Next Permutation

```python
def next_permutation(nums):
    """
    Modifies nums in-place to the next permutation.
    """
    # Find the first decreasing element from the end
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1

    if i >= 0:
        # Find the element just larger than nums[i] to the right of nums[i]
        j = len(nums) - 1
        while nums[j] <= nums[i]:
            j -= 1
        # Swap elements at i and j
        nums[i, nums[j] = nums[j], nums[i]

    # Reverse the elements to the right of i
    nums[i + 1:] = reversed(nums[i + 1:])

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

3-Sum Problem

```python
 def three_sum(nums):
     """
     Given an array nums of n integers, find all unique triplets in the array
which gives the sum of zero.

     :param nums: List[int]
     :return: List[List[int]]
     """
     nums.sort()
     result = []

     for i in range(len(nums) - 2):
         if i > 0 and nums[i] == nums[i - 1]:
             continue  # Skip duplicate triplets
         left, right = i + 1, len(nums) - 1
         while left < right:
             total = nums[i] + nums[left] + nums[right]
             if total == 0:
                 result.append([nums[i], nums[left], nums[right]])
```

```python
                while left < right and nums[left] == nums[left + 1]:
                    left += 1  # Skip duplicates
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1  # Skip duplicates
                left += 1
                right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1

    return result

# Example usage:
nums = [-1, 0, 1, 2, -1, -4]
print(three_sum(nums))  # Output: [[-1, -1, 2], [-1, 0, 1]]
```

Kadane's Algorithm

```python
def max_subarray_sum(nums):
    """
    Finds the maximum sum of a contiguous subarray using Kadane's Algorithm.

    :param nums: List[int]
    :return: int
    """
    max_current = max_global = nums[0]

    for num in nums[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current

    return max_global

# Example usage:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(nums))  # Output: 6 (subarray [4, -1, 2, 1])
```

Majority Element (n/3 times)

```python
def majority_element_n3(nums):
    """
    Finds all elements that appear more than n/3 times in the array.

    :param nums: List[int]
```

```python
    :return: List[int]
    """
    if not nums:
        return []

    # Step 1: Identify potential candidates
    candidate1, candidate2, count1, count2 = None, None, 0, 0

    for num in nums:
        if candidate1 == num:
            count1 += 1
        elif candidate2 == num:
            count2 += 1
        elif count1 == 0:
            candidate1, count1 = num, 1
        elif count2 == 0:
            candidate2, count2 = num, 1
        else:
            count1 -= 1
            count2 -= 1

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

    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(majority_element_n3(nums))  # Output: [3]

nums = [1, 1, 1, 3, 3, 2, 2, 2]
print(majority_element_n3(nums))  # Output: [1, 2]
```

Count number of subarrays with given xor K

```
def count_subarrays_with_xor(nums, K):
    """
    Counts the number of subarrays with a given XOR K.

    :param nums: List[int]
    :param K: int
    :return: int
    """
    from collections import defaultdict

    prefix_xor_count = defaultdict(int)
    prefix_xor_count[0] = 1  # To handle the case when prefix_xor itself is
equal to K
    prefix_xor = 0
    count = 0

    for num in nums:
        prefix_xor ^= num

        # If (prefix_xor ^ K) exists in the map, add its frequency to the
count
        target_xor = prefix_xor ^ K
        if target_xor in prefix_xor_count:
            count += prefix_xor_count[target_xor]

        # Increment the frequency of the current prefix_xor in the map
        prefix_xor_count[prefix_xor] += 1

    return count

# Example usage:
nums = [4, 2, 2, 6, 4]
K = 6
print(count_subarrays_with_xor(nums, K))  # Output: 4

nums = [5, 6, 7, 8, 9]
K = 5
print(count_subarrays_with_xor(nums, K))  # Output: 2
```

Find the repeating and missing number

```
def find_missing_and_repeating(nums):
    """
    Finds the missing and repeating numbers in the array.

    :param nums: List[int]
```

```
        :return: Tuple[int, int] (repeating, missing)
        """
        n = len(nums)
        sum_n = n * (n + 1) // 2
        sum_n_sq = n * (n + 1) * (2 * n + 1) // 6

        sum_nums = sum(nums)
        sum_nums_sq = sum(x * x for x in nums)

        # The difference between sum_n and sum_nums will give (missing -
repeating)
        sum_diff = sum_n - sum_nums  # (missing - repeating)

        # The difference between sum_n_sq and sum_nums_sq will give (missing^2 -
repeating^2)
        sum_sq_diff = sum_n_sq - sum_nums_sq  # (missing^2 - repeating^2)

        # Now we can solve the two equations:
        # missing - repeating = sum_diff
        # missing^2 - repeating^2 = sum_sq_diff

        # sum_sq_diff = (missing + repeating) * (missing - repeating)
        # sum_sq_diff = (missing + repeating) * sum_diff

        # Therefore:
        missing_plus_repeating = sum_sq_diff // sum_diff

        # Now we have two equations:
        # missing - repeating = sum_diff
        # missing + repeating = missing_plus_repeating

        # Solving these equations will give us the values of missing and
repeating:
        missing = (sum_diff + missing_plus_repeating) // 2
        repeating = missing - sum_diff

        return repeating, missing

    # Example usage:
    nums = [4, 3, 6, 2, 1, 1]
    print(find_missing_and_repeating(nums))  # Output: (1, 5)

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

Count Inversions

```python
def merge_and_count(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:
            temp_arr[k] = arr[j]
            inv_count += (mid-i + 1) # There are mid - i inversions, because
all elements left to i in the left subarray are greater than arr[j]
            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:
        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_and_count(arr, temp_arr, left, mid, right)

    return inv_count
```

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

# Example usage:
arr = [1, 20, 6, 4, 5]
print(count_inversions(arr))  # Output: 5

arr = [8, 4, 2, 1]
print(count_inversions(arr))  # Output: 6

arr = [3, 1, 2]
print(count_inversions(arr))  # Output: 2
```

Maximum Product Subarray

```python
def max_product_subarray(nums):
    """
    Finds the maximum product of a subarray.

    :param nums: List[int]
    :return: int
    """
    if not nums:
        return 0

    max_product = min_product = result = nums[0]

    for num in nums[1:]:
        if num < 0:
            max_product, min_product = min_product, max_product

        max_product = max(num, max_product * num)
        min_product = min(num, min_product * num)

        result = max(result, max_product)
    return result

nums = [2, 3, -2, 4]
print(max_product_subarray(nums))  # Output: 6

nums = [-2, 0, -1]
print(max_product_subarray(nums))  # Output: 0

nums = [-2, 3, -4]
print(max_product_subarray(nums))  # Output: 24
```

Search in Rotated Sorted Array II

```python
def search(nums, target):
    if not nums:
        return False

    start, end = 0, nums.length - 1

    while start <= end:
        mid = start + (end - start) // 2

        if nums[mid] == target:
            return True

        if nums[start] == nums[mid] == nums[end]:
            start += 1
            end -= 1
        elif nums[start] <= nums[mid]:
            if nums[start] <= target < nums[mid]:
                end = mid - 1
            else:
                start = mid + 1
        else:
            if nums[mid] < target <= nums[end]:
                start = mid + 1
            else:
                end = mid - 1

    return False
```

Find minimum in Rotated Sorted Array

```python
def find_min(nums):
    if not nums:
        return None

    start, end = 0, len(nums) - 1

    while start < end:
        mid = start + (end - start) // 2

        if nums[mid] > nums[end]:
            start = mid + 1
        elif nums[mid] < nums[end]:
            end = mid
        else:
            end -= 1
```

```
    return nums[start]

# Example usage:
rotated_array = [4, 5, 6, 7, 0, 1, 2]
min_element = find_min(rotated_array)
print(f"The minimum element in the rotated sorted array {rotated_array} is:
{min_element}")
```

Find peak element

```
def find_peak_element(nums):
    if not nums:
        return None

    start, end = 0, len(nums) - 1

    while start <= end:
        mid = start + (end - start) // 2

        # Check if mid is a peak
        if (mid == 0 or nums[mid] > nums[mid - 1]) and (mid == len(nums) - 1
or nums[mid] > nums[mid + 1]):
            return mid
        elif mid > 0 and nums[mid - 1] > nums[mid]:
            end = mid - 1
        else:
            start = mid + 1

    return -1  # If no peak is found (though problem states there's at least
one peak)

# Example usage:
nums = [1, 2, 3, 1]
peak_index = find_peak_element(nums)
print(f"The peak element in the array {nums} is at index: {peak_index}, with
value: {nums[peak_index]}")
```

Koko Eating Bananas

```
def min_eating_speed(piles, k):
    # Binary search for the minimum eating speed
    left, right = 1, max(piles)

    while left <= right:
        mid = left + (right - left) // 2
```

```
        if can_eat_all(piles, mid, k):
            right = mid - 1
        else:
            left = mid + 1

    return left

def can_eat_all(piles, speed, k):
    hours = 0
    for bananas in piles:
        hours += (bananas + speed - 1) // speed  # Ceiling division to
calculate hours needed

    return hours <= k

# Example usage:
piles = [3, 6, 7, 11]
k = 8
min_speed = min_eating_speed(piles, k)
print(f"The minimum eating speed required for Koko to eat all bananas in
{piles} within {k} hours is: {min_speed}")
```

Aggressive Cows

```
def can_place_cows(positions, C, min_distance):
    count = 1  # Place the first cow in the first position
    last_position = positions[0]

    for i in range(1, len(positions)):
        if positions[i] - last_position >= min_distance:
            count += 1
            last_position = positions[i]
            if count == C:
                return True

    return False

def aggressive_cows(N, C, positions):
    positions.sort()

    left, right = 1, positions[-1] - positions[0]
    max_min_distance = 0

    while left <= right:
        mid = left + (right - left) // 2

        if can_place_cows(positions, C, mid):
```

```
            max_min_distance = mid  # Found a valid distance, try for a
larger one
            left = mid + 1
        else:
            right = mid - 1

    return max_min_distance

# Example usage:
N = 5
C = 3
positions = [1, 2, 8, 4, 9]
max_min_distance = aggressive_cows(N, C, positions)
print(f"The maximum possible minimum distance between any two cows is:
{max_min_distance}")
```

Book Allocation Problem

```
def can_allocate_books(books, M, max_pages):
    students = 1
    current_pages = 0

    for pages in books:
        if current_pages + pages > max_pages:
            students += 1
            current_pages = pages
            if students > M:
                return False
        else:
            current_pages += pages

    return True

def min_max_pages(books, M):
    if M > len(books):
        return -1  # More students than books, not possible

    left, right = max(books), sum(books)
    min_max_pages = right

    while left <= right:
        mid = left + (right - left) // 2

        if can_allocate_books(books, M, mid):
            min_max_pages = mid
            right = mid - 1
        else:
```

```python
            left = mid + 1

    return min_max_pages

# Example usage:
books = [12, 34, 67, 90]
M = 2
min_pages = min_max_pages(books, M)
print(f"The minimum number of pages that a student needs to read is:
{min_pages}")
```

Median of 2 sorted arrays

```python
def findMedianSortedArrays(nums1, nums2):
    m, n = len(nums1), len(nums2)

    # Ensure nums1 is the smaller array or swap
    if m > n:
        nums1, nums2, m, n = nums2, nums1, n, m

    # Binary search on nums1 to partition
    left, right = 0, m
    half_len = (m + n + 1) // 2

    while left <= right:
        partition_nums1 = (left + right) // 2
        partition_nums2 = half_len - partition_nums1

        max_left_nums1 = float('-inf') if partition_nums1 == 0 else nums1[partition_nums1 - 1]
        min_right_nums1 = float('inf') if partition_nums1 == m else nums1[partition_nums1]

        max_left_nums2 = float('-inf') if partition_nums2 == 0 else nums2[partition_nums2 - 1]
        min_right_nums2 = float('inf') if partition_nums2 == n else nums2[partition_nums2]

        if max_left_nums1 <= min_right_nums2 and max_left_nums2 <= min_right_nums1:
            if (m + n) % 2 == 0:
                return (max(max_left_nums1, max_left_nums2) + min(min_right_nums1,
min_right_nums2)) / 2.0
            else:
                return float(max(max_left_nums1, max_left_nums2))
        elif max_left_nums1 > min_right_nums2:
            right = partition_nums1 - 1
        else:
            left = partition_nums1 + 1

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

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

Minimize Max Distance to Gas Station

```
def min_max_distance(stations, K):
    def can_place_gas_stations(stations, K, distance):
        count = 0
        for i in range(len(stations) - 1):
            count += (stations[i+1] - stations[i] - 1) // distance
            if count > K:
                return False
        return count <= K

    stations.sort()
    left, right = 0, stations[-1] - stations[0]
    min_max_dist = right

    while left <= right:
        mid = left + (right - left) // 2
        if can_place_gas_stations(stations, K, mid):
            min_max_dist = mid
            right = mid - 1
        else:
            left = mid + 1

    return min_max_dist

# Example usage:
stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
K = 9
min_max_dist = min_max_distance(stations, K)
print(f"The minimum maximum distance to place {K} additional gas stations is: {min_max_dist}")
```

Middle of a LinkedList [TortoiseHare Method]

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

def find_middle(head):
    if not head:
        return None

    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow

# Example usage:
# Helper function to create a linked list for testing
def create_linked_list(values):
    if not values:
        return None

    dummy = ListNode()
    curr = dummy
```

```python
    for val in values:
        curr.next = ListNode(val)
        curr = curr.next

    return dummy.next

# Example usage:
values = [1, 2, 3, 4, 5]
head = create_linked_list(values)
middle_node = find_middle(head)

if middle_node:
    print(f"The middle node of the linked list is: {middle_node.val}")
else:
    print("The linked list is empty.")
```

Detect a loop in LL

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

def hasCycle(head):
    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

    return True

# Example usage:
# Helper function to create a linked list with a cycle for testing
def create_linked_list_with_cycle(values, pos):
    if not values:
        return None

    dummy = ListNode(0)
    curr = dummy
    nodes = []

    for val in values:
        node = ListNode(val)
        curr.next = node
        curr = curr.next
        nodes.append(node)

    if pos >= 0:
        nodes[-1].next = nodes[pos]  # create the cycle

    return dummy.next

# Example usage:
```

```python
values = [1, 2, 3, 4, 5]
pos = 1  # position of the node where the cycle starts (index 1 in this case)
head = create_linked_list_with_cycle(values, pos)
has_cycle = hasCycle(head)

if has_cycle:
    print("The linked list has a cycle.")
else:
    print("The linked list does not have a cycle.")
```

Remove Nth node from the back of the LL

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

def removeNthFromEnd(head, n):
    dummy = ListNode(0)
    dummy.next = head
    length = 0
    first = head

    while first:
        length += 1
        first = first.next

    length -= n
    first = dummy

    while length > 0:
        length -= 1
        first = first.next

    first.next = first.next.next
    return dummy.next

# Example usage:
# Helper function to create a linked list for testing
def create_linked_list(values):
    if not values:
        return None

    dummy = ListNode(0)
    curr = dummy

    for val in values:
        curr.next = ListNode(val)
        curr = curr.next

    return dummy.next

# Example usage:
values = [1, 2, 3, 4, 5]
head = create_linked_list(values)
n = 2
new_head = removeNthFromEnd(head, n)

# Print the modified linked list
curr = new_head
```

```python
while curr:
    print(curr.val, end=" -> ")
    curr = curr.next
```

Find the intersection point of Y LL

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

def getIntersectionNode(head1, head2):
    if not head1 or not head2:
        return None

    # Calculate lengths of both lists
    len1 = getLength(head1)
    len2 = getLength(head2)

    # Move head1 and head2 to the same starting point
    p1, p2 = head1, head2
    if len1 > len2:
        for _ in range(len1 - len2):
            p1 = p1.next
    elif len2 > len1:
        for _ in range(len2 - len1):
            p2 = p2.next

    # Find the intersection point
    while p1 and p2:
        if p1 == p2:
            return p1
        p1 = p1.next
        p2 = p2.next

    return None

def getLength(head):
    length = 0
    current = head
    while current:
        length += 1
        current = current.next
    return length

# Example usage:
# Helper function to create linked lists for testing
def create_linked_list(values, pos):
    if not values:
        return None

    dummy = ListNode(0)
    curr = dummy
    nodes = []

    for val in values:
        node = ListNode(val)
        curr.next = node
        curr = curr.next
        nodes.append(node)
```

```python
    if pos >= 0:
        nodes[-1].next = nodes[pos]  # create the intersection

    return dummy.next

# Example usage:
values1 = [1, 2, 3, 4, 5]
values2 = [6, 7]
intersection_pos = 2  # position of intersection node (index 2 in values1)
head1 = create_linked_list(values1, intersection_pos)
head2 = create_linked_list(values2, -1)  # no intersection in this case

intersection_node = getIntersectionNode(head1, head2)

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

Sort LL

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

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

    # Step 1: Split the list into two halves
    mid = getMiddle(head)
    left_half = head
    right_half = mid.next
    mid.next = None

    # Step 2: Recursively sort each half
    left_sorted = sortList(left_half)
    right_sorted = sortList(right_half)

    # Step 3: Merge sorted halves
    return merge(left_sorted, right_sorted)

def getMiddle(head):
    if not head:
        return head

    slow = head
    fast = head
    prev = None

    while fast and fast.next:
        prev = slow
        slow = slow.next
        fast = fast.next.next

    return prev

def merge(l1, l2):
```

```python
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    if l1:
        current.next = l1
    if l2:
        current.next = l2

    return dummy.next

# Helper function to create a linked list from a list of values
def create_linked_list(values):
    if not values:
        return None

    dummy = ListNode(0)
    curr = dummy

    for val in values:
        curr.next = ListNode(val)
        curr = curr.next

    return dummy.next

# Helper function to print the linked list
def print_linked_list(head):
    curr = head
    while curr:
        print(curr.val, end=" -> ")
        curr = curr.next
    print("None")

# Example usage:
values = [4, 2, 1, 3]
head = create_linked_list(values)
sorted_head = sortList(head)
print("Sorted linked list:")
print_linked_list(sorted_head)
```

Segrregate odd and even nodes in LL

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

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

    # Create dummy nodes for odd and even lists
```

```python
    odd_dummy = ListNode(0)
    even_dummy = ListNode(0)
    odd_tail = odd_dummy
    even_tail = even_dummy

    current = head

    while current:
        if current.val % 2 == 1:  # Odd node
            odd_tail.next = current
            odd_tail = odd_tail.next
        else:  # Even node
            even_tail.next = current
            even_tail = even_tail.next

        current = current.next

    # Connect odd_tail to even_dummy.next and terminate even_tail
    odd_tail.next = even_dummy.next
    even_tail.next = None

    # Return the head of the odd list
    return odd_dummy.next

# Helper function to create a linked list from a list of values
def create_linked_list(values):
    if not values:
        return None

    dummy = ListNode(0)
    curr = dummy

    for val in values:
        curr.next = ListNode(val)
        curr = curr.next

    return dummy.next

# Helper function to print the linked list
def print_linked_list(head):
    curr = head
    while curr:
        print(curr.val, end=" -> ")
        curr = curr.next
    print("None")

# Example usage:
values = [1, 2, 3, 4, 5, 6, 7, 8, 9]
head = create_linked_list(values)
segregated_head = segregateOddEven(head)
print("Segregated linked list:")
print_linked_list(segregated_head)
```

Print all subsequences/Power Set

```python
def generate_subsequences(nums):
    def backtrack(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
```

```
            backtrack(i + 1, path)
            path.pop()

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

# Example usage:
nums = [1, 2, 3]
all_subsequences = generate_subsequences(nums)

print("All subsequences (Power Set):")
for subset in all_subsequences:
    print(subset)
```

Combination Sum

```
def combinationSum(candidates, target):
    def backtrack(start, path, curr_sum):
        if curr_sum == target:
            result.append(path[:])
            return
        if curr_sum > target:
            return

        for i in range(start, len(candidates)):
            path.append(candidates[i])
            backtrack(i, path, curr_sum + candidates[i])
            path.pop()

    candidates.sort()
    result = []
    backtrack(0, [], 0)
    return result

# Example usage:
candidates = [2, 3, 6, 7]
target = 7
combinations = combinationSum(candidates, target)

print("All combinations summing up to", target, ":")
for combination in combinations:
    print(combination)
```

N Queen

```
def solveNQueens(n):
    def is_safe(board, row, col):
        # Check if there is a queen in the same column
        for i in range(row):
            if board[i] == col:
                return False
            # Check diagonals: (row1 - row2) == abs(col1 - col2)
            if abs(board[i] - col) == row - i:
                return False
        return True

    def backtrack(row):
```

```
        if row == n:
            solutions.append(["".join(["Q" if board[i] == col else "." for col in range(n)])
for i in range(n)])
            return

        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1

    board = [-1] * n
    solutions = []
    backtrack(0)
    return solutions

# Example usage:
n = 4
solutions = solveNQueens(n)

print(f"All solutions for {n}-Queens problem:")
for idx, solution in enumerate(solutions):
    print(f"Solution {idx + 1}:")
    for row in solution:
        print(row)
    print()
```

Sudoko Solver

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

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

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

        return True

    def solve():
        for row in range(9):
            for col in range(9):
                if board[row][col] == 0:
                    for num in range(1, 10):
                        if is_valid(row, col, num):
                            board[row][col] = num
                            if solve():
                                return True
```

```python
                                board[row][col] = 0
                        return False
        return True

    solve()

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

solveSudoku(board)

print("Solved Sudoku:")
for row in board:
    print(row)
```

M Coloring Problem

```python
def is_safe(v, graph, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

def m_coloring_util(graph, m, color, v):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(v, graph, color, c):
            color[v] = c
            if m_coloring_util(graph, m, color, v + 1):
                return True
            color[v] = 0

    return False

def m_coloring(graph, m):
    color = [0] * len(graph)
    if not m_coloring_util(graph, m, color, 0):
        return False

    print("Solution exists. The coloring is:")
    for i in range(len(graph)):
        print(f"Vertex {i + 1} -> Color {color[i]}")

    return True

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

```
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 0]
]
m = 3  # Number of colors

m_coloring(graph, m)
```

Word Search

```
def exist(board, word):
    def backtrack(row, col, index):
        # Base case: if we have matched all characters in word
        if index == len(word):
            return True

        # Check boundary conditions and if the cell matches the current character
        if row < 0 or row >= len(board) or col < 0 or col >= len(board[0]) or board[row][col]
!= word[index]:
            return False

        # Mark the cell as visited
        temp = board[row][col]
        board[row][col] = '#'

        # Explore neighbors (up, down, left, right)
        found = (backtrack(row + 1, col, index + 1) or
                 backtrack(row - 1, col, index + 1) or
                 backtrack(row, col + 1, index + 1) or
                 backtrack(row, col - 1, index + 1))

        # Backtrack: restore the cell
        board[row][col] = temp

        return found

    # Start searching for the word from each cell in the board
    for r in range(len(board)):
        for c in range(len(board[0])):
            if backtrack(r, c, 0):
                return True

    return False

# Example usage:
board = [
    ['A','B','C','E'],
    ['S','F','C','S'],
    ['A','D','E','E']
]
word = "ABCCED"

print("Does the word exist in the board?", exist(board, word))
```

Next Greater Element

```
def nextGreaterElements(nums):
    n = len(nums)
```

```
    result = [-1] * n
    stack = []

    # Traverse the array from right to left
    for i in range(n - 1, -1, -1):
        # Pop elements from the stack that are less than or equal to nums[i]
        while stack and stack[-1] <= nums[i]:
            stack.pop()

        # If stack is not empty, the top element is the next greater element
        if stack:
            result[i] = stack[-1]

        # Push current element onto the stack
        stack.append(nums[i])

    return result

# Example usage:
nums = [4, 2, 7, 3, 1, 5]
print("Original array:", nums)
print("Next Greater Elements:", nextGreaterElements(nums))
```

Trapping Rainwater

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

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

    while left <= right:
        left_max = max(left_max, height[left])
        right_max = max(right_max, height[right])

        if left_max < right_max:
            water_trapped += left_max - height[left]
            left += 1
        else:
            water_trapped += right_max - height[right]
            right -= 1

    return water_trapped

# Example usage:
height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
print("Height of bars:", height)
print("Trapped rainwater:", trap(height))
```

Largest rectangle in a histogram

```
def largestRectangleArea(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 as the
smallest 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:
heights = [2, 1, 5, 6, 2, 3]
print("Heights of histogram bars:", heights)
print("Largest rectangle area:", largestRectangleArea(heights))
```

Asteroid Collision

```
def asteroidCollision(asteroids):
    stack = []

    for asteroid in asteroids:
        while stack and asteroid < 0 and stack[-1] > 0:
            if stack[-1] < abs(asteroid):
                stack.pop()
                continue
            elif stack[-1] == abs(asteroid):
                stack.pop()
            break
        else:
            stack.append(asteroid)

    return stack

# Example usage:
asteroids = [5, 10, -5]
print("Initial asteroids:", asteroids)
print("After collision:", asteroidCollision(asteroids))

asteroids = [8, -8]
print("\nInitial asteroids:", asteroids)
print("After collision:", asteroidCollision(asteroids))

asteroids = [10, 2, -5]
print("\nInitial asteroids:", asteroids)
```

```
print("After collision:", asteroidCollision(asteroids))

asteroids = [-2, -1, 1, 2]
print("\nInitial asteroids:", asteroids)
print("After collision:", asteroidCollision(asteroids))
```

Sliding Window maximum

```
from collections import deque

def maxSlidingWindow(nums, k):
    n = len(nums)
    if n * k == 0:
        return []
    if k == 1:
        return nums

    deque = []
    result = []

    for i in range(n):
        # Remove indices from deque that are out of current window
        if deque and deque[0] <= i - k:
            deque.pop(0)

        # Remove elements from deque that are less than the current element nums[i]
        while deque and nums[deque[-1]] <= nums[i]:
            deque.pop()

        # Add current index to deque
        deque.append(i)

        # Append the maximum of current window to the result
        if i >= k - 1:
            result.append(nums[deque[0]])

    return result

# Example usage:
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
print("Original array:", nums)
print("Sliding window maximum (k =", k, "):", maxSlidingWindow(nums, k))
```

LRU cache (IMPORTANT)

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key: int) -> int:
        if key in self.cache:
            # Move the accessed key to the end to mark it as recently used
            self.cache.move_to_end(key)
            return self.cache[key]
```

```
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            # Update the value and move the key to the end
            self.cache[key] = value
            self.cache.move_to_end(key)
        else:
            # Add the new key-value pair
            if len(self.cache) >= self.capacity:
                # Remove the least recently used item (first item in OrderedDict)
                self.cache.popitem(last=False)
            self.cache[key] = value

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

Kth largest element in an array [use priority queue]

```
import heapq

def findKthLargest(nums, k):
    # Create a min-heap
    heap = []

    # Push first k elements into the heap
    for num in nums[:k]:
        heapq.heappush(heap, num)

    # Iterate through remaining elements
    for num in nums[k:]:
        # If current element is larger than the smallest element in heap
        if num > heap[0]:
            # Replace the smallest element (root) with current element
            heapq.heapreplace(heap, num)

    # The root of the heap is the Kth largest element
    return heap[0]

# Example usage:
nums = [3, 2, 1, 5, 6, 4]
k = 2
print("Array:", nums)
print(f"{k}th largest element:", findKthLargest(nums, k))
```

Task Scheduler

```
import heapq
from collections import defaultdict
```

```python
def leastInterval(tasks, n):
    # Step 1: Count frequencies of each task
    task_counts = defaultdict(int)
    for task in tasks:
        task_counts[task] += 1

    # Step 2: Use a max-heap to prioritize tasks by their frequency
    max_heap = []
    for count in task_counts.values():
        heapq.heappush(max_heap, -count)  # Use negative to simulate max-heap

    # Step 3: Execute tasks in order of highest frequency
    total_time = 0
    while max_heap:
        # Step 4: Execute up to n + 1 tasks (cooldown period)
        cooldown_tasks = []
        for _ in range(n + 1):
            if max_heap:
                count = -heapq.heappop(max_heap)  # Pop the task with highest frequency
                if count > 1:
                    cooldown_tasks.append(count - 1)  # Push back if task still needs to be
executed
            total_time += 1
            if not max_heap and not cooldown_tasks:
                break  # No more tasks left to execute

        # Step 5: Push the cooldown tasks back into the max-heap
        for count in cooldown_tasks:
            heapq.heappush(max_heap, -count)

    return total_time

# Example usage:
tasks = ["A", "A", "A", "B", "B", "B"]
n = 2
print("Tasks:", tasks)
print("Minimum time required:", leastInterval(tasks, n))
```

Min Heap and Max Heap Implementation

```python
import heapq

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

    def push(self, val):
        heapq.heappush(self.heap, val)

    def pop(self):
        return heapq.heappop(self.heap)

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

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

import heapq
```

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

    def push(self, val):
        heapq.heappush(self.heap, -val)  # Push negative value for max-heap behavior

    def pop(self):
        return -heapq.heappop(self.heap)  # Return negated value to restore original value

    def peek(self):
        return -self.heap[0] if self.heap else None

    def size(self):
        return len(self.heap)
```

```python
# Example usage of MinHeap
min_heap = MinHeap()
min_heap.push(3)
min_heap.push(2)
min_heap.push(5)
print("MinHeap size:", min_heap.size())  # Output: 3
print("MinHeap peek:", min_heap.peek())  # Output: 2
print("MinHeap pop:", min_heap.pop())    # Output: 2
print("MinHeap peek after pop:", min_heap.peek())  # Output: 3

# Example usage of MaxHeap
max_heap = MaxHeap()
max_heap.push(3)
max_heap.push(2)
max_heap.push(5)
print("\nMaxHeap size:", max_heap.size())  # Output: 3
print("MaxHeap peek:", max_heap.peek())    # Output: 5
print("MaxHeap pop:", max_heap.pop())      # Output: 5
print("MaxHeap peek after pop:", max_heap.peek())  # Output: 3
```

Diameter of Binary Tree

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

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.diameter = 0

        def maxDepth(node):
            if not node:
                return 0
            left_depth = maxDepth(node.left)
            right_depth = maxDepth(node.right)
            # Update diameter at each node
            self.diameter = max(self.diameter, left_depth + right_depth)
            # Return the maximum depth of the current node
            return 1 + max(left_depth, right_depth)
```

```
        maxDepth(root)
        return self.diameter

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

solution = Solution()
print("Diameter of binary tree:", solution.diameterOfBinaryTree(root))  # Output: 3
```

Maximum path sum

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

class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        self.max_sum = float('-inf')

        def maxPathDown(node):
            if not node:
                return 0
            # Recursively find the maximum sum from left and right subtrees
            left_max = max(0, maxPathDown(node.left))
            right_max = max(0, maxPathDown(node.right))
            # Update the maximum path sum found so far
            self.max_sum = max(self.max_sum, node.val + left_max + right_max)
            # Return the maximum sum path that can be extended upwards
            return node.val + max(left_max, right_max)

        maxPathDown(root)
        return self.max_sum

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)

solution = Solution()
print("Maximum path sum:", solution.maxPathSum(root))  # Output: 6
```

Bottom View of Binary Tree

```
from collections import deque, defaultdict

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

```python
        self.right = right

class Solution:
    def bottomView(self, root: TreeNode):
        if not root:
            return []

        bottom_view_map = {}
        queue = deque([(root, 0, 0)])  # (node, horizontal_distance, depth)

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

            # Update the bottom view map
            bottom_view_map[hd] = (node.val, depth)

            if node.left:
                queue.append((node.left, hd - 1, depth + 1))
            if node.right:
                queue.append((node.right, hd + 1, depth + 1))

        # Extract the nodes from the bottom view map sorted by horizontal distance
        bottom_view = [value[0] for key, value in sorted(bottom_view_map.items())]

        return bottom_view

# Example usage:
# Constructing a sample binary tree
root = TreeNode(20)
root.left = TreeNode(8)
root.right = TreeNode(22)
root.left.left = TreeNode(5)
root.left.right = TreeNode(3)
root.right.right = TreeNode(25)
root.left.right.left = TreeNode(10)
root.left.right.right = TreeNode(14)

solution = Solution()
print("Bottom view of binary tree:", solution.bottomView(root))  # Output: [5, 10, 14, 22, 25]
```

LCA in Binary Tree

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

class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
        # Base case: if root is None or if either p or q is found, return root
        if not root or root == p or root == q:
            return root

        # Recursively search in left and right subtrees
        left_lca = self.lowestCommonAncestor(root.left, p, q)
        right_lca = self.lowestCommonAncestor(root.right, p, q)

        # If both left_lca and right_lca are not None, root is the LCA
        if left_lca and right_lca:
```

```python
            return root
        # Otherwise, return whichever is not None
        return left_lca if left_lca else right_lca

# Example usage:
# Constructing a sample binary tree
root = TreeNode(3)
root.left = TreeNode(5)
root.right = TreeNode(1)
root.left.left = TreeNode(6)
root.left.right = TreeNode(2)
root.right.left = TreeNode(0)
root.right.right = TreeNode(8)
root.left.right.left = TreeNode(7)
root.left.right.right = TreeNode(4)

p = root.left
q = root.right

solution = Solution()
lca_node = solution.lowestCommonAncestor(root, p, q)
print("LCA of nodes", p.val, "and", q.val, "is:", lca_node.val)  # Output: 3
```

Minimum time taken to BURN the Binary Tree from a Node

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

class Solution:
    def minTimeToBurnTree(self, root: TreeNode, target: int) -> int:
        # Helper function to perform DFS and calculate distances
        def dfs(node, parent):
            if not node:
                return 0
            if node.val == target:
                distance[node] = 0
                return 0

            left_distance = dfs(node.left, node)
            right_distance = dfs(node.right, node)

            if left_distance != -1:
                distance[node] = left_distance + 1
                return left_distance + 1
            if right_distance != -1:
                distance[node] = right_distance + 1
                return right_distance + 1

            return -1

        # Find the target node
        def findTarget(node, target):
            if not node:
                return False
            if node.val == target:
                return True
            return findTarget(node.left, target) or findTarget(node.right, target)
```

```
        if not findTarget(root, target):
            return -1

        distance = {}
        dfs(root, None)

        return max(distance.values())

# Example usage:
# Constructing a sample binary tree
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)

target_node = 5
solution = Solution()
print("Minimum time to burn the tree from node", target_node, ":",
solution.minTimeToBurnTree(root, target_node))  # Output: 2
```

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

class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if not preorder or not inorder:
            return None

        root_val = preorder[0]
        root = TreeNode(root_val)

        # Find the index of root_val in inorder traversal
        root_index_inorder = inorder.index(root_val)

        # Recursively build left and right subtrees
        root.left = self.buildTree(preorder[1:1 + root_index_inorder],
inorder[:root_index_inorder])
        root.right = self.buildTree(preorder[1 + root_index_inorder:],
inorder[root_index_inorder + 1:])

        return root

# Example usage:
preorder = [3, 9, 20, 15, 7]
inorder = [9, 3, 15, 20, 7]

solution = Solution()
root = solution.buildTree(preorder, inorder)

# Function to print inorder traversal of the constructed tree
def printInorder(node):
```

```python
    if not node:
        return
    printInorder(node.left)
    print(node.val, end=' ')
    printInorder(node.right)

print("Inorder traversal of the constructed tree:")
printInorder(root)  # Output: 9 3 15 20 7
```

Morris Preorder Traversal of 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 Solution:
    def morrisPreorderTraversal(self, root: TreeNode):
        result = []
        current = root

        while current:
            if not current.left:
                result.append(current.val)
                current = current.right
            else:
                # Find the predecessor node (rightmost node in the left subtree)
                predecessor = current.left
                while predecessor.right and predecessor.right != current:
                    predecessor = predecessor.right

                if not predecessor.right:
                    # Establish the temporary link from predecessor's right child to current
                    result.append(current.val)  # Print in preorder here
                    predecessor.right = current
                    current = current.left
                else:
                    # Remove the link and move to the right child
                    predecessor.right = None
                    current = current.right

        return result

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

solution = Solution()
print("Morris Preorder Traversal:", solution.morrisPreorderTraversal(root))  # Output: [1, 2, 
4, 5, 3]
```

Delete a Node in Binary Search Tree

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

class Solution:
    def minValueNode(self, node):
        current = node
        # Find the leftmost leaf node
        while current.left:
            current = current.left
        return current

    def deleteNode(self, root: TreeNode, key: int) -> TreeNode:
        if not root:
            return root

        # If the key to be deleted is smaller than the root's key, then it lies in the left
subtree
        if key < root.val:
            root.left = self.deleteNode(root.left, key)

        # If the key to be deleted is greater than the root's key, then it lies in the right
subtree
        elif key > root.val:
            root.right = self.deleteNode(root.right, key)

        # If key is same as root's key, then this is the node to be deleted
        else:
            # Case 1: Node with only one child or no child
            if not root.left:
                temp = root.right
                root = None
                return temp
            elif not root.right:
                temp = root.left
                root = None
                return temp

            # Case 2: Node with two children: Get the inorder successor (smallest in the right
subtree)
            temp = self.minValueNode(root.right)
            # Copy the inorder successor's content to this node
            root.val = temp.val
            # Delete the inorder successor
            root.right = self.deleteNode(root.right, temp.val)

        return root

# Helper function to perform inorder traversal of the BST
def inorderTraversal(node):
    if node:
        inorderTraversal(node.left)
        print(node.val, end=" ")
        inorderTraversal(node.right)

# Example usage:
# Constructing a sample Binary Search Tree
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
```

```
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.right = TreeNode(7)

solution = Solution()
print("Inorder traversal of the original BST:")
inorderTraversal(root)  # Output: 2 3 4 5 6 7
print("\nDeleting node with key 3...")
root = solution.deleteNode(root, 3)
print("Inorder traversal after deletion:")
inorderTraversal(root)  # Output: 2 4 5 6 7
```

LCA in Binary Search Tree

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

class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
        # Value of current node
        parent_val = root.val

        # Value of p
        p_val = p.val

        # Value of q
        q_val = q.val

        # If both p and q are greater than parent_val
        if p_val > parent_val and q_val > parent_val:
            return self.lowestCommonAncestor(root.right, p, q)

        # If both p and q are lesser than parent_val
        elif p_val < parent_val and q_val < parent_val:
            return self.lowestCommonAncestor(root.left, p, q)

        # We have found the split point, i.e. the LCA node
        else:
            return root

# Constructing a sample Binary Search Tree
root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode(8)
root.left.left = TreeNode(0)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)

p = root.left  # Node with value 2
q = root.right  # Node with value 8

solution = Solution()
lca_node = solution.lowestCommonAncestor(root, p, q)
print("Lowest Common Ancestor of nodes", p.val, "and", q.val, "is:", lca_node.val)  # Output:
6
```

Two Sum In BST | Check if there exists a pair with Sum K

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

class Solution:
    def findTarget(self, root: TreeNode, k: int) -> bool:
        # Helper function to perform inorder traversal and store values in a list
        def inorderTraversal(node):
            if not node:
                return []
            return inorderTraversal(node.left) + [node.val] + inorderTraversal(node.right)

        # Get the sorted list of values using inorder traversal
        sorted_values = inorderTraversal(root)

        # Initialize two pointers
        left, right = 0, len(sorted_values) - 1

        # Two-pointer technique to find the pair with sum equal to K
        while left < right:
            current_sum = sorted_values[left] + sorted_values[right]
            if current_sum == k:
                return True
            elif current_sum < k:
                left += 1
            else:
                right -= 1

        return False

# Example usage:
# Constructing a sample Binary Search Tree
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.right = TreeNode(7)

solution = Solution()
K = 9
print("Does there exist a pair with sum", K, "?", solution.findTarget(root, K))  # Output:
True (since 2 + 7 = 9)
```

Largest BST in Binary Tree

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

class Solution:
    def largestBSTSubtree(self, root: TreeNode) -> int:
        # Helper function to determine if a subtree is a valid BST and its size
        def find_largest_bst(node):
```

```python
            if not node:
                return (0, float('inf'), float('-inf'), True)

            # Postorder traversal to get results from children
            left_size, left_min, left_max, is_left_bst = find_largest_bst(node.left)
            right_size, right_min, right_max, is_right_bst = find_largest_bst(node.right)

            # Check if current subtree rooted at 'node' is a valid BST
            if is_left_bst and is_right_bst and left_max < node.val < right_min:
                current_size = 1 + left_size + right_size
                current_min = min(left_min, node.val)
                current_max = max(right_max, node.val)
                return (current_size, current_min, current_max, True)
            else:
                # If not a valid BST, return information about this subtree
                return (max(left_size, right_size), float('-inf'), float('inf'), False)

        # Call the recursive function on the root of the tree
        max_size, _, _, _ = find_largest_bst(root)
        return max_size

# Example usage:
# Constructing a sample Binary Tree
root = TreeNode(10)
root.left = TreeNode(5)
root.right = TreeNode(15)
root.left.left = TreeNode(1)
root.left.right = TreeNode(8)
root.right.right = TreeNode(7)

solution = Solution()
print("Size of the largest BST in the Binary Tree:", solution.largestBSTSubtree(root))  #
Output: 3 (BST: 5, 1, 8)
```

Rotten Oranges

```python
from collections import deque

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

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

    # Count fresh oranges and initialize queue with rotten oranges
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                fresh_count += 1
            elif grid[i][j] == 2:
                queue.append((i, j, 0))  # (row, col, minutes)

    minutes = 0
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    while queue:
        x, y, minutes = queue.popleft()
```

```python
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                grid[nx][ny] = 2  # Mark fresh orange as rotten
                fresh_count -= 1
                queue.append((nx, ny, minutes + 1))

    if fresh_count > 0:
        return -1  # There are fresh oranges left
    else:
        return minutes

# Example usage:
grid = [
    [2, 1, 1],
    [1, 1, 0],
    [0, 1, 1]
]

print("Minutes to rot all oranges:", orangesRotting(grid))
```

Word ladder – 1

```python
from collections import deque

def ladderLength(beginWord, endWord, wordList):
    # Create a set for faster lookup
    word_set = set(wordList)
    if endWord not in word_set:
        return 0

    queue = deque([(beginWord, 1)])  # (current_word, length of transformation sequence)
    visited = set()
    visited.add(beginWord)

    while queue:
        current_word, length = queue.popleft()

        # Generate all possible next words by changing one letter at a time
        for i in range(len(current_word)):
            for char in 'abcdefghijklmnopqrstuvwxyz':
                next_word = current_word[:i] + char + current_word[i+1:]

                if next_word in word_set and next_word not in visited:
                    if next_word == endWord:
                        return length + 1
                    queue.append((next_word, length + 1))
                    visited.add(next_word)

    return 0  # If no transformation sequence is found

# Example usage:
beginWord = "hit"
endWord = "cog"
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

print("Length of shortest transformation sequence:", ladderLength(beginWord, endWord,
wordList))
```

Number of Distinct Islands [dfs multisource]

```
def numDistinctIslands(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    visited = set()
    distinct_islands = set()

    def dfs(x, y, shape, start_x, start_y):
        if x < 0 or x >= rows or y < 0 or y >= cols or grid[x][y] == 0 or (x, y) in visited:
            return

        visited.add((x, y))
        # Record the relative position of the current cell to the starting point
        shape.append((x - start_x, y - start_y))

        # Explore neighbors in 4 directions (up, down, left, right)
        dfs(x - 1, y, shape, start_x, start_y)  # up
        dfs(x + 1, y, shape, start_x, start_y)  # down
        dfs(x, y - 1, shape, start_x, start_y)  # left
        dfs(x, y + 1, shape, start_x, start_y)  # right

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1 and (i, j) not in visited:
                shape = []
                dfs(i, j, shape, i, j)
                # Convert shape to tuple to make it hashable and store in distinct_islands
                distinct_islands.add(tuple(shape))

    return len(distinct_islands)

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

print("Number of distinct islands:", numDistinctIslands(grid))
```

Course Schedule – II

```
from collections import defaultdict, deque

def findOrder(numCourses, prerequisites):
    # Step 1: Build the graph and indegree array
    graph = defaultdict(list)
    indegree = [0] * numCourses

    for course, prereq in prerequisites:
        graph[prereq].append(course)
        indegree[course] += 1

    # Step 2: Initialize a queue with all courses having zero indegree
    queue = deque([course for course in range(numCourses) if indegree[course] == 0])
    topo_order = []
```

```python
    # Step 3: Perform BFS
    while queue:
        course = queue.popleft()
        topo_order.append(course)

        for neighbor in graph[course]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # Step 4: Check for cycle
    if len(topo_order) != numCourses:
        return []  # Cycle detected

    return topo_order

# Example usage:
numCourses = 4
prerequisites = [[1, 0], [2, 0], [3, 1], [3, 2]]

print("Course order:", findOrder(numCourses, prerequisites))
```

Alien dictionary

```python
from collections import defaultdict, deque

def alienOrder(words):
    # Step 1: Build the graph and indegree array
    graph = defaultdict(set)
    indegree = {ch: 0 for word in words for ch in word}

    # Step 2: Build graph edges and update indegrees
    for i in range(len(words) - 1):
        word1, word2 = words[i], words[i + 1]
        min_len = min(len(word1), len(word2))

        for j in range(min_len):
            if word1[j] != word2[j]:
                if word2[j] not in graph[word1[j]]:
                    graph[word1[j]].add(word2[j])
                    indegree[word2[j]] += 1
                break
        else:
            if len(word1) > len(word2):
                return ""  # Invalid order since longer word should come after shorter ones

    # Step 3: Topological sorting using Kahn's algorithm (BFS)
    queue = deque([ch for ch in indegree if indegree[ch] == 0])
    result = []

    while queue:
        ch = queue.popleft()
        result.append(ch)

        for neighbor in graph[ch]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
```

```python
    # Step 4: Check for cycle
    if len(result) != len(indegree):
        return ""  # Cycle detected

    return "".join(result)

# Example usage:
words = ["wrt", "wrf", "er", "ett", "rftt"]
print("Lexicographical order:", alienOrder(words))
```

Djisktra's Algorithm

```python
import heapq
from collections import defaultdict

def dijkstra(graph, start):
    # Initialize distances with infinity for all nodes except the start node
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Priority queue (min-heap) to store (distance, node)
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # If we have already found a shorter way to this node, skip it
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If found a shorter path to neighbor, update distance and push to queue
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage:
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

start_node = 'A'
distances = dijkstra(graph, start_node)

print("Shortest distances from", start_node + ":")
for node, distance in distances.items():
    print(node, "-", distance)
```

Cheapest flights within k stops

```python
import heapq
from collections import defaultdict, deque

def findCheapestPrice(n, flights, src, dst, k):
    # Step 1: Build the graph using adjacency list and store (node, cost) pairs
    graph = defaultdict(list)
    for u, v, cost in flights:
        graph[u].append((v, cost))

    # Step 2: Implement Dijkstra's algorithm with a priority queue
    min_heap = [(0, src, 0)]  # (cost, node, stops)
    heapq.heapify(min_heap)

    while min_heap:
        current_cost, current_node, stops = heapq.heappop(min_heap)

        if current_node == dst:
            return current_cost

        if stops <= k:
            for neighbor, cost in graph[current_node]:
                heapq.heappush(min_heap, (current_cost + cost, neighbor, stops + 1))

    return -1  # If destination cannot be reached within k stops

# Example usage:
n = 5
flights = [
    [0, 1, 100],
    [1, 2, 100],
    [0, 2, 500]
]
src = 0
dst = 2
k = 1

print("Cheapest price within", k, "stops:", findCheapestPrice(n, flights, src, dst, k))
```

Bellman Ford Algorithm

```python
def bellman_ford(graph, V, src):
    # Step 1: Initialize distances from source to all other vertices as INFINITE
    distances = [float('inf')] * V
    distances[src] = 0

    # Step 2: Relax all edges |V| - 1 times. A simple shortest path from src to any other
    # vertex can have at-most |V| - 1 edges
    for _ in range(V - 1):
        for u, v, weight in graph:
            if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    # 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, weight in graph:
        if distances[u] != float('inf') and distances[u] + weight < distances[v]:
            print("Graph contains negative weight cycle")
            return
```

```
    return distances

# Example usage:
V = 5  # Number of vertices in graph
graph = [
    (0, 1, -1),
    (0, 2, 4),
    (1, 2, 3),
    (1, 3, 2),
    (1, 4, 2),
    (3, 2, 5),
    (3, 1, 1),
    (4, 3, -3)
]
src = 0  # Source vertex

distances = bellman_ford(graph, V, src)
if distances:
    print("Vertex Distance from Source:")
    for i in range(V):
        print(f"{i} \t\t {distances[i]}")
```

Floyd Warshal Algorithm

```
INF = float('inf')

def floyd_warshall(graph):
    V = len(graph)
    dist = [[INF] * V for _ in range(V)]

    # Step 1: Initialize distances based on the graph edges
    for i in range(V):
        for j in range(V):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]

    # Step 2: Compute shortest paths
    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] != INF and dist[k][j] != INF and dist[i][j] > dist[i][k] +
dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    # Step 3: Detect and handle negative weight cycles
    for i in range(V):
        if dist[i][i] < 0:
            print("Negative weight cycle detected")
            return None

    return dist

# Example usage:
graph = [
    [0, 5, INF, 10],
    [INF, 0, 3, INF],
    [INF, INF, 0, 1],
    [INF, INF, INF, 0]
```

```
]

result = floyd_warshall(graph)
if result:
    print("Shortest distances between all pairs of vertices:")
    for row in result:
        print(row)
```

Kruskal's Algorithm

```
class UnionFind:
    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
        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(n, edges):
    # Step 1: Sort all edges in non-decreasing order of their weight
    edges.sort(key=lambda x: x[2])

    # Step 2: Initialize Union-Find data structure
    uf = UnionFind(n)
    mst = []

    # Step 3: Iterate through sorted edges and add to MST if they do not form a cycle
    for u, v, weight in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))

    return mst

# Example usage:
n = 4  # Number of vertices
edges = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]
```

```
minimum_spanning_tree = kruskal(n, edges)
print("Minimum Spanning Tree (MST):")
for edge in minimum_spanning_tree:
    print(edge)
```

Accounts merge

```
from collections import defaultdict

class UnionFind:
    def __init__(self):
        self.parent = {}

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            self.parent[rootY] = rootX

    def add(self, x):
        if x not in self.parent:
            self.parent[x] = x

    def getGroups(self):
        groups = defaultdict(list)
        for key in self.parent:
            root = self.find(key)
            groups[root].append(key)
        return groups

def mergeAccounts(accounts):
    uf = UnionFind()
    email_to_name = {}

    # Step 1: Build union-find structure and email to name mapping
    for account in accounts:
        name = account[0]
        for email in account[1:]:
            uf.add(email)
            email_to_name[email] = name
            uf.union(email, account[1])  # Union with the first email to ensure all emails in
account are connected

    # Step 2: Retrieve groups and format output
    groups = uf.getGroups()
    merged_accounts = []

    for root, emails in groups.items():
        name = email_to_name[emails[0]]
        merged_accounts.append([name] + sorted(emails))

    return merged_accounts
```

```python
# Example usage:
accounts = [
    ["John", "johnsmith@mail.com", "john00@mail.com"],
    ["John", "johnnybravo@mail.com"],
    ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
    ["Mary", "mary@mail.com"]
]

merged_accounts = mergeAccounts(accounts)
print("Merged accounts:", merged_accounts)
```

Bridges in Graph

```python
from collections import defaultdict

def findBridges(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    discovery = [-1] * n  # Discovery time of nodes
    low = [-1] * n        # Lowest discovery time reachable from the node
    parent = [-1] * n     # Parent nodes in DFS tree
    time = [0]            # Global time variable

    bridges = []

    def dfs(u):
        nonlocal time
        discovery[u] = low[u] = time[0]
        time[0] += 1

        for v in graph[u]:
            if discovery[v] == -1:  # v is not visited
                parent[v] = u
                dfs(v)

                # Check if the subtree rooted at v has a connection back to u
                low[u] = min(low[u], low[v])

                # If the lowest vertex reachable from subtree under v is below u in DFS tree,
                # then u-v is a bridge
                if low[v] > discovery[u]:
                    bridges.append((u, v))

            elif v != parent[u]:  # Update low value of u for parent function calls.
                low[u] = min(low[u], discovery[v])

    # Run DFS from all vertices to handle disconnected graph
    for i in range(n):
        if discovery[i] == -1:
            dfs(i)

    return bridges

# Example usage:
n = 5  # Number of vertices
edges = [
    (0, 1),
```

```
    (0, 2),
    (1, 2),
    (2, 3),
    (3, 4)
]

print("Bridges in the graph:")
print(findBridges(n, edges))
```

Maximum sum of non-adjacent elements (DP 5)

```python
def max_sum_non_adjacent(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    dp = [0] * len(nums)
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in range(2, len(nums)):
        dp[i] = max(dp[i-1], nums[i] + dp[i-2])

    return dp[-1]

# Example usage:
nums = [2, 1, 5, 8, 4]
print(max_sum_non_adjacent(nums))  # Output: 11
```

Ninja's Training (DP 7)

```python
def max_coins_collected(grid):
    if not grid:
        return 0

    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]

    dp[0][0] = grid[0][0]

    # Initialize first row
    for j in range(1, n):
        dp[0][j] = grid[0][j] + dp[0][j-1]

    # Initialize first column
    for i in range(1, m):
        dp[i][0] = grid[i][0] + dp[i-1][0]

    # Fill the dp table
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = grid[i][j] + max(dp[i-1][j], dp[i][j-1])

    return dp[m-1][n-1]

# Example usage:
grid = [
```

```
    [1, 3, 1],
    [1, 5, 1],
    [4, 2, 1]
]

print(max_coins_collected(grid))  # Output: 12
```

Minimum path sum in Grid (DP 10)

```
def min_path_sum(grid):
    if not grid:
        return 0

    m, n = len(grid), len(grid[0])

    # Create a DP table
    dp = [[0] * n for _ in range(m)]

    # Initialize the starting point
    dp[0][0] = grid[0][0]

    # Initialize the first row
    for j in range(1, n):
        dp[0][j] = grid[0][j] + dp[0][j-1]

    # Initialize the first column
    for i in range(1, m):
        dp[i][0] = grid[i][0] + dp[i-1][0]

    # Fill the DP table
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])

    # Return the minimum path sum to reach the bottom-right corner
    return dp[m-1][n-1]

# Example usage:
grid = [
    [1, 3, 1],
    [1, 5, 1],
    [4, 2, 1]
]

print(min_path_sum(grid))  # Output: 7
```

Subset sum equal to target (DP- 14)

```
def subset_sum(nums, target):
    n = len(nums)

    # Create a DP table
    dp = [[False] * (target + 1) for _ in range(n + 1)]

    # Base case initialization
    for i in range(n + 1):
        dp[i][0] = True  # True for sum = 0 (empty subset)
```

```python
    # Fill the DP table
    for i in range(1, n + 1):
        for j in range(1, target + 1):
            if j < nums[i - 1]:
                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
```

0/1 Knapsack (DP - 19)

```python
def knapsack_01(W, wt, val, n):
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if wt[i - 1] <= w:
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][W]

# Example usage:
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)

print(knapsack_01(W, wt, val, n))  # Output: 220
```

Rod Cutting Problem | (DP - 24)

```python
def rod_cutting(n, prices):
    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 = len(prices)

print(rod_cutting(n, prices))  # Output: 22
```

Longest Common Subsequence

```
def longest_common_subsequence(X, Y):
    m = len(X)
    n = len(Y)

    # Create a DP table
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[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:
X = "abcde"
Y = "ace"

print(longest_common_subsequence(X, Y))  # Output: 3 (LCS is "ace")
```

Longest Palindromic Subsequence | (DP-28)

```
def longest_palindromic_subsequence(s):
    n = len(s)

    # Create a DP table
    dp = [[0] * n for _ in range(n)]

    # Every single character is a palindrome of length 1
    for i in range(n):
        dp[i][i] = 1

    # Fill the DP table
    for length in range(2, n + 1):  # Length of substring
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    return dp[0][n-1]

# Example usage:
s = "bbbab"

print(longest_palindromic_subsequence(s))  # Output: 4 (LPS is "bbbb" or "bbab")
```

Edit Distance | (DP-33)

```
def min_distance(word1, word2):
    m = len(word1)
    n = len(word2)
```

```python
    # Create a DP table
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the DP table
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j  # All insertions
            elif j == 0:
                dp[i][j] = i  # All deletions
            elif word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i][j - 1],     # Insertion
                               dp[i - 1][j],     # Deletion
                               dp[i - 1][j - 1]  # Substitution
                              ) + 1

    return dp[m][n]

# Example usage:
word1 = "horse"
word2 = "ros"

print(min_distance(word1, word2))  # Output: 3
```

Buy and Stock Sell IV |(DP-38)

```python
def max_profit(k, prices):
    n = len(prices)
    if n <= 1:
        return 0

    # If k >= n/2, it's equivalent to the unlimited transactions case (Buy and Sell Stock II)
    if k >= n // 2:
        max_profit = 0
        for i in range(1, n):
            if prices[i] > prices[i - 1]:
                max_profit += prices[i] - prices[i - 1]
        return max_profit

    # Create hold and sold arrays
    hold = [[float('-inf')] * n for _ in range(k + 1)]
    sold = [[0] * n for _ in range(k + 1)]

    for i in range(n):
        hold[0][i] = -prices[i]

    for t in range(1, k + 1):
        for i in range(1, n):
            hold[t][i] = max(hold[t][i - 1], sold[t - 1][i - 1] - prices[i])
            sold[t][i] = max(sold[t][i - 1], hold[t][i - 1] + prices[i])

    return sold[k][n - 1]

# Example usage:
prices = [3, 2, 6, 5, 0, 3]
k = 2
```

```
  print(max_profit(k, prices))  # Output: 7 (Buy on day 2 (price = 2) and sell on day 3 (price =
6), then buy on day 5 (price = 0) and sell on day 6 (price = 3))
```

Longest Increasing Subsequence

```
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[j] < nums[i]:
                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 (one longest increasing subsequence is [2, 3, 7, 101])
```

Burst Balloons|(DP-51)

```
def maxCoins(nums):
    nums = [1] + nums + [1]  # Add virtual balloons with value 1 at both ends
    n = len(nums)
    dp = [[0] * n for _ in range(n)]

    for length in range(1, n - 1):  # Length of subarray
        for i in range(1, n - length - 1):
            j = i + length - 1
            for k in range(i, j + 1):
                coins = nums[i - 1] * nums[k] * nums[j + 1]
                dp[i][j] = max(dp[i][j], dp[i][k - 1] + coins + dp[k + 1][j])

    return dp[1][n - 2]

# Example usage:
nums = [3, 1, 5, 8]

print(maxCoins(nums))  # Output: 167 (burst balloons in the order 1, 3, 2, 4)
```

Implement Trie - 2 (Prefix Tree)

```
class TrieNode:
    def __init__(self):
        self.children = {}  # Dictionary to store child nodes
        self.is_end_of_word = False  # Flag to indicate if this node represents the end of a
word
class Trie:
    def __init__(self):
        self.root = TrieNode()  # Initialize the Trie with an empty root node
```

```
    def insert(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                current.children[char] = TrieNode()  # Create a new node if the character
doesn't exist
            current = current.children[char]
        current.is_end_of_word = True  # Mark the end of the word

    def search(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                return False
            current = current.children[char]
        return current.is_end_of_word

    def starts_with(self, prefix):
        current = self.root
        for char in prefix:
            if char not in current.children:
                return False
            current = current.children[char]
        return True

# Create a Trie instance
trie = Trie()

# Insert words into the Trie
trie.insert("apple")
trie.insert("banana")
trie.insert("app")
trie.insert("appetizer")

# Search for words
print(trie.search("apple"))  # Output: True
print(trie.search("banana"))  # Output: True
print(trie.search("appetizer"))  # Output: True
print(trie.search("apricot"))  # Output: False

# Check prefixes
print(trie.starts_with("app"))  # Output: True
print(trie.starts_with("ban"))  # Output: True
print(trie.starts_with("pea"))  # Output: False
```

Maximum XOR With an Element From Array

```
class TrieNode:
    def __init__(self):
        self.children = {}

class Trie:
    def __init__(self, maximumBit):
        self.root = TrieNode()
        self.maximumBit = maximumBit
        self.mask = (1 << maximumBit) - 1

    def insert(self, num):
        current = self.root
        for i in range(self.maximumBit - 1, -1, -1):
```

```python
            bit = (num >> i) & 1
            if bit not in current.children:
                current.children[bit] = TrieNode()
            current = current.children[bit]

    def find_maximum_xor(self, num):
        current = self.root
        result = 0
        for i in range(self.maximumBit - 1, -1, -1):
            bit = (num >> i) & 1
            opposite_bit = 1 - bit
            if opposite_bit in current.children:
                result |= (1 << i)
                current = current.children[opposite_bit]
            else:
                current = current.children[bit]
        return result

def maximizeXor(nums, queries):
    maximumBit = 0
    for num in nums:
        maximumBit = max(maximumBit, num.bit_length())

    trie = Trie(maximumBit)
    for num in nums:
        trie.insert(num)

    n = len(queries)
    result = []
    for xi, mi in queries:
        max_xor = trie.find_maximum_xor(xi)
        result.append(max_xor)

    return result

# Example usage:
nums = [0, 1, 2, 3, 4]
queries = [[3, 1], [1, 3], [5, 6]]
print(maximizeXor(nums, queries))  # Output: [3, 3, 7]
```

Number of Distinct Substrings in a String

```python
def build_suffix_array(s):
    n = len(s)
    suffix_array = sorted(range(n), key=lambda i: s[i:])
    return suffix_array

def build_lcp_array(s, suffix_array):
    n = len(s)
    rank = [0] * n
    lcp = [0] * n

    for i, suffix_index in enumerate(suffix_array):
        rank[suffix_index] = i

    h = 0
    for i in range(n):
        if rank[i] > 0:
            j = suffix_array[rank[i] - 1]
            while i + h < n and j + h < n and s[i + h] == s[j + h]:
```

```
                h += 1
            lcp[rank[i]] = h
            if h > 0:
                h -= 1

    return lcp

def count_distinct_substrings(s):
    suffix_array = build_suffix_array(s)
    lcp_array = build_lcp_array(s, suffix_array)

    n = len(s)
    distinct_substrings = n * (n + 1) // 2 - sum(lcp_array)

    return distinct_substrings

# Example usage:
s = "banana"
print(count_distinct_substrings(s))  # Output: 21
```

Minimum number of bracket reversals needed to make an expression balanced

```
def min_reversals_to_balance(expression):
    if len(expression) % 2 != 0:
        return -1  # If the length of expression is odd, it can't be balanced

    stack = []
    for char in expression:
        if char == '(':
            stack.append(char)
        elif char == ')':
            if stack and stack[-1] == '(':
                stack.pop()
            else:
                stack.append(char)

    # At this point, stack contains unbalanced brackets
    unbalanced_open = stack.count('(')
    unbalanced_close = stack.count(')')

    # Calculate minimum reversals needed
    reversals_open = (unbalanced_open + 1) // 2
    reversals_close = (unbalanced_close + 1) // 2

    return reversals_open + reversals_close

# Example usage:
expression = "))(())(("
print("Minimum reversals needed:", min_reversals_to_balance(expression))
```

Rabin Karp

```
class RabinKarp:
    def __init__(self, pattern):
        self.pattern = pattern
        self.pattern_hash = hash(pattern)
        self.pattern_length = len(pattern)
```

```python
    def search(self, text):
        text_length = len(text)
        pattern_length = self.pattern_length
        pattern_hash = self.pattern_hash

        results = []

        # Compute the hash of the first window in the text
        text_hash = hash(text[:pattern_length])

        # Slide over the text and compare hash values
        for i in range(text_length - pattern_length + 1):
            if text_hash == pattern_hash and text[i:i+pattern_length] == self.pattern:
                results.append(i)

            # Update the hash for the next window
            if i < text_length - pattern_length:
                text_hash = hash(text[i+1:i+1+pattern_length])

        return results

# Example usage:
pattern = "ab"
text = "abcabcabc"
rk = RabinKarp(pattern)
matches = rk.search(text)
print("Pattern matches found at positions:", matches)
```

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  # Z[0] is always n
    return Z

text = "aabcaabxaaaz"
Z = compute_z(text)
print("Z array for text '{}' is:".format(text), Z)
```

KMP algo / LPS(pi) array

```python
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length = 0  # length of the previous longest prefix suffix

    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

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[]
    matches = []

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

        if j == m:
            matches.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return matches

# Example usage:
text = "ababcabcabababd"
pattern = "ababd"
matches = kmp_search(text, pattern)
print("Pattern matches found at positions:", matches)
```