node.right)
        self.count += 1
        if self.count == k:
            self.result = node.val
```

```
            return
        reverse_inorder(node.left)

    self.count = 0
    self.result = None
    reverse_inorder(root)
    return self.result

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node5 = TreeNode(5)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node2 = TreeNode(2)
    node4 = TreeNode(4)
    node1 = TreeNode(1)

    # Level 1
    node5.left = node3
    node5.right = node6
    node3.left = node2
    node3.right = node4
    node2.left = node1

    return node5  # Return the root node

root = build_sample_bst()
k = 2
print(f"The {k}-th largest element in the BST is: {kth_largest(root, k)}")
```

Find a pair with a given sum in BST

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def inorder_traversal(root, result):
    if not root:
        return
    inorder_traversal(root.left, result)
    result.append(root.val)
    inorder_traversal(root.right, result)

def find_pair_with_sum(root, target_sum):
    if not root:
        return None

    # Get sorted list of values from the BST
    nodes = []
    inorder_traversal(root, nodes)

    # Use two-pointer technique to find the pair
    left, right = 0, len(nodes) - 1
    while left < right:
        current_sum = nodes[left] + nodes[right]
        if current_sum == target_sum:
```

```python
                return (nodes[left], nodes[right])
            elif current_sum < target_sum:
                left += 1
            else:
                right -= 1

    return None

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node5 = TreeNode(5)
    node3 = TreeNode(3)
    node6 = TreeNode(6)
    node2 = TreeNode(2)
    node4 = TreeNode(4)
    node1 = TreeNode(1)

    # Level 1
    node5.left = node3
    node5.right = node6
    node3.left = node2
    node3.right = node4
    node2.left = node1

    return node5  # Return the root node

root = build_sample_bst()
target_sum = 9
pair = find_pair_with_sum(root, target_sum)
if pair:
    print(f"Pair with sum {target_sum} is: {pair}")
else:
    print(f"No pair found with sum {target_sum}")
```

BST iterator

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class BSTIterator:
    def __init__(self, root):
        self.stack = []
        self._leftmost_inorder(root)

    def _leftmost_inorder(self, root):
        while root:
            self.stack.append(root)
            root = root.left

    def next(self):
        """
        @return the next smallest number
        """
      # Node at the top of the stack is the next smallest   element
        topmost_node = self.stack.pop()
```

```python
        # If the node has a right child, call the helper   function for the right child
        if topmost_node.right:
            self._leftmost_inorder(topmost_node.right)

        return topmost_node.val

    def hasNext(self):
        """
        @return whether we have a next smallest number
        """
        return len(self.stack) > 0

# Example usage:
# Constructing a sample BST
def build_sample_bst():
    # Level 0
    node7 = TreeNode(7)
    node3 = TreeNode(3)
    node15 = TreeNode(15)
    node9 = TreeNode(9)
    node20 = TreeNode(20)

    # Level 1
    node7.left = node3
    node7.right = node15
    node15.left = node9
    node15.right = node20

    return node7  # Return the root node

root = build_sample_bst()
iterator = BSTIterator(root)
while iterator.hasNext():
    print(iterator.next())
```

Size of the largest BST in a Binary Tree

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class ReturnType:
    def __init__(self, is_bst, size, min_val, max_val):
        self.is_bst = is_bst
        self.size = size
        self.min_val = min_val
        self.max_val = max_val

def largest_bst_subtree(root):
    def postorder(node):
        if not node:
            return ReturnType(True, 0, float('inf'), float('-inf'))

        left = postorder(node.left)
        right = postorder(node.right)
```

```python
        if left.is_bst and right.is_bst and node.val > left.max_val and node.val < right.min_val:
            size = left.size + right.size + 1
            min_val = min(left.min_val, node.val)
            max_val = max(right.max_val, node.val)
            return ReturnType(True, size, min_val, max_val)

        return ReturnType(False, max(left.size, right.size), 0, 0)

    return postorder(root).size

# Example usage:
# Constructing a sample binary tree
def build_sample_binary_tree():
    # Level 0
    node10 = TreeNode(10)
    node5 = TreeNode(5)
    node15 = TreeNode(15)
    node1 = TreeNode(1)
    node8 = TreeNode(8)
    node7 = TreeNode(7)
    node12 = TreeNode(12)
    node20 = TreeNode(20)

    # Level 1
    node10.left = node5
    node10.right = node15
    node5.left = node1
    node5.right = node8
    node15.right = node7
    node15.left = node12
    node12.right = node20

    return node10  # Return the root node

root = build_sample_binary_tree()
print(f"The size of the largest BST in the given binary tree is: {largest_bst_subtree(root)}")
```

Serialize and deserialize Binary Tree

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string."""
        def helper(node):
            if not node:
                result.append('#')
                return
            result.append(str(node.val))
            helper(node.left)
            helper(node.right)

        result = []
        helper(root)
        return ' '.join(result)
```

```python
    def deserialize(self, data):
        """Decodes your encoded data to tree."""
        def helper():
            val = next(values)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = helper()
            node.right = helper()
            return node

        values = iter(data.split())
        return helper()

# Example usage:
# Constructing a sample binary tree
def build_sample_binary_tree():
    # Level 0
    node1 = TreeNode(1)
    node2 = TreeNode(2)
    node3 = TreeNode(3)
    node4 = TreeNode(4)
    node5 = TreeNode(5)

    # Level 1
    node1.left = node2
    node1.right = node3
    node3.left = node4
    node3.right = node5

    return node1  # Return the root node

# Create a binary tree
root = build_sample_binary_tree()

# Serialize the binary tree
codec = Codec()
serialized = codec.serialize(root)
print(f"Serialized tree: {serialized}")

# Deserialize the string back to a binary tree
deserialized_root = codec.deserialize(serialized)
print(f"Deserialized tree (root value): {deserialized_root.val}")
```

Binary Tree to Double Linked List

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left  # Previous pointer for DLL
        self.right = right  # Next pointer for DLL

def tree_to_dll(root):
    if not root:
        return None

    # Initialize pointers for the DLL
    prev = None
    head = None
```

```python
  # Helper function to perform in-order traversal and convert   to DLL
    def inorder(node):
        nonlocal prev, head

        if not node:
            return

        # Recursively traverse left subtree
        inorder(node.left)

        # Adjust pointers for DLL
        if prev:
            prev.right = node
            node.left = prev
        else:
            head = node  # Set head of the DLL

        prev = node

        # Recursively traverse right subtree
        inorder(node.right)

    # Start in-order traversal from root
    inorder(root)

    # Return the head of the DLL
    return head

# Function to print the doubly linked list
def print_dll(head):
    current = head
    while current:
        print(current.val, end=" ")
        if current.right:
            print("<=> ", end="")
        current = current.right
    print()

# Example usage:
if __name__ == "__main__":
    # Create a sample binary search tree
    root = TreeNode(4)
    root.left = TreeNode(2)
    root.right = TreeNode(6)
    root.left.left = TreeNode(1)
    root.left.right = TreeNode(3)
    root.right.left = TreeNode(5)
    root.right.right = TreeNode(7)

    # Convert BST to DLL
    dll_head = tree_to_dll(root)

    # Print the doubly linked list
    print("Doubly Linked List:")
    print_dll(dll_head)
```

Find median in a stream of running integers

```python
class TreeNode:
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None
        self.left_count = 0  # Number of nodes in the left subtree (excluding the node itself)

class MedianFinder:
    def __init__(self):
        self.root = None
        self.size = 0

    def add_num(self, num):
        self.root = self._insert(self.root, num)
        self.size += 1

    def _insert(self, node, val):
        if not node:
            return TreeNode(val)

        if val <= node.val:
            node.left = self._insert(node.left, val)
            node.left_count += 1
        else:
            node.right = self._insert(node.right, val)

        return node

    def find_median(self):
        if self.size == 0:
            return None

        if self.size % 2 == 1:
            return self._find_kth_element((self.size // 2) + 1)
        else:
            left_median = self._find_kth_element(self.size // 2)
            right_median = self._find_kth_element((self.size // 2) + 1)
            return (left_median + right_median) / 2.0

    def _find_kth_element(self, k):
        current = self.root
        while current:
            left_size = current.left_count + 1  # Including the current node itself
            if k == left_size:
                return current.val
            elif k < left_size:
                current = current.left
            else:
                k -= left_size
                current = current.right

        return None

# Example usage:
if __name__ == "__main__":
    median_finder = MedianFinder()
    nums = [2, 1, 5, 7, 2, 0, 5]

    for num in nums:
        median_finder.add_num(num)
        print(f"After adding {num}, current median is: {median_finder.find_median()}")
```

K-th largest element in a stream.

```
class TreeNode:
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None
        self.right_size = 0  # Size of the right subtree (excluding the node itself)

class KthLargest:
    def __init__(self, k):
        self.root = None
        self.k = k

    def add(self, val):
        self.root = self._insert(self.root, val)

    def _insert(self, node, val):
        if not node:
            return TreeNode(val)

        if val <= node.val:
            node.left = self._insert(node.left, val)
        else:
            node.right = self._insert(node.right, val)
            node.right_size += 1

        return node

    def get_kth_largest(self):
        return self._find_kth_largest(self.root, self.k)

    def _find_kth_largest(self, node, k):
        if not node:
            return None

        # Calculate the position of the current node in the sorted order
        position = node.right_size + 1

        if position == k:
            return node.val
        elif position < k:
            return self._find_kth_largest(node.left, k - position)
        else:
            return self._find_kth_largest(node.right, k)

# Example usage:
if __name__ == "__main__":
    kth_largest = KthLargest(3)
    nums = [4, 5, 8, 2, 1, 6]

    for num in nums:
        kth_largest.add(num)
        print(f"After adding {num}, current {kth_largest.k}-th largest element is: 
{kth_largest.get_kth_largest()}")
```

Distinct numbers in Window.

```
from collections import defaultdict
```

```python
def count_distinct_in_window(nums, k):
    if k <= 0:
        return []

    n = len(nums)
    if k > n:
        return []

    result = []
    frequency = defaultdict(int)
    distinct_count = 0

    # Initialize the frequency map for the first window
    for i in range(k):
        frequency[nums[i]] += 1
        if frequency[nums[i]] == 1:
            distinct_count += 1

    result.append(distinct_count)

    # Slide the window across the array
    for i in range(k, n):
        leftmost = nums[i - k]
        rightmost = nums[i]

        # Update frequency for the new element entering the window
        frequency[rightmost] += 1
        if frequency[rightmost] == 1:
            distinct_count += 1

        # Remove frequency for the element leaving the window
        frequency[leftmost] -= 1
        if frequency[leftmost] == 0:
            distinct_count -= 1

        result.append(distinct_count)

    return result

# Example usage:
if __name__ == "__main__":
    nums = [1, 2, 1, 3, 4, 2, 3]
    k = 4
    result = count_distinct_in_window(nums, k)
    print(f"Distinct numbers in each window of size {k}: {result}")
```

K-th largest element in an unsorted array.

```python
import random

def find_kth_largest(nums, k):
    # Edge case handling
    if k < 1 or k > len(nums):
        return None

    # Quickselect algorithm to find the K-th largest element
    def quickselect(nums, left, right, k):
        if left == right:
            return nums[left]
```

```python
        # Partition the array and get the pivot index
        pivot_index = partition(nums, left, right)

        # Calculate the position of the pivot
        position = pivot_index - left + 1

        if position == k:
            return nums[pivot_index]
        elif position > k:
            return quickselect(nums, left, pivot_index - 1, k)
        else:
            return quickselect(nums, pivot_index + 1, right, k - position)

    # Helper function to partition the array around a pivot
    def partition(nums, left, right):
        # Randomly choose a pivot index and swap with the rightmost element
        pivot_index = random.randint(left, right)
        nums[right], nums[pivot_index] = nums[pivot_index], nums[right]
        pivot = nums[right]

        # Partition the array around the pivot
        i = left - 1
        for j in range(left, right):
            if nums[j] >= pivot:
                i += 1
                nums[i], nums[j] = nums[j], nums[i]

        # Move the pivot element to its correct position
        nums[i + 1], nums[right] = nums[right], nums[i + 1]

        return i + 1

    # Call quickselect to find the K-th largest element
    return quickselect(nums, 0, len(nums) - 1, k)

# Example usage:
if __name__ == "__main__":
    nums = [3, 2, 1, 5, 6, 4]
    k = 2
    kth_largest = find_kth_largest(nums, k)
    print(f"The {k}-th largest element in the array is: {kth_largest}")
```

Flood-fill Algorithm

```python
def flood_fill(matrix, sr, sc, new_color):
    if not matrix or not matrix[0]:
        return matrix

    rows, cols = len(matrix), len(matrix[0])
    original_color = matrix[sr][sc]

    def dfs(r, c):
        if r < 0 or r >= rows or c < 0 or c >= cols or matrix[r][c] != original_color or
matrix[r][c] == new_color:
            return

        matrix[r][c] = new_color

        dfs(r + 1, c)  # Down
        dfs(r - 1, c)  # Up
```

```python
        dfs(r, c + 1)  # Right
        dfs(r, c - 1)  # Left

    dfs(sr, sc)
    return matrix

# Example usage:
if __name__ == "__main__":
    matrix = [
        [1, 1, 1, 1, 0],
        [1, 1, 0, 1, 0],
        [1, 0, 0, 1, 1],
        [1, 1, 1, 0, 1]
    ]
    sr, sc = 1, 2  # Seed point
    new_color = 2  # New color to fill

    print("Original Matrix:")
    for row in matrix:
        print(row)

    flooded_matrix = flood_fill(matrix, sr, sc, new_color)

    print("\nMatrix after flood fill:")
    for row in flooded_matrix:
        print(row)
```

Clone a graph

```python
class GraphNode:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

def clone_graph(node):
    if not node:
        return None

    # Dictionary to save the visited nodes
    visited = {}

    def dfs(node):
        if node in visited:
            return visited[node]

        # Clone the node
        copy = GraphNode(node.value)
        visited[node] = copy

        # Clone all the neighbors
        for neighbor in node.neighbors:
            copy.neighbors.append(dfs(neighbor))

        return copy

    return dfs(node)

# Helper function to print the graph (for testing)
def print_graph(node, visited=set()):
    if node in visited:
```

```python
        return
    visited.add(node)
    print(f'Node {node.value} neighbors:', [n.value for n in node.neighbors])
    for neighbor in node.neighbors:
        print_graph(neighbor, visited)

# Example usage
if __name__ == "__main__":
    # Creating a sample graph
    node1 = GraphNode(1)
    node2 = GraphNode(2)
    node3 = GraphNode(3)
    node4 = GraphNode(4)

    node1.neighbors = [node2, node4]
    node2.neighbors = [node1, node3]
    node3.neighbors = [node2, node4]
    node4.neighbors = [node1, node3]

    print("Original graph:")
    print_graph(node1)

    cloned_graph = clone_graph(node1)

    print("\nCloned graph:")
    print_graph(cloned_graph)
```

DFS

```python
class GraphNode:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

def clone_graph(node):
    if not node:
        return None

    # Dictionary to save the visited nodes and their clones
    visited = {}

    def dfs(node):
        if node in visited:
            return visited[node]

        # Clone the node
        copy = GraphNode(node.value)
        visited[node] = copy

        # Clone all the neighbors using DFS
        for neighbor in node.neighbors:
            copy.neighbors.append(dfs(neighbor))

        return copy

    return dfs(node)

# Helper function to print the graph (for testing)
def print_graph(node, visited=set()):
    if node in visited:
```

```python
        return
    visited.add(node)
    print(f'Node {node.value} neighbors:', [n.value for n.neighbors])
    for neighbor in node.neighbors:
        print_graph(neighbor, visited)

# Example usage
if __name__ == "__main__":
    # Creating a sample graph
    node1 = GraphNode(1)
    node2 = GraphNode(2)
    node3 = GraphNode(3)
    node4 = GraphNode(4)

    node1.neighbors = [node2, node4]
    node2.neighbors = [node1, node3]
    node3.neighbors = [node2, node4]
    node4.neighbors = [node1, node3]

    print("Original graph:")
    print_graph(node1)

    cloned_graph = clone_graph(node1)

    print("\nCloned graph:")
    print_graph(cloned_graph)
```

BFS

```python
from collections import deque

class GraphNode:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

def clone_graph(node):
    if not node:
        return None

    # Dictionary to save the visited nodes and their clones
    visited = {}

    # Initialize the queue for BFS
    queue = deque([node])

    # Clone the root node
    visited[node] = GraphNode(node.value)

    while queue:
        current = queue.popleft()

        # Traverse all the neighbors of the current node
        for neighbor in current.neighbors:
            if neighbor not in visited:
                # Clone the neighbor and put it in the queue
                visited[neighbor] = GraphNode(neighbor.value)
                queue.append(neighbor)
            # Add the cloned neighbor to the current node's neighbors
            visited[current].neighbors.append(visited[neighbor])
```

```python
    return visited[node]

# Helper function to print the graph (for testing)
def print_graph(node, visited=set()):
    if node in visited:
        return
    visited.add(node)
    print(f'Node {node.value} neighbors:', [n.value for n in node.neighbors])
    for neighbor in node.neighbors:
        print_graph(neighbor, visited)

# Example usage
if __name__ == "__main__":
    # Creating a sample graph
    node1 = GraphNode(1)
    node2 = GraphNode(2)
    node3 = GraphNode(3)
    node4 = GraphNode(4)

    node1.neighbors = [node2, node4]
    node2.neighbors = [node1, node3]
    node3.neighbors = [node2, node4]
    node4.neighbors = [node1, node3]

    print("Original graph:")
    print_graph(node1)

    cloned_graph = clone_graph(node1)

    print("\nCloned graph:")
    print_graph(cloned_graph)
```

Detect A cycle in Undirected Graph using BFS

```python
from collections import deque, defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def has_cycle(self):
        visited = set()

        for node in self.graph:
            if node not in visited:
                if self.bfs_cycle_check(node, visited):
                    return True
        return False

    def bfs_cycle_check(self, start, visited):
        queue = deque([(start, -1)])  # (node, parent)
        visited.add(start)

        while queue:
            node, parent = queue.popleft()
```

```python
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, node))
                elif parent != neighbor:
                    return True
        return False

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(2, 3)
    g.add_edge(3, 4)
    g.add_edge(4, 5)
    g.add_edge(5, 6)
    g.add_edge(6, 3)  # Adding a cycle

    print("Cycle detected:" if g.has_cycle() else "No cycle detected")

    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 4)

    print("Cycle detected:" if g2.has_cycle() else "No cycle detected")
```

Detect A cycle in Undirected Graph using DFS

```python
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def has_cycle(self):
        visited = set()

        for node in self.graph:
            if node not in visited:
                if self.dfs_cycle_check(node, visited, -1):
                    return True
        return False

    def dfs_cycle_check(self, node, visited, parent):
        visited.add(node)

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if self.dfs_cycle_check(neighbor, visited, node):
                    return True
            elif neighbor != parent:
                return True
        return False
```

```python
# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(2, 3)
    g.add_edge(3, 4)
    g.add_edge(4, 5)
    g.add_edge(5, 6)
    g.add_edge(6, 3)  # Adding a cycle

    print("Cycle detected:" if g.has_cycle() else "No cycle detected")

    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 4)

    print("Cycle detected:" if g2.has_cycle() else "No cycle detected")
```

Detect A cycle in a Directed Graph using DFS

```python
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def has_cycle(self):
        visited = set()
        rec_stack = set()

        for node in self.graph:
            if node not in visited:
                if self.dfs_cycle_check(node, visited, rec_stack):
                    return True
        return False

    def dfs_cycle_check(self, node, visited, rec_stack):
        visited.add(node)
        rec_stack.add(node)

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if self.dfs_cycle_check(neighbor, visited, rec_stack):
                    return True
            elif neighbor in rec_stack:
                return True

        rec_stack.remove(node)
        return False

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(2, 3)
    g.add_edge(3, 4)
```

```python
    g.add_edge(4, 5)
    g.add_edge(5, 6)
    g.add_edge(6, 3)  # Adding a cycle

    print("Cycle detected:" if g.has_cycle() else "No cycle detected")

    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 4)

    print("Cycle detected:" if g2.has_cycle() else "No cycle detected")
```

Detect A cycle in a Directed Graph using BFS

```python
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
        self.in_degree = defaultdict(int)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.in_degree[v] += 1
        if u not in self.in_degree:
            self.in_degree[u] = 0

    def has_cycle(self):
        queue = deque()

        # Add all nodes with in-degree 0 to the queue
        for node in self.in_degree:
            if self.in_degree[node] == 0:
                queue.append(node)

        count = 0

        while queue:
            node = queue.popleft()
            count += 1

            for neighbor in self.graph[node]:
                self.in_degree[neighbor] -= 1
                if self.in_degree[neighbor] == 0:
                    queue.append(neighbor)

        # If count of visited nodes is not equal to the number of nodes, there is a cycle
        return count != len(self.in_degree)

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(2, 3)
    g.add_edge(3, 4)
    g.add_edge(4, 5)
    g.add_edge(5, 6)
    g.add_edge(6, 3)  # Adding a cycle
```

```
    print("Cycle detected:" if g.has_cycle() else "No cycle detected")

    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 4)

    print("Cycle detected:" if g2.has_cycle() else "No cycle detected")
```

Topological Sort BFS

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
        self.in_degree = defaultdict(int)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.in_degree[v] += 1
        if u not in self.in_degree:
            self.in_degree[u] = 0

    def topological_sort(self):
        queue = deque()

        # Add all nodes with in-degree 0 to the queue
        for node in self.in_degree:
            if self.in_degree[node] == 0:
                queue.append(node)

        topological_order = []

        while queue:
            node = queue.popleft()
            topological_order.append(node)

            for neighbor in self.graph[node]:
                self.in_degree[neighbor] -= 1
                if self.in_degree[neighbor] == 0:
                    queue.append(neighbor)

        # If topological_order does not include all nodes, there is a cycle
        if len(topological_order) != len(self.in_degree):
            return "Graph has a cycle, topological sort not possible"

        return topological_order

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(5, 2)
    g.add_edge(5, 0)
    g.add_edge(4, 0)
    g.add_edge(4, 1)
    g.add_edge(2, 3)
    g.add_edge(3, 1)

    print("Topological Sort:", g.topological_sort())
```

```python
    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 1)  # Adding a cycle

    print("Topological Sort:", g2.topological_sort())
```

Topological Sort DFS

```python
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def topological_sort(self):
        visited = set()
        stack = []

        for node in self.graph:
            if node not in visited:
                self.dfs(node, visited, stack)

        return stack[::-1]  # Return the reverse of the completed stack

    def dfs(self, node, visited, stack):
        visited.add(node)

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self.dfs(neighbor, visited, stack)

        stack.append(node)

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(5, 2)
    g.add_edge(5, 0)
    g.add_edge(4, 0)
    g.add_edge(4, 1)
    g.add_edge(2, 3)
    g.add_edge(3, 1)

    print("Topological Sort:", g.topological_sort())

    g2 = Graph()
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 4)
    g2.add_edge(4, 5)

    print("Topological Sort:", g2.topological_sort())
```

Number of islands(Do in Grid and Graph Both)

```
def num_islands(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    visited = [[False] * cols for _ in range(rows)]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    num_islands = 0

    def dfs(row, col):
        stack = [(row, col)]
        while stack:
            r, c = stack.pop()
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and not visited[nr][nc] and grid[nr][nc]
== '1':
                    visited[nr][nc] = True
                    stack.append((nr, nc))

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '1' and not visited[i][j]:
                num_islands += 1
                visited[i][j] = True
                dfs(i, j)

    return num_islands

# Example usage
if __name__ == "__main__":
    grid1 = [
        ["1","1","1","1","0"],
        ["1","1","0","1","0"],
        ["1","1","0","0","0"],
        ["0","0","0","0","0"]
    ]
    print("Number of islands in grid1:", num_islands(grid1))

    grid2 = [
        ["1","1","0","0","0"],
        ["1","1","0","0","0"],
        ["0","0","1","0","0"],
        ["0","0","0","1","1"]
    ]
    print("Number of islands in grid2:", num_islands(grid2))
```

Bipartite Check using BFS

```
from collections import deque, defaultdict

def is_bipartite(graph):
    if not graph:
        return True

    colors = {}  # Dictionary to store color of each node
    queue = deque()
```

```python
    # Iterate over each node (in case the graph is disconnected)
    for node in graph:
        if node not in colors:  # Not colored yet
            queue.append(node)
            colors[node] = 0  # Color the first node with color 0

            while queue:
                current = queue.popleft()
                current_color = colors[current]

                for neighbor in graph[current]:
                    if neighbor not in colors:  # Not colored yet
                        colors[neighbor] = 1 - current_color  # Assign opposite color
                        queue.append(neighbor)
                    elif colors[neighbor] == current_color:
                        return False  # Found a conflict, not bipartite

    return True

# Example usage
if __name__ == "__main__":
    graph1 = {
        0: [1, 3],
        1: [0, 2],
        2: [1, 3],
        3: [0, 2]
    }
    print("Graph1 is bipartite:", is_bipartite(graph1))

    graph2 = {
        0: [1, 2, 3],
        1: [0, 2],
        2: [1, 0],
        3: [0]
    }
    print("Graph2 is bipartite:", is_bipartite(graph2))
```

Bipartite Check using DFS

```python
def is_bipartite(graph):
    if not graph:
        return True

    colors = {}  # Dictionary to store color of each node

    def dfs(node, color):
        colors[node] = color
        for neighbor in graph[node]:
            if neighbor not in colors:
                if not dfs(neighbor, 1 - color):  # Color the neighbor with opposite color
                    return False
            elif colors[neighbor] == color:
                return False
        return True

    # Iterate over each node (in case the graph is disconnected)
    for node in graph:
        if node not in colors:  # Not colored yet
            if not dfs(node, 0):  # Start coloring with color 0
                return False
```

```python
    return True

# Example usage
if __name__ == "__main__":
    graph1 = {
        0: [1, 3],
        1: [0, 2],
        2: [1, 3],
        3: [0, 2]
    }
    print("Graph1 is bipartite:", is_bipartite(graph1))

    graph2 = {
        0: [1, 2, 3],
        1: [0, 2],
        2: [1, 0],
        3: [0]
    }
    print("Graph2 is bipartite:", is_bipartite(graph2))
```

Strongly Connected Component(using KosaRajuΓÇÖs algo)

```python
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
        self.vertices = set()

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.vertices.add(u)
        self.vertices.add(v)

    def dfs(self, node, visited, stack):
        visited.add(node)
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self.dfs(neighbor, visited, stack)
        stack.append(node)

    def transpose(self):
        transposed_graph = defaultdict(list)
        for u in self.graph:
            for v in self.graph[u]:
                transposed_graph[v].append(u)
        return transposed_graph

    def dfs_scc(self, node, visited, result):
        visited.add(node)
        result.append(node)
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self.dfs_scc(neighbor, visited, result)

    def kosaraju_scc(self):
        stack = []
        visited = set()
```

```
        # Step 1: Perform DFS and fill the stack based on finishing times
        for node in self.vertices:
            if node not in visited:
                self.dfs(node, visited, stack)

        # Step 2: Transpose the graph
        transposed_graph = self.transpose()

        # Step 3: Perform DFS on the transposed graph in the order of stack
        visited.clear()
        scc_list = []
        while stack:
            node = stack.pop()
            if node not in visited:
                scc = []
                self.dfs_scc(node, visited, scc)
                scc_list.append(scc)

        return scc_list

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(1, 3)
    g.add_edge(3, 4)
    g.add_edge(4, 5)
    g.add_edge(5, 3)
    g.add_edge(6, 5)
    g.add_edge(6, 7)
    g.add_edge(7, 8)
    g.add_edge(8, 6)

    sccs = g.kosaraju_scc()
    print("Strongly Connected Components:")
    for scc in sccs:
        print(scc)
```

DijkstraΓÇÖs Algorithm

```
import heapq
from collections import defaultdict

def dijkstra(graph, start):
    # Initialize distances from the start node to all other nodes as infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0  # Distance from start to itself is 0

    # Priority queue to store nodes to be processed: (distance, node)
    pq = [(0, start)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        # Skip processing if current distance is greater than recorded distance
        if current_distance > distances[current_node]:
            continue
```

```python
        # Traverse neighbors of current node
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If found shorter path to neighbor, update distance and push to queue
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

# Example usage
if __name__ == "__main__":
    # Example graph as adjacency list with weighted edges
    graph = {
        'A': {'B': 3, 'C': 6},
        'B': {'A': 3, 'C': 2, 'D': 1},
        'C': {'A': 6, 'B': 2, 'D': 1},
        'D': {'B': 1, 'C': 1}
    }

    start_node = 'A'
    shortest_distances = dijkstra(graph, start_node)

    print("Shortest distances from node", start_node, ":")
    for node, distance in shortest_distances.items():
        print(f"To node {node}: Distance {distance}")
```

Bellman-Ford Algo

```python
import sys

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def bellman_ford(self, src):
        # Step 1: Initialize distances from src to all other vertices as INFINITE
        distances = [float("Inf")] * self.V
        distances[src] = 0

        # Step 2: Relax all edges |V| - 1 times.
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if distances[u] != float("Inf") and distances[u] + w < distances[v]:
                    distances[v] = distances[u] + w

        # Step 3: Check for negative-weight cycles.
        # The above step guarantees shortest distances if graph doesn't contain negative
weight cycle.
        # If we get a shorter path, then there is a cycle.
        for u, v, w in self.graph:
            if distances[u] != float("Inf") and distances[u] + w < distances[v]:
                print("Graph contains negative weight cycle")
                return
```

```python
        self.print_solution(distances)

    def print_solution(self, distances):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print(f"{i}\t\t{distances[i]}")

# Example usage
if __name__ == "__main__":
    g = Graph(5)
    g.add_edge(0, 1, -1)
    g.add_edge(0, 2, 4)
    g.add_edge(1, 2, 3)
    g.add_edge(1, 3, 2)
    g.add_edge(1, 4, 2)
    g.add_edge(3, 2, 5)
    g.add_edge(3, 1, 1)
    g.add_edge(4, 3, -3)

    # Print the solution
    g.bellman_ford(0)
```

Floyd Warshall Algorithm

```python
INF = float('inf')

def floyd_warshall(graph):
    # Initialize distance matrix with infinities and 0s on diagonal
    dist = [[INF if i != j else 0 for j in range(len(graph))] for i in range(len(graph))]

    # Fill distance matrix with initial weights from the graph
    for u in range(len(graph)):
        for v, weight in graph[u].items():
            dist[u][v] = weight

    # Compute shortest paths
    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

if __name__ == "__main__":
    # Example graph as adjacency list with weighted edges
    graph = {
        0: {1: 3, 2: 8, 4: -4},
        1: {3: 1, 4: 7},
        2: {1: 4},
        3: {0: 2, 2: -5},
        4: {3: 6}
    }

    shortest_paths = floyd_warshall(graph)

    print("Shortest paths between all pairs of vertices:")
    for row in shortest_paths:
        print(row)
```

MST using Prim's Algo

```
import heapq
from collections import defaultdict

def prim_mst(graph):
    min_heap = []
    heapq.heapify(min_heap)

    # Start with any arbitrary node as the root of MST
    start_node = next(iter(graph))
    mst = []
    visited = set()

    # Push edges from the start node to the min-heap
    push_edges(start_node, graph, min_heap, visited)

    while min_heap:
        weight, u, v = heapq.heappop(min_heap)

        if v not in visited:
            visited.add(v)
            mst.append((u, v, weight))
            push_edges(v, graph, min_heap, visited)

    return mst

def push_edges(node, graph, min_heap, visited):
    for neighbor, weight in graph[node].items():
        if neighbor not in visited:
            heapq.heappush(min_heap, (weight, node, neighbor))

# Example usage
if __name__ == "__main__":
    # Example graph as adjacency list with weighted edges
    graph = {
        'A': {'B': 2, 'D': 3},
        'B': {'A': 2, 'D': 5, 'C': 2},
        'C': {'B': 2, 'D': 3, 'E': 1},
        'D': {'A': 3, 'B': 5, 'C': 3, 'E': 1},
        'E': {'C': 1, 'D': 1}
    }

    mst = prim_mst(graph)

    # Print the MST
    print("Minimum Spanning Tree (MST):")
    for u, v, weight in mst:
        print(f"Edge: {u}-{v}, Weight: {weight}")
```

MST using KruskalΓÇÖs Algo

```
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
```

```python
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            # Union by rank
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False

def kruskal_mst(graph):
    edges = []
    for u in graph:
        for v, weight in graph[u].items():
            edges.append((weight, u, v))

    # Sort edges by weight
    edges.sort()

    n = len(graph)
    dsu = DisjointSetUnion(n)
    mst = []

    for weight, u, v in edges:
        if dsu.union(u, v):
            mst.append((u, v, weight))
            # Stop when MST has n-1 edges
            if len(mst) == n - 1:
                break

    return mst

# Example usage
if __name__ == "__main__":
    # Example graph as adjacency list with weighted edges
    graph = {
        'A': {'B': 2, 'D': 3},
        'B': {'A': 2, 'D': 5, 'C': 2},
        'C': {'B': 2, 'D': 3, 'E': 1},
        'D': {'A': 3, 'B': 5, 'C': 3, 'E': 1},
        'E': {'C': 1, 'D': 1}
    }

    mst = kruskal_mst(graph)

    # Print the MST
    print("Minimum Spanning Tree (MST) using Kruskal's Algorithm:")
    for u, v, weight in mst:
        print(f"Edge: {u}-{v}, Weight: {weight}")
```

Max Product Subarray

```python
def max_product_subarray(nums):
    if not nums:
        return 0

    n = len(nums)
    if n == 1:
        return nums[0]

    max_product = float('-inf')
    current_max = 1
    current_min = 1

    for num in nums:
        if num > 0:
            current_max = max(current_max * num, num)
            current_min = min(current_min * num, num)
        elif num == 0:
            current_max = 1
            current_min = 1
        else:  # num < 0
            temp = current_max
            current_max = max(current_min * num, num)
            current_min = min(temp * num, num)

        max_product = max(max_product, current_max)

    return max_product
```

Longest Increasing Subsequence

```python
def length_of_lis(nums):
    if not nums:
        return 0

    n = len(nums)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Example usage:
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis(nums))  # Output: 4 (the LIS is [2, 3, 7, 101])
```

Longest Common Subsequence

```python
def longest_common_subsequence(text1, text2):
    m = len(text1)
    n = len(text2)

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
```

```
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage:
text1 = "abcde"
text2 = "ace"
print(longest_common_subsequence(text1, text2))  # Output: 3 (LCS is "ace")
```

0-1 Knapsack

```
def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] > w:
                dp[i][w] = dp[i - 1][w]
            else:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])

    return dp[n][capacity]

# Example usage:
weights = [1, 2, 3]
values = [6, 10, 12]
capacity = 5
print(knapsack_01(weights, values, capacity))  # Output: 22 (select items with weights 2 and 3
for max value)
```

Edit Distance

```
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize base cases
    for i in range(1, m + 1):
        dp[i][0] = i
    for j in range(1, n + 1):
        dp[0][j] = j

    # Build DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j],     # deletion from word1
                                   dp[i][j - 1],     # insertion to word1
                                   dp[i - 1][j - 1]) # substitution in word1

    return dp[m][n]
```

```
  # Example usage:
  word1 = "horse"
  word2 = "ros"
  print(min_distance(word1, word2))  # Output: 3 (replace 'h' with 'r', delete 'o', delete 'e'
to match "ros")
```

Maximum sum increasing subsequence

```
  def max_sum_increasing_subsequence(nums):
      if not nums:
          return 0

      n = len(nums)
      dp = nums[:]  # Initialize dp array with the same values as nums

      for i in range(1, n):
          for j in range(i):
              if nums[j] < nums[i]:
                  dp[i] = max(dp[i], dp[j] + nums[i])

      return max(dp)

  # Example usage:
  nums = [1, 101, 2, 3, 100, 4, 5]
  print(max_sum_increasing_subsequence(nums))  # Output: 106 (maximum sum increasing subsequence
is [1, 2, 3, 100])
```

Matrix Chain Multiplication

```
  def matrix_chain_order(dims):
      n = len(dims) - 1  # Number of matrices
      dp = [[0] * n for _ in range(n)]

      for length in range(2, n + 1):  # Length of chain
          for i in range(1, n - length + 2):
              j = i + length - 1
              dp[i - 1][j - 1] = float('inf')
              for k in range(i, j):
                  q = dp[i - 1][k - 1] + dp[k][j - 1] + dims[i - 1] * dims[k] * dims[j]
                  if q < dp[i - 1][j - 1]:
                      dp[i - 1][j - 1] = q

      return dp[0][n - 1]

  # Example usage:
  dims = [30, 35, 15, 5, 10, 20, 25]
  print(matrix_chain_order(dims))  # Output: 15125 (minimum scalar multiplications for optimal
multiplication order)
```

Minimum sum path in the matrix, (count paths and similar type do, also backtrack to find the Minimum path)

```
  def min_path_sum(grid):
      if not grid:
          return 0
```

```python
    m = len(grid)
    n = len(grid[0])

    # Initialize dp array and count array
    dp = [[float('inf')] * n for _ in range(m)]
    count = [[0] * n for _ in range(m)]

    # Base case
    dp[0][0] = grid[0][0]
    count[0][0] = 1

    # Fill dp array
    for i in range(m):
        for j in range(n):
            if i == 0 and j == 0:
                continue
            if i > 0:
                if dp[i-1][j] + grid[i][j] < dp[i][j]:
                    dp[i][j] = dp[i-1][j] + grid[i][j]
                    count[i][j] = count[i-1][j]
                elif dp[i-1][j] + grid[i][j] == dp[i][j]:
                    count[i][j] += count[i-1][j]
            if j > 0:
                if dp[i][j-1] + grid[i][j] < dp[i][j]:
                    dp[i][j] = dp[i][j-1] + grid[i][j]
                    count[i][j] = count[i][j-1]
                elif dp[i][j-1] + grid[i][j] == dp[i][j]:
                    count[i][j] += count[i][j-1]

    # Minimum sum path value
    min_sum = dp[m-1][n-1]

    # Backtrack to find at least one path achieving the minimum sum
    path = []
    i, j = m-1, n-1
    while i >= 0 and j >= 0:
        path.append((i, j))
        if i == 0 and j == 0:
            break
        if i > 0 and dp[i-1][j] + grid[i][j] == dp[i][j]:
            i -= 1
        elif j > 0 and dp[i][j-1] + grid[i][j] == dp[i][j]:
            j -= 1

    path.reverse()  # Reverse to get the path from (0, 0) to (m-1, n-1)

    return min_sum, count[m-1][n-1], path

# Example usage:
grid = [
    [1, 3, 1],
    [1, 5, 1],
    [4, 2, 1]
]

min_sum, num_paths, min_path = min_path_sum(grid)
print("Minimum sum path:", min_sum)
print("Number of paths with minimum sum:", num_paths)
print("One of the paths with minimum sum:", min_path)
```

Coin change

```
def coin_change(coins, amount):
    # Initialize dp array with infinity (amount + 1 means impossible amount initially)
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 0 coins needed to make amount 0

    # Build dp array
    for amt in range(1, amount + 1):
        for coin in coins:
            if coin <= amt:
                dp[amt] = min(dp[amt], dp[amt - coin] + 1)

    # Check if it's possible to make up the amount
    if dp[amount] == float('inf'):
        return -1
    else:
        return dp[amount]

# Example usage:
coins = [1, 2, 5]
amount = 11
print(coin_change(coins, amount))  # Output: 3 (coins used: 5 + 5 + 1 = 11)
```

Subset Sum

```
def subset_sum(nums, target):
    n = len(nums)
    dp = [[False] * (target + 1) for _ in range(n + 1)]

    # Base case
    for i in range(n + 1):
        dp[i][0] = True

    # Build dp table
    for i in range(1, n + 1):
        for j in range(1, target + 1):
            if nums[i - 1] > j:
                dp[i][j] = dp[i - 1][j]
            else:
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i - 1]]

    return dp[n][target]

# Example usage:
nums = [2, 3, 7, 8, 10]
target = 11
print(subset_sum(nums, target))  # Output: True (subset [3, 8] sums up to 11)
```

Rod Cutting

```
def rod_cutting(prices, n):
    dp = [0] * (n + 1)

    for i in range(1, n + 1):
        max_profit = float('-inf')
        for j in range(1, i + 1):
            max_profit = max(max_profit, prices[j - 1] + dp[i - j])
```

```
        dp[i] = max_profit

    return dp[n]

# Example usage:
prices = [1, 5, 8, 9, 10, 17, 17, 20]
n = 8
print("Maximum profit:", rod_cutting(prices, n))  # Output: 22 (cut the rod into pieces of length 2 and 6)
```

Egg Dropping

```
def egg_drop(k, n):
    dp = [[0] * (n + 1) for _ in range(k + 1)]

    for j in range(1, n + 1):
        dp[1][j] = j

    for i in range(2, k + 1):
        for j in range(1, n + 1):
            dp[i][j] = float('inf')
            for x in range(1, j + 1):
                dp[i][j] = min(dp[i][j], max(dp[i - 1][x - 1], dp[i][j - x]) + 1)

    return dp[k][n]

# Example usage:
k = 2  # number of eggs
n = 10  # number of floors
print("Minimum number of drops:", egg_drop(k, n))  # Output: 4
```

Word Break

```
def wordBreak(s, wordDict):
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True # Empty string is always true

    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break

    return dp[n]
```

Palindrome Partitioning (MCM Variation)

```
def minCut(s):
    n = len(s)
    dp = [float('inf')] * n
    isPalindrome = [[False] * n for _ in range(n)]

    for i in range(n):
        for j in range(i + 1):
```

```
            if s[j] == s[i] and (i - j <= 2 or isPalindrome[j + 1][i - 1]):
                isPalindrome[j][i] = True

    for i in range(n):
        if isPalindrome[0][i]:
            dp[i] = 0
        else:
            for j in range(i):
                if isPalindrome[j + 1][i]:
                    dp[i] = min(dp[i], dp[j] + 1)

    return dp[n - 1]
```

Maximum profit in Job scheduling

```
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    n = len(jobs)
    dp = [0] * n
    dp[0] = jobs[0][2]  # Initialize with the profit of the first job

    for i in range(1, n):
        start_i, end_i, profit_i = jobs[i]
        # Binary search to find the latest job that doesn't overlap with current job
        low, high = 0, i - 1
        while low <= high:
            mid = (low + high) // 2
            if jobs[mid][1] <= start_i:
                low = mid + 1
            else:
                high = mid - 1
        p = high  # p is the largest index < i such that jobs[p] doesn't overlap with jobs[i]

        # Choose the maximum profit between including current job or excluding it
        include_profit = profit_i + (dp[p] if p >= 0 else 0)
        dp[i] = max(dp[i - 1], include_profit)

    return dp[n - 1]

# Example usage:
startTime = [1, 2, 4, 6, 5]
endTime = [3, 5, 6, 7, 8]
profit = [5, 6, 5, 8, 11]

max_profit = jobScheduling(startTime, endTime, profit)
print("Maximum profit:", max_profit)
```

Implement Trie (Prefix Tree)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()
```

```python
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def startsWith(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

trie = Trie()

trie.insert("apple")
print(trie.search("apple"))   # Output: True
print(trie.search("app"))     # Output: False
print(trie.startsWith("app")) # Output: True

trie.insert("app")
print(trie.search("app"))     # Output: True
```

Longest String with All Prefixes

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def longest_string_with_all_prefixes(self):
        return self._dfs(self.root, "")

    def _dfs(self, node, path):
        if not node.is_end_of_word:
            return ""
```

```python
        longest_string = path
        for char, child in node.children.items():
            candidate = self._dfs(child, path + char)
            if len(candidate) > len(longest_string):
                longest_string = candidate

        return longest_string

# Example usage
words = ["a", "ap", "app", "appl", "apple", "appla", "apples"]
trie = Trie()
for word in words:
    trie.insert(word)

print(trie.longest_string_with_all_prefixes())  # Output: "apple"
```

Number of Distinct Substrings in a String

```python
class TrieNode:
    def __init__(self):
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()
        self.num_nodes = 0

    def insert_suffix(self, suffix):
        node = self.root
        for char in suffix:
            if char not in node.children:
                node.children[char] = TrieNode()
                self.num_nodes += 1
            node = node.children[char]

def count_distinct_substrings(s):
    trie = Trie()
    for i in range(len(s)):
        trie.insert_suffix(s[i:])
    # The number of distinct substrings is the number of nodes in the trie
    return trie.num_nodes

# Example usage
s = "ababa"
print(count_distinct_substrings(s))  # Output: 10
```

Power Set (this is very important)

```python
def power_set(s):
    # Start with the empty set
    result = [[]]

    for element in s:
        # For each element in the input set, add it to all existing subsets in the result
        result.extend([subset + [element] for subset in result])

    return result
```

```python
# Example usage
s = [1, 2, 3]
print(power_set(s))
def power_set(s):
    # Start with the empty set
    result = [[]]

    for element in s:
        # For each element in the input set, add it to all existing subsets in the result
        result.extend([subset + [element] for subset in result])

    return result

# Example usage
s = [1, 2, 3]
print(power_set(s))
```

Maximum XOR of two numbers in an array

```python
class TrieNode:
    def __init__(self):
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, number):
        node = self.root
        for i in range(31, -1, -1):  # 31 to 0 for 32-bit integer representation
            bit = (number >> i) & 1
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

    def find_max_xor(self, number):
        node = self.root
        max_xor = 0
        for i in range(31, -1, -1):  # 31 to 0 for 32-bit integer representation
            bit = (number >> i) & 1
            # We want to go to the opposite bit if possible to maximize XOR
            opposite_bit = 1 - bit
            if opposite_bit in node.children:
                max_xor = (max_xor << 1) | 1
                node = node.children[opposite_bit]
            else:
                max_xor = (max_xor << 1)
                node = node.children[bit]
        return max_xor

def find_maximum_xor(nums):
    trie = Trie()
    max_xor = 0
    for num in nums:
        trie.insert(num)
    for num in nums:
        max_xor = max(max_xor, trie.find_max_xor(num))
    return max_xor
```

```python
# Example usage
nums = [3, 10, 5, 25, 2, 8]
print(find_maximum_xor(nums))  # Output: 28
```

Maximum XOR With an Element From Array

```python
class TrieNode:
    def __init__(self):
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, number):
        node = self.root
        for i in range(31, -1, -1):  # 31 to 0 for 32-bit integer representation
            bit = (number >> i) & 1
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

    def find_max_xor(self, number):
        node = self.root
        max_xor = 0
        for i in range(31, -1, -1):  # 31 to 0 for 32-bit integer representation
            bit = (number >> i) & 1
            # We want to go to the opposite bit if possible to maximize XOR
            opposite_bit = 1 - bit
            if opposite_bit in node.children:
                max_xor = (max_xor << 1) | 1
                node = node.children[opposite_bit]
            else:
                max_xor = (max_xor << 1)
                node = node.children[bit]
        return max_xor

def find_maximum_xor_with_element(nums, x):
    trie = Trie()
    for num in nums:
        trie.insert(num)
    return trie.find_max_xor(x)

# Example usage
nums = [3, 10, 5, 25, 2, 8]
x = 5
print(find_maximum_xor_with_element(nums, x))  # Output: 28
```