Learn The Basics

```
#num = 153
#digit = 3
#num = 1^3 + 5^3 + 3^3 = 153
#armstrong = true

#first find out digits in the number
#use the digits to calculate their sum

num = 170
n = num
count = 0
temp = 0
arm = num

while(n!=0):
    n = n//10
    count = count + 1

while (arm!=0):
    temp = pow((arm%10),count) + temp
    arm = arm//10

if temp == num:
    print("Armstrong number")
else:
    print("Not an armstrong")
```

To count digits of a number

```
#num = 123
#digits = 3

#123//10 = 12
#123%10 = 3

num = 123
count = 0
while(num!=0):
    num = num//10
    count = count + 1

print(count)
```

To find all the divisors of a number

```
# num = 36
# 36 times iteration that is not good

# Optimal Solution
# a * a = n
# a = sqrt n
# a * b < = n
# b = n//a

num = 36

for i in range(1, int(num**0.5)+1):
    if num%i==0:
        print(i)
        if (i!=num//i):
            print(num//i)
```

To find greatest common divisors

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage:
num1 = 24
num2 = 36
print("GCF of", num1, "and", num2, "is:", gcd(num1, num2))
```

Check whether a number is palindrome or not

```
#num = 121
#palindrome = true
#if rev == num return true

num = 121
n = num
```

```
rev = 0
while(n!=0):
   rev = rev*10 + n%10
   n = n//10

if rev == num:
   print("Yes it is a palindrome")
else:
   print("No it is not a palindrome")
```

Basic Recursion

```
def print_name(name, n):
   if n <= 0:
      return
   print(name)
   print_name(name, n-1)

# Example usage:
name = "Alice"
n = 5
print_name(name, n)
```

Print 1 to N

```
def print_numbers(n):
   if n <= 0:
      return
   print_numbers(n-1)
   print(n)

# Example usage:
N = 5
print_numbers(N)
```

Print N to 1

```
def print_numbers_reverse(n):
   if n <= 0:
      return
```

```
    print(n)
    print_numbers_reverse(n-1)

# Example usage:
N = 5
print_numbers_reverse(N)
```

Sum of first N numbers

```
def sum_first_n_numbers(n):
    if n == 0:
        return 0
    return n + sum_first_n_numbers(n - 1)

# Example usage:
N = 5
result = sum_first_n_numbers(N)
print("Sum of the first", N, "numbers is:", result)
```

Factorial using recursion

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

# Example usage:
N = 5
result = factorial(N)
print("Factorial of", N, "is:", result)
```

Reverse an array

```
def reverse_array(arr):
    if len(arr) == 0:
        return []
    return [arr[-1]] + reverse_array(arr[:-1])

# Example usage:
arr = [1, 2, 3, 4, 5]
reversed_arr = reverse_array(arr)
print("Original array:", arr)
```

```
print("Reversed array:", reversed_arr)
```

Palindrome

```
def is_palindrome(s):
    # Base case: if the string is empty or has only one character, it's a palindrome
    if len(s) <= 1:
        return True
    # Compare the first and last characters
    if s[0] != s[-1]:
        return False
    # Recur on the substring excluding the first and last characters
    return is_palindrome(s[1:-1])

# Example usage:
string1 = "radar"
string2 = "hello"
print(string1, "is palindrome:", is_palindrome(string1))
print(string2, "is palindrome:", is_palindrome(string2))
```

Fibonacci Series

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Example usage:
N = 10
fib_value = fibonacci(N)
print("Fibonacci number at position", N, "is:", fib_value)
```

Hashing

```
def count_frequencies(arr):
    frequency = {}
    for element in arr:
        if element in frequency:
            frequency[element] += 1
        else:
```

```python
        frequency[element] = 1
    return frequency

# Example usage:
arr = [1, 2, 3, 4, 5, 1, 2, 3, 4, 1, 2]
frequency = count_frequencies(arr)
print("Frequencies of array elements:", frequency)
```

Sorting Techniques

Selection Sort

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Example usage:
arr = [64, 25, 12, 22, 11]
sorted_arr = selection_sort(arr)
print("Sorted array is:", sorted_arr)
```

Bubble Sort

```python
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
    # Last i elements are already in place, so we don't need to check them
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Example usage:
arr = [64, 25, 12, 22, 11]
sorted_arr = bubble_sort(arr)
```

```
print("Sorted array is:", sorted_arr)
```

Insertion Sort

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Example usage:
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        # Merge the two halves
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Check if any element was left
        while i < len(L):
```

```
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Example usage:
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Sorted array is:", arr)
```

Recursive Bubble Sort

```
def recursive_bubble_sort(arr, n):
    # Base case: If array is of length 1, it's already sorted
    if n == 1:
        return

    # One pass of bubble sort on the entire array
    for i in range(n - 1):
        if arr[i] > arr[i + 1]:
            arr[i], arr[i + 1] = arr[i + 1], arr[i]

    recursive_bubble_sort(arr, n - 1)

# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
recursive_bubble_sort(arr, len(arr))
print("Sorted array is:", arr)
```

Recursive Insertion Sort

```
def recursive_insertion_sort(arr, n):
        # Base case: If the array has only one element, it is already sorted
    if n <= 1:
        return

    # Sort the first n-1 elements
    recursive_insertion_sort(arr, n-1)

    # Insert the last element at its correct position in the sorted array
```

```
    last_element = arr[n-1]
    j = n-2

    # Move elements of arr[0..n-1] that are greater than last_element to one position ahead
of their current position
    while j >= 0 and arr[j] > last_element:
        arr[j+1] = arr[j]
        j -= 1

    arr[j+1] = last_element

# Example usage:
arr = [12, 11, 13, 5, 6]
recursive_insertion_sort(arr, len(arr))
print("Sorted array is:", arr)
```

Heap Sort

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from heap one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap root with last element
        heapify(arr, i, 0)  # Heapify root element

    return arr
```

```python
# Example usage
arr = [12, 11, 13, 5, 6, 7]
sorted_arr = heap_sort(arr)
print("Sorted array is:", sorted_arr)
```

Quick Sort

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[-1]
        less_than_pivot = [x for x in arr[:-1] if x <= pivot]
        greater_than_pivot = [x for x in arr[:-1] if x > pivot]
        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print(sorted_arr)
```

Arrays

Maximum Element In An Array

```python
def find_largest_element(arr):
    if not arr:
        return None  # Handle empty array case

    max_element = arr[0]  # Initialize max_element with the first element

    for num in arr[1:]:  # Iterate through the array starting from the second element
        if num > max_element:
            max_element = num  # Update max_element if a larger element is found

    return max_element

# Example usage:
array = [10, 5, 20, 8, 15]
largest = find_largest_element(array)
print("The largest element in the array is:", largest)
```

Second Largest Element

```
def find_second_largest(arr):
    if len(arr) < 2:
        return None  # Handle case where array has less than two elements

    max_element = float('-inf')  # Initialize max_element with negative infinity
    second_max_element = float('-inf')  # Initialize second_max_element with negative
infinity

    for num in arr:
        if num > max_element:
            second_max_element = max_element
            max_element = num
        elif num > second_max_element and num != max_element:
            second_max_element = num

    return second_max_element

# Example usage:
array = [10, 5, 20, 8, 15]
second_largest = find_second_largest(array)
print("The second largest element in the array is:", second_largest)
```

Check if the array is sorted

```
def is_sorted(arr):
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))

# Example usage:
arr1 = [1, 2, 3, 4, 5]
arr2 = [5, 3, 2, 1, 0]

print(is_sorted(arr1))  # Output: True
print(is_sorted(arr2))  # Output: False
```

Remove Duplicates From Array

```
def remove_duplicates(arr):
    if len(arr) == 0:
        return 0
```

```python
    # Index to store the next unique element
    unique_index = 0

    # Iterate through the array starting from the second element
    for i in range(1, len(arr)):
        # If the current element is different from the previous unique element
        if arr[i] != arr[unique_index]:
            unique_index += 1
            # Copy the current element to the next unique index
            arr[unique_index] = arr[i]

    # The length of the unique elements is one more than the unique index
    return unique_index + 1, arr[:unique_index + 1]

# Example usage:
arr = [1, 1, 2, 2, 3, 4, 4, 5, 5]
new_length, unique_arr = remove_duplicates(arr)
print("Length of the array after removing duplicates:", new_length)
print("Array with duplicates removed:", unique_arr)
```

Left Rotate an array by one place

```python
def left_rotate_by_one(arr):
    # Store the first element of the array
    first_element = arr[0]

    # Shift all elements one position to the left
    for i in range(len(arr) - 1):
        arr[i] = arr[i + 1]

    # Assign the first element to the last position
    arr[-1] = first_element

# Example usage:
arr = [1, 2, 3, 4, 5]
print("Original array:", arr)
left_rotate_by_one(arr)
print("Array after left rotation by one place:", arr)
```

Left rotate an array by D places

```python
def left_rotate_by_d(arr, d):
    # Calculate the effective rotation amount
```

```
    effective_rotation = d % len(arr)

    # Perform left rotation using array slicing
    arr[:] = arr[effective_rotation:] + arr[:effective_rotation]

# Example usage:
arr = [1, 2, 3, 4, 5]
d = 2
print("Original array:", arr)
left_rotate_by_d(arr, d)
print("Array after left rotation by", d, "places:", arr)
```

Move Zeros to end

```
def move_zeros_to_end(arr):
    # Initialize a pointer to track the position for non-zero elements
    non_zero_pos = 0

    # Iterate through the array
    for num in arr:
        # If the current element is non-zero, move it to the front
        if num != 0:
            arr[non_zero_pos] = num
            non_zero_pos += 1

    # Fill the remaining positions with zeros
    while non_zero_pos < len(arr):
        arr[non_zero_pos] = 0
        non_zero_pos += 1

# Example usage:
arr = [0, 1, 0, 3, 12]
print("Original array:", arr)
move_zeros_to_end(arr)
print("Array after moving zeros to end:", arr)
```

Linear Search

```
def linear_search(arr, target):
```

```
    # Iterate through each element in the array
    for i in range(len(arr)):
        # If the current element matches the target, return its index
        if arr[i] == target:
            return i
    # If the target is not found, return -1
    return -1

# Example usage:
arr = [4, 2, 7, 1, 9, 5]
target = 7
index = linear_search(arr, target)
if index != -1:
    print("Target", target, "found at index:", index)
else:
    print("Target", target, "not found in the array.")
```

Find The Union

```
def find_union(arr1, arr2):
    # Convert arrays to sets to remove duplicates
    set1 = set(arr1)
    set2 = set(arr2)

    # Find the union of the two sets
    union_set = set1.union(set2)

    # Convert the union set back to a list
    union_list = list(union_set)

    return union_list

# Example usage:
arr1 = [1, 2, 3, 4, 5]
arr2 = [4, 5, 6, 7, 8]
union = find_union(arr1, arr2)
print("Union of arrays:", union)
```

Find missing number in an array

```
def find_missing_number(arr):
    # Calculate the expected sum of numbers in the range
    n = len(arr) + 1
```

```python
    expected_sum = n * (n + 1) // 2

    # Calculate the actual sum of numbers in the array
    actual_sum = sum(arr)

    # The missing number is the difference between the expected sum and the actual sum
    missing_number = expected_sum - actual_sum

    return missing_number

# Example usage:
arr = [1, 2, 4, 5, 6]
missing_number = find_missing_number(arr)
print("Missing number in the array:", missing_number)
```

Maximum Consecutive Ones

```python
def max_consecutive_ones(arr):
    max_ones = 0
    current_ones = 0

    for num in arr:
        if num == 1:
            current_ones += 1
            max_ones = max(max_ones, current_ones)
        else:
            current_ones = 0

    return max_ones

# Example usage:
arr = [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
max_ones = max_consecutive_ones(arr)
print("Maximum consecutive ones:", max_ones)
```

Find the number that appears once, and other numbers twice.

```python
def find_single_number(arr):
    # Initialize the result to 0
    result = 0

    # XOR all the numbers in the array
    for num in arr:
```

```
        result ^= num

    return result

# Example usage:
arr = [4, 3, 2, 4, 1, 2, 3]
single_number = find_single_number(arr)
print("Number that appears once:", single_number)
```

Longest subarray with given sum K(positives)

```
def longest_subarray_with_sum(arr, K):
    # Initialize variables
    max_length = 0
    current_sum = 0
    left = 0

    # Iterate through the array
    for right in range(len(arr)):
        # Add the current element to the current sum
        current_sum += arr[right]

        # Shrink the window from the left until the current sum is greater than K
        while current_sum > K:
            current_sum -= arr[left]
            left += 1

        # Update the maximum length if the current length is greater
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
arr = [1, 2, 1, 3, 4]
K = 4
max_length = longest_subarray_with_sum(arr, K)
print("Longest subarray length with sum", K, ":", max_length)
```

Longest subarray with sum K (Positives + Negatives)

```

def longest_subarray_with_sum(arr, K):
    # Initialize variables
    max_length = 0
    current_sum = 0
```

```python
    sum_index_map = {}

    # Iterate through the array
    for i in range(len(arr)):
        # Calculate the current sum
        current_sum += arr[i]

        # If the current sum equals K, update the maximum length
        if current_sum == K:
            max_length = i + 1

        # If the difference (current sum - K) exists in the map, update the maximum length
        if current_sum - K in sum_index_map:
            max_length = max(max_length, i - sum_index_map[current_sum - K])

        # If the current sum is not in the map, add it with its index
        if current_sum not in sum_index_map:
            sum_index_map[current_sum] = i

    return max_length

# Example usage:
arr = [1, -1, 5, 2, 3]
K = 3
max_length = longest_subarray_with_sum(arr, K)
print("Longest subarray length with sum", K, ":", max_length)
```

Two Sum

```python
def two_sum(nums, target):
    # Create a hashmap to store the index of each number
    num_index_map = {}

    # Iterate through the array
    for i, num in enumerate(nums):
        # Calculate the complement needed to reach the target
        complement = target - num

        # If the complement exists in the map, return its index along with the current index
        if complement in num_index_map:
            return [num_index_map[complement], i]

        # Otherwise, add the current number and its index to the map
        num_index_map[num] = i
```

```
    # If no solution is found, return None
    return None

# Example usage:
nums = [2, 7, 11, 15]
target = 9
indices = two_sum(nums, target)
if indices is not None:
    print("Indices of the two numbers:", indices)
    print("Numbers:", nums[indices[0]], nums[indices[1]])
else:
    print("No two numbers in the array sum up to the target.")
```

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

```
def sort_colors(nums):
    # Initialize pointers for the three partitions
    low = 0
    mid = 0
    high = len(nums) - 1

    # Iterate until mid crosses over high
    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:
            # Move mid pointer to the right
            mid += 1
        else:
            # Swap nums[mid] and nums[high]
            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]
sorted_nums = sort_colors(nums)
```

```
print("Sorted array:", sorted_nums)
```

Majority Element (>n/2 times)

```
def majority_element(nums):
    # Initialize the candidate and count
    candidate = None
    count = 0

    # Find the candidate for the majority element
    for num in nums:
        if count == 0:
            candidate = num
            count = 1
        elif num == candidate:
            count += 1
        else:
            count -= 1

    # Verify if the candidate is the majority element
    count = 0
    for num in nums:
        if num == candidate:
            count += 1

    if count > len(nums) // 2:
        return candidate
    else:
        return None

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
majority = majority_element(nums)
if majority is not None:
    print("Majority element:", majority)
else:
    print("No majority element found.")
```

Kadane's Algorithm, maximum subarray sum

```
def max_subarray_sum(nums):
    max_ending_here = max_so_far = nums[0]
```

```python
    for num in nums[1:]:
        max_ending_here = max(num, max_ending_here + num)
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far

# Example usage:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
max_sum = max_subarray_sum(nums)
print("Maximum subarray sum:", max_sum)
```

Print subarray with maximum subarray sum (extended version of above problem)

```python
def max_subarray_sum_with_subarray(nums):
    max_ending_here = max_so_far = nums[0]
    start = end = temp_start = 0

    for i in range(1, len(nums)):
        if nums[i] > max_ending_here + nums[i]:
            temp_start = i
            max_ending_here = nums[i]
        else:
            max_ending_here += nums[i]

        if max_ending_here > max_so_far:
            start = temp_start
            end = i
            max_so_far = max_ending_here

    return nums[start:end + 1]

# Example usage:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
max_subarray = max_subarray_sum_with_subarray(nums)
print("Maximum subarray:", max_subarray)
```

Stock Buy and Sell

```python
def max_profit(prices):
    if not prices or len(prices) == 1:
        return 0

    min_price = prices[0]
```

```
    max_profit = 0

    for price in prices[1:]:
        max_profit = max(max_profit, price - min_price)
        min_price = min(min_price, price)

    return max_profit

# Example usage:
prices = [7, 1, 5, 3, 6, 4]
profit = max_profit(prices)
print("Maximum profit:", profit)
```

Rearrange the array in alternating positive and negative items

```
def rearrange_alternating(nums):
    # Separate positive and negative numbers
    pos_nums = [num for num in nums if num >= 0]
    neg_nums = [num for num in nums if num < 0]

    # Initialize indices for positive and negative numbers
    pos_index = neg_index = 0

    # Start with positive number if the first number is positive
    if nums[0] >= 0:
        result = [pos_nums[pos_index]]
        pos_index += 1
    else:
        result = [neg_nums[neg_index]]
        neg_index += 1

    # Alternate positive and negative numbers
    while pos_index < len(pos_nums) and neg_index < len(neg_nums):
        if len(result) % 2 == 0:
            result.append(pos_nums[pos_index])
            pos_index += 1
        else:
            result.append(neg_nums[neg_index])
            neg_index += 1

    # Add remaining positive numbers if any
    while pos_index < len(pos_nums):
        result.append(pos_nums[pos_index])
        pos_index += 1
```

```
    # Add remaining negative numbers if any
    while neg_index < len(neg_nums):
        result.append(neg_nums[neg_index])
        neg_index += 1

    return result

# Example usage:
nums = [-1, 2, -3, 4, -5, 6, -7, 8]
rearranged_nums = rearrange_alternating(nums)
print("Rearranged array with alternating positive and negative items:", rearranged_nums)
```

Next Permutation

```
def next_permutation(nums):
    # Find the first element from the right that is smaller than the element next to it
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1

    # If no such element is found, the sequence is in descending order, so reverse it
    if i == -1:
        nums.reverse()
        return

    # Find the smallest element from the right that is greater than nums[i]
    j = len(nums) - 1
    while nums[j] <= nums[i]:
        j -= 1

    # Swap nums[i] and nums[j]
    nums[i], nums[j] = nums[j], nums[i]

    # Reverse the sequence from i+1 to the end
    nums[i + 1:] = reversed(nums[i + 1:])

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

Leaders in an Array problem

```
def find_leaders(arr):
```

```python
    n = len(arr)
    leaders = []
    max_right = float('-inf')

    # Traverse the array from right to left
    for i in range(n - 1, -1, -1):
        if arr[i] > max_right:
            leaders.append(arr[i])
            max_right = arr[i]

    # Reverse the list to get the leaders in correct order
    leaders.reverse()
    return leaders

# Example usage:
arr = [16, 17, 4, 3, 5, 2]
print("Leaders in the array:", find_leaders(arr))
```

Longest Consecutive Sequence in an Array

```python
def longest_consecutive_sequence(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:  # Check consecutive numbers
                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("Longest consecutive sequence length:", longest_consecutive_sequence(nums))
```

Set Matrix Zeros

```
def set_zeros(matrix):
    if not matrix:
        return

    rows_to_zero = set()
    cols_to_zero = set()

    # Identify rows and columns containing zeros
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 0:
                rows_to_zero.add(i)
                cols_to_zero.add(j)

    # Set rows to zero
    for row in rows_to_zero:
        for j in range(len(matrix[0])):
            matrix[row][j] = 0

    # Set columns to zero
    for col in cols_to_zero:
        for i in range(len(matrix)):
            matrix[i][col] = 0

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 8, 9]
]
set_zeros(matrix)
for row in matrix:
    print(row)
```

Rotate Matrix by 90 degrees

```
def rotate(matrix):
    if not matrix:
        return

    n = len(matrix)

    # Transpose the matrix
    for i in range(n):
```

```python
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # Reverse each row
    for i in range(n):
        matrix[i] = matrix[i][::-1]

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
rotate(matrix)
for row in matrix:
    print(row)
```

Print the matrix in spiral manner

```python
def spiral_order(matrix):
    if not matrix:
        return []

    result = []
    rows, cols = len(matrix), len(matrix[0])
    top, bottom, left, right = 0, rows - 1, 0, cols - 1

    while top <= bottom and left <= right:
        # Traverse top row
        for j in range(left, right + 1):
            result.append(matrix[top][j])
        top += 1

        # Traverse right column
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1

        # Traverse bottom row (if top hasn't crossed bottom)
        if top <= bottom:
            for j in range(right, left - 1, -1):
                result.append(matrix[bottom][j])
            bottom -= 1

        # Traverse left column (if left hasn't crossed right)
        if left <= right:
```

```
        for i in range(bottom, top - 1, -1):
            result.append(matrix[i][left])
        left += 1

    return result

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Matrix in spiral order:", spiral_order(matrix))
```

Count subarrays with given sum

```
def count_subarrays_with_sum(nums, target):
    count = 0
    sum_frequency = {0: 1}
    current_sum = 0

    for num in nums:
        current_sum += num
        # Check if there is a subarray ending at the current index with the given sum
        if current_sum - target in sum_frequency:
            count += sum_frequency[current_sum - target]
        # Update the frequency of the current sum
        sum_frequency[current_sum] = sum_frequency.get(current_sum, 0) + 1

    return count

# Example usage:
nums = [1, 2, 3, 4, 5]
target = 9
print("Number of subarrays with sum", target, ":", count_subarrays_with_sum(nums,
target))
```

Pascal's Triangle

```

def generate_pascals_triangle(num_rows):
    if num_rows <= 0:
        return []

    triangle = [[1]]
```

```
    for i in range(1, num_rows):
        prev_row = triangle[-1]
        current_row = [1]

        for j in range(1, i):
            current_row.append(prev_row[j - 1] + prev_row[j])

        current_row.append(1)
        triangle.append(current_row)

    return triangle

# Example usage:
num_rows = 5
pascals_triangle = generate_pascals_triangle(num_rows)
for row in pascals_triangle:
    print(row)
```

Majority Element (n/3 times)

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

    count1 = count2 = 0
    candidate1 = candidate2 = None

    # Step 1: Find 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: Count occurrences of candidates
    count1 = count2 = 0
    for num in nums:
        if num == candidate1:
            count1 += 1
        elif num == candidate2:
            count2 += 1

    # Step 3: Check if candidates appear more than n/3 times
    result = []
    if count1 > len(nums) // 3:
        result.append(candidate1)
    if count2 > len(nums) // 3:
        result.append(candidate2)

    return result

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
print("Majority elements (appearing more than n/3 times):", majority_element(nums))
```

3-Sum Problem

```
def three_sum(nums, target):
    nums.sort()  # Sort the array to use two-pointer approach
    result = []

    for i in range(len(nums) - 2):  # Fix the first element
        if i > 0 and nums[i] == nums[i - 1]:
            continue  # Skip duplicates
        left, right = i + 1, len(nums) - 1  # Use two pointers approach
        while left < right:
            curr_sum = nums[i] + nums[left] + nums[right]
            if curr_sum == target:
                result.append([nums[i], nums[left], nums[right]])
                # Skip duplicates for left and right pointers
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1
                left += 1
                right -= 1
            elif curr_sum < target:
                left += 1
            else:
```

```python
            right -= 1

    return result

# Example usage:
nums = [-1, 0, 1, 2, -1, -4]
target = 0
print("Unique triplets with sum equal to", target, ":", three_sum(nums, target))
```

4-Sum Problem

```python
def fourSum(nums, target):
    nums.sort()
    result = []
    n = len(nums)
    for i in range(n - 3):
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        for j in range(i + 1, n - 2):
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue
            left, right = j + 1, n - 1
            while left < right:
                total = nums[i] + nums[j] + nums[left] + nums[right]
                if total == target:
                    result.append([nums[i], nums[j], nums[left], nums[right]])
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1
                    left += 1
                    right -= 1
                elif total < 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]]
```

Largest Subarray with 0 Sum

```python
def largest_subarray_with_zero_sum(arr):
    max_len = 0
    sum_map = {}
    curr_sum = 0

    for i in range(len(arr)):
        curr_sum += arr[i]

        if arr[i] == 0 and max_len == 0:
            max_len = 1

        if curr_sum == 0:
            max_len = i + 1

        if curr_sum in sum_map:
            max_len = max(max_len, i - sum_map[curr_sum])
        else:
            sum_map[curr_sum] = i

    return max_len

# Example usage:
arr = [15, -2, 2, -8, 1, 7, 10, 23]
print("Length of the largest subarray with zero sum:",
largest_subarray_with_zero_sum(arr))
```

Count number of subarrays with given xor K

```python
def count_subarrays_with_xor(arr, K):
    xor_count = {}
    xor_sum = 0
    count = 0

    for num in arr:
        xor_sum ^= num

        if xor_sum == K:
            count += 1

        if xor_sum ^ K in xor_count:
            count += xor_count[xor_sum ^ K]

        xor_count[xor_sum] = xor_count.get(xor_sum, 0) + 1
```

```
    return count

# Example usage:
arr = [4, 2, 2, 6, 4]
K = 6
print("Number of subarrays with XOR", K, ":", count_subarrays_with_xor(arr, K))
```

Merge Overlapping Subintervals

```
def merge_overlapping_intervals(intervals):
    if not intervals:
        return []

    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for i in range(1, len(intervals)):
        current_start, current_end = intervals[i]
        previous_start, previous_end = merged[-1]

        if current_start <= previous_end:
            merged[-1] = [previous_start, max(previous_end, current_end)]
        else:
            merged.append([current_start, current_end])

    return merged

# Example usage:
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
merged_intervals = merge_overlapping_intervals(intervals)
print("Merged Intervals:", merged_intervals)
```

Merge two sorted arrays without extra space

```
def merge_sorted_arrays(arr1, arr2):
    m = len(arr1)
    n = len(arr2)
    arr1.extend([float('inf')] * n)  # Extend arr1 with 'inf' to accommodate merged elements
    i = m - 1  # Index of last element in arr1
    j = n - 1  # Index of last element in arr2
    k = m + n - 1  # Index of last element in merged array

    while i >= 0 and j >= 0:
        if arr1[i] > arr2[j]:
```

```
            arr1[k] = arr1[i]
            i -= 1
        else:
            arr1[k] = arr2[j]
            j -= 1
        k -= 1

    # If elements remaining in arr2, copy them to arr1
    while j >= 0:
        arr1[k] = arr2[j]
        j -= 1
        k -= 1

# Example usage:
arr1 = [1, 3, 5, 7]
arr2 = [2, 4, 6]
merge_sorted_arrays(arr1, arr2)
print("Merged Array:", arr1)
```

Find the repeating and missing number

```
def find_repeating_missing(nums):
    n = len(nums)
    total_sum = (n * (n + 1)) // 2
    arr_sum = sum(nums)

    # Find the repeating number
    repeating = arr_sum - total_sum + sum(set(nums))

    # Find the missing number
    missing = repeating + total_sum - arr_sum

    return repeating, missing

# Example usage:
nums = [3, 1, 2, 5, 3]  # Example array
repeating, missing = find_repeating_missing(nums)
print("Repeating number:", repeating)
print("Missing number:", missing)
```

Count Inversions

```
def merge_sort(arr):
    if len(arr) <= 1:
```

```
        return arr, 0

    mid = len(arr) // 2
    left, left_inv = merge_sort(arr[:mid])
    right, right_inv = merge_sort(arr[mid:])
    merged, merge_inv = merge(left, right)

    return merged, left_inv + right_inv + merge_inv

def merge(left, right):
    merged = []
    inversions = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inversions += len(left) - i  # Count inversions

    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged, inversions

# Example usage:
arr = [1, 3, 5, 2, 4, 6]  # Example array
sorted_arr, inversions = merge_sort(arr)
print("Sorted array:", sorted_arr)
print("Number of inversions:", inversions)
```

Reverse Pairs

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr, 0

    mid = len(arr) // 2
    left, count_left = merge_sort(arr[:mid])
    right, count_right = merge_sort(arr[mid:])

    merged = []
    count = count_left + count_right
    i = j = 0
```

```python
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            count += len(left) - i  # Increment count by the remaining elements in left
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged, count

def count_reverse_pairs(arr):
    _, count = merge_sort(arr)
    return count

# Example usage:
arr = [1, 3, 2, 4, 5]
print("Reverse pairs in", arr, ":", count_reverse_pairs(arr))
```

Maximum Product Subarray

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

    # Initialize variables to track maximum and minimum product ending at the current index
    max_prod = min_prod = result = nums[0]

    for num in nums[1:]:
        # If the current number is negative, swap max_prod and min_prod
        # This is because multiplying a negative number by a negative number yields a positive
number
        if num < 0:
            max_prod, min_prod = min_prod, max_prod

        # Update the maximum and minimum products ending at the current index
        max_prod = max(num, max_prod * num)
        min_prod = min(num, min_prod * num)

        # Update the overall maximum product
        result = max(result, max_prod)

    return result
```

```python
# Example usage:
nums = [2, 3, -2, 4]
print("Maximum product subarray:", max_product_subarray(nums))
```

Binary Search

Binary Search to find X in sorted array

```python
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

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

        # Check if target is present at mid
        if arr[mid] == target:
            return mid

        # If target is greater, ignore left half
        elif arr[mid] < target:
            left = mid + 1

        # If target is smaller, ignore right half
        else:
            right = mid - 1

    # If target is not found in the array
    return -1

# Example usage:
arr = [1, 2, 3, 4, 5, 6, 7]
target = 5
result = binary_search(arr, target)
if result != -1:
    print("Element", target, "is present at index:", result)
else:
    print("Element", target, "is not present in the array.")
```

Implement Lower Bound

```python
def lower_bound(arr, target):
    left = 0
    right = len(arr)
```

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

        # If arr[mid] is less than target, search in the right half
        if arr[mid] < target:
            left = mid + 1
        # If arr[mid] is greater than or equal to target, search in the left half
        else:
            right = mid

    return left

# Example usage:
arr = [1, 2, 3, 4, 4, 4, 6, 7]
target = 4
lower_bound_index = lower_bound(arr, target)
print("Lower bound index of", target, "in the array is:", lower_bound_index)
```

Implement Upper Bound

```
def upper_bound(arr, target):
    left = 0
    right = len(arr)

    while left < right:
        mid = left + (right - left) // 2
    # If arr[mid] is less than or equal to target, search in the right half
        if arr[mid] <= target:
            left = mid + 1
        # If arr[mid] is greater than target, search in the left half
        else:
            right = mid

    return left

# Example usage:
arr = [1, 2, 3, 4, 4, 4, 6, 7]
target = 4
upper_bound_index = upper_bound(arr, target)
print("Upper bound index of", target, "in the array is:", upper_bound_index)
```

Search Insert Position

```
def search_insert_position(nums, target):
```

```
    left = 0
    right = len(nums)

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

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid

    return left

# Example usage:
nums = [1, 3, 5, 6]
target = 5
insert_position = search_insert_position(nums, target)
print("Insert position of", target, "in the array is:", insert_position)
```

Floor/Ceil in Sorted Array

```
def find_floor(arr, target):
    left = 0
    right = len(arr) - 1
    floor = -1

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

        if arr[mid] == target:
            return arr[mid]
        elif arr[mid] < target:
            floor = arr[mid]
            left = mid + 1
        else:
            right = mid - 1

    return floor

def find_ceiling(arr, target):
    left = 0
    right = len(arr) - 1
    ceiling = float('inf')
```

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

        if arr[mid] == target:
            return arr[mid]
        elif arr[mid] < target:
            left = mid + 1
        else:
            ceiling = arr[mid]
            right = mid - 1

    return ceiling

# Example usage:
arr = [1, 2, 8, 10, 10, 12, 19]
target = 5
print("Floor of", target, "in the array is:", find_floor(arr, target))
print("Ceiling of", target, "in the array is:", find_ceiling(arr, target))
```

Find the first or last occurrence of a given number in a sorted array

```python
def find_first_occurrence(arr, target):
    left = 0
    right = len(arr) - 1
    first_occurrence = -1

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

        if arr[mid] == target:
            first_occurrence = mid
            right = mid - 1  # Continue searching for first occurrence in the left half
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return first_occurrence

def find_last_occurrence(arr, target):
    left = 0
    right = len(arr) - 1
    last_occurrence = -1

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

```python
        if arr[mid] == target:
            last_occurrence = mid
            left = mid + 1  # Continue searching for last occurrence in the right half
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return last_occurrence

# Example usage:
arr = [1, 2, 3, 4, 4, 4, 6, 7]
target = 4
print("First occurrence of", target, "in the array is at index:", find_first_occurrence(arr,
target))
print("Last occurrence of", target, "in the array is at index:", find_last_occurrence(arr,
target))
```

Count occurrences of a number in a sorted array with duplicates

```python
def count_occurrences(arr, target):
    def first_occurrence(arr, target):
        left, right = 0, len(arr) - 1
        result = -1
        while left <= right:
            mid = left + (right - left) // 2
            if arr[mid] == target:
                result = mid
                right = mid - 1
            elif arr[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return result

    def last_occurrence(arr, target):
        left, right = 0, len(arr) - 1
        result = -1
        while left <= right:
            mid = left + (right - left) // 2
            if arr[mid] == target:
                result = mid
                left = mid + 1
            elif arr[mid] < target:
                left = mid + 1
```

```python
            else:
                right = mid - 1
        return result

    first_index = first_occurrence(arr, target)
    last_index = last_occurrence(arr, target)

    if first_index == -1 or last_index == -1:
        return 0
    return last_index - first_index + 1

# Example usage:
arr = [1, 2, 2, 3, 3, 3, 4, 5, 5]
target = 3
print("Number of occurrences of", target, ":", count_occurrences(arr, target))  # Output: 3
```

Search in Rotated Sorted Array I

```python
def search(nums, target):
    if not nums:
        return -1

    left, right = 0, len(nums) - 1

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

        if nums[mid] == target:
            return mid

        # Check if the left half is sorted
        if nums[left] <= nums[mid]:
            # Check if the target is in the left half
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Otherwise, the right half must be sorted
        else:
            # Check if the target is in the right half
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

```
# Example usage:
nums = [4, 5, 6, 7, 0, 1, 2]
target = 0
print("Index of", target, ":", search(nums, target))  # Output: 4
```

Search in Rotated Sorted Array II

```
def search(nums, target):
    left, right = 0, len(nums) - 1

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

        # If the middle element is the target, return its index
        if nums[mid] == target:
            return True

        # Handling the case when there are duplicates at start or end
        while left < mid and nums[left] == nums[mid]:
            left += 1
        while right > mid and nums[right] == nums[mid]:
            right -= 1

        # If left half is sorted
        if nums[left] <= nums[mid]:
            # Check if the target is within the left half
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # If right half is sorted
        else:
            # Check if the target is within the right half
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return False

# Example usage:
nums = [2, 5, 6, 0, 0, 1, 2]
target = 0
print("Is", target, "in the array?", search(nums, target))  # Output: True
```

Find minimum in Rotated Sorted Array

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

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

        # If the middle element is greater than the last element,
        # the minimum element must be in the right half.
        if nums[mid] > nums[right]:
            left = mid + 1
        # If the middle element is less than or equal to the last element,
        # the minimum element must be in the left half (including the middle element).
        else:
            right = mid

    # At this point, left and right will converge to the index of the minimum element.
    # Return the value at that index.
    return nums[left]

# Example usage:
nums = [4, 5, 6, 7, 0, 1, 2]
print("Minimum element:", find_min(nums))  # Output: 0
```

Find out how many times has an array been rotated

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

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

        # If the middle element is greater than the last element,
        # the rotation must have occurred after the middle element.
        if nums[mid] > nums[right]:
            left = mid + 1
        # If the middle element is less than or equal to the last element,
        # the rotation must have occurred before or at the middle element.
        else:
            right = mid

    # The number of rotations corresponds to the index of the minimum element.
    return left

# Example usage:
```

```python
nums = [4, 5, 6, 7, 0, 1, 2]
print("Number of rotations:", count_rotations(nums))  # Output: 4
```

Single element in a Sorted Array

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

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

        # Ensure mid is always at even index
        if mid % 2 == 1:
            mid -= 1

        # Check if the single element is on the left or right side
        if nums[mid] != nums[mid + 1]:
            right = mid
        else:
            left = mid + 2
    return nums[left]

# Example usage:
nums = [1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 6]
print("Single element:", singleNonDuplicate(nums))  # Output: 2
```

Find peak element

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

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

        # Check if mid is a peak
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
nums = [1, 2, 1, 3, 5, 6, 4]
print("Index of peak element:", findPeakElement(nums))  # Output: Index of 5 or 6
(depending on the array)
```

Find square root of a number in log n

```
def sqrt_binary_search(x):
    if x == 0 or x == 1:
        return x

    left, right = 1, x // 2
    result = 0

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

        # If the square of mid is equal to x, return mid
        if mid * mid == x:
            return mid

        # If mid*mid is less than x, search in the right half
        if mid * mid < x:
            left = mid + 1
            result = mid
        else:
            right = mid - 1
    return result

# Example usage:
num = 16
print("Square root of", num, ":", sqrt_binary_search(num))  # Output: 4
```

Find the Nth root of a number using binary search

```
def nth_root_binary_search(x, n, precision=0.000001):
    if x == 0:
        return 0
    left, right = 0, x
    result = 0

    while abs(right - left) > precision:
        mid = (left + right) / 2
      # Check if mid raised to power n is approximately equal to x
        if abs(mid ** n - x) < precision:
            return mid

      # If mid raised to power n is less than x, search in the right half
        if mid ** n < x:
            left = mid
            result = mid
```

```
        # If mid raised to power n is greater than x, search in the left half
        else:
            right = mid
    return result

# Example usage:
x = 16
n = 4
print("Nth root of", x, ":", nth_root_binary_search(x, n))  # Output: 2.0
```

Koko Eating Bananas

```
def minEatingSpeed(piles, H):
    def possible(K):
        return sum((p - 1) // K + 1 for p in piles) <= H

    left, right = 1, max(piles)
    while left < right:
        mid = left + (right - left) // 2
        if not possible(mid):
            left = mid + 1
        else:
            right = mid
    return left

# Example usage:
piles = [3, 6, 7, 11]
H = 8
print("Minimum bananas per hour:", minEatingSpeed(piles, H))  # Output: 4
```

Minimum days to make M bouquets

```
def minDays(bloomDay, m, k):
    def check(days):
        bouquets = flowers = 0
        for bloom in bloomDay:
            if bloom > days:
                flowers = 0
            else:
                flowers += 1
                if flowers == k:
                    bouquets += 1
                    flowers = 0
        return bouquets >= m

    if len(bloomDay) < m * k:
```

```
        return -1

    left, right = min(bloomDay), max(bloomDay)
    while left < right:
        mid = left + (right - left) // 2
        if check(mid):
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
bloomDay = [1, 10, 3, 10, 2]
m = 3
k = 1
print("Minimum days to make", m, "bouquets with", k, "flowers each:",
minDays(bloomDay, m, k))  # Output: 3
```

Find the smallest Divisor

```
def smallestDivisor(nums, threshold):
    def feasible(divisor):
        return sum((num - 1) // divisor + 1 for num in nums) <= threshold

    left, right = 1, max(nums)
    while left < right:
        mid = left + (right - left) // 2
        if feasible(mid):
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
nums = [1, 2, 5, 9]
threshold = 6
print("Smallest divisor:", smallestDivisor(nums, threshold))  # Output: 5
```

Capacity to Ship Packages within D Days

```
def shipWithinDays(weights, D):
    def feasible(capacity):
        days_needed = 1
        current_weight = 0
```

```
        for weight in weights:
            current_weight += weight
            if current_weight > capacity:
                days_needed += 1
                current_weight = weight
        return days_needed <= D

    left, right = max(weights), sum(weights)
    while left < right:
        mid = left + (right - left) // 2
        if feasible(mid):
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
D = 5
print("Minimum capacity required:", shipWithinDays(weights, D))  # Output: 15
```

Kth Missing Positive Number

```
def findKthMissing(arr, k):
    missing_count = 0
    n = len(arr)

    for i in range(n):
        if arr[i] - missing_count - 1 >= k:
            return missing_count + k
        else:
            missing_count += arr[i] - missing_count - 1

    return arr[-1] + k - missing_count

# Example usage:
arr = [2, 3, 4, 7, 11]
k = 5
print("The", k, "th missing positive number is:", findKthMissing(arr, k))
```

Aggressive Cows

```
def aggressive_cows(stalls, cows):
    stalls.sort()
```

```python
    min_dist = 0
    max_dist = stalls[-1] - stalls[0]

    while min_dist <= max_dist:
        mid_dist = (min_dist + max_dist) // 2
        placed_cows = 1
        prev_stall = stalls[0]

        for i in range(1, len(stalls)):
            if stalls[i] - prev_stall >= mid_dist:
                placed_cows += 1
                prev_stall = stalls[i]

        if placed_cows >= cows:
            min_dist = mid_dist + 1
        else:
            max_dist = mid_dist - 1

    return max_dist

# Example usage:
stalls = [1, 2, 4, 8, 9]
cows = 3
print("The largest minimum distance is:", aggressive_cows(stalls, cows))
```

Book Allocation Problem

```python
def allocate_books(books, num_students):
    if len(books) < num_students:
        return -1  # Not enough books to allocate to all students

    books.sort()  # Sort the books in ascending order of their pages

    min_pages = float('inf')
    left, right = 0, sum(books)

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

        if is_valid(books, num_students, mid):
            min_pages = min(min_pages, mid)
            right = mid - 1
        else:
            left = mid + 1

    return min_pages
```

```python
def is_valid(books, num_students, max_pages):
    num_allocated_students = 1
    pages_read = 0

    for pages in books:
        if pages_read + pages > max_pages:
            num_allocated_students += 1
            pages_read = 0
            if num_allocated_students > num_students:
                return False
        pages_read += pages

    return True

# Example usage:
books = [10, 20, 30, 40]
num_students = 2
print("Minimum pages each student must read:", allocate_books(books, num_students))
```

Split array - Largest Sum

```python
def splitArray(nums, m):
    def valid(mid):
        count, curr_sum = 1, 0
        for num in nums:
            curr_sum += num
            if curr_sum > mid:
                count += 1
                curr_sum = num
        return count <= m

    left, right = max(nums), sum(nums)
    while left < right:
        mid = left + (right - left) // 2
        if valid(mid):
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
nums = [7,2,5,10,8]
m = 2
print(splitArray(nums, m))  # Output will be 18
```

Painter's partition

```
def sum_subarray(arr, start, end):
    total = 0
    for i in range(start, end + 1):
        total += arr[i]
    return total

def min_painters_partition(arr, n, k):
    dp = [[float('inf')] * (k + 1) for _ in range(n + 1)]
    prefix_sum = [0] * (n + 1)
    prefix_sum[0] = arr[0]
    for i in range(1, n):
        prefix_sum[i] = prefix_sum[i - 1] + arr[i]

    for i in range(n):
        dp[i][1] = prefix_sum[i]

    for i in range(1, n):
        for j in range(2, k + 1):
            for p in range(i):
                dp[i][j] = min(dp[i][j], max(dp[p][j - 1], prefix_sum[i] - prefix_sum[p]))

    return dp[n - 1][k]

# Example usage
arr = [12, 34, 67, 90]
k = 2
n = len(arr)
print("Minimum time to paint:", min_painters_partition(arr, n, k))
```

Minimize Max Distance to Gas Station

```
def min_max_distance_gas_station(stations, k):
    def feasible(distance, k):
        count = 0
        for i in range(1, len(stations)):
            count += (stations[i] - stations[i - 1] - 1) // distance
        return count <= k

    left, right = 1, stations[-1] - stations[0]
    while left < right:
        mid = (left + right) // 2
        if feasible(mid, k):
            right = mid
```

```
        else:
            left = mid + 1
    return left

# Example usage
stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
k = 9
print("Minimum maximum distance to gas station:",
min_max_distance_gas_station(stations, k))
```

Median of 2 sorted arrays

```
def findMedianSortedArrays(nums1, nums2):
    merged = []
    i, j = 0, 0

    while i < len(nums1) and j < len(nums2):
        if nums1[i] < nums2[j]:
            merged.append(nums1[i])
            i += 1
        else:
            merged.append(nums2[j])
            j += 1

    while i < len(nums1):
        merged.append(nums1[i])
        i += 1

    while j < len(nums2):
        merged.append(nums2[j])
        j += 1

    total_length = len(merged)
    if total_length % 2 == 0:
        median_index = total_length // 2
        median = (merged[median_index - 1] + merged[median_index]) / 2
    else:
        median_index = total_length // 2
        median = merged[median_index]

    return median

# Example usage:
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))  # Output: 2.0
```

Kth element of 2 sorted arrays

```
def kth_element_of_sorted_arrays(arr1, arr2, k):
    m, n = len(arr1), len(arr2)
    i, j, kth = 0, 0, 0

    while i < m and j < n:
        if arr1[i] < arr2[j]:
            kth = arr1[i]
            i += 1
        else:
            kth = arr2[j]
            j += 1
        k -= 1
        if k == 0:
            return kth

    while i < m:
        kth = arr1[i]
        i += 1
        k -= 1
        if k == 0:
            return kth

    while j < n:
        kth = arr2[j]
        j += 1
        k -= 1
        if k == 0:
            return kth

    return kth

# Example usage:
arr1 = [1, 3, 5, 7, 9]
arr2 = [2, 4, 6, 8, 10]
k = 5
print("The", k, "th element is:", kth_element_of_sorted_arrays(arr1, arr2, k))
```

Find the row with maximum number of 1's

```
def max_ones_row(matrix):
    max_ones = 0
    max_ones_row_index = -1
```

```python
    for i, row in enumerate(matrix):
        count_ones = sum(row)
        if count_ones > max_ones:
            max_ones = count_ones
            max_ones_row_index = i

    return max_ones_row_index

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

max_ones_row_index = max_ones_row(matrix)
print("Row with maximum number of 1's:", max_ones_row_index)
```

Search in a 2 D matrix

```python
def search_2d_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])
    row, col = 0, cols - 1

    while row < rows and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] < target:
            row += 1
        else:
            col -= 1

    return False

# Example usage:
matrix = [
    [1, 4, 7, 11],
    [2, 5, 8, 12],
    [3, 6, 9, 16],
    [10, 13, 14, 17]
]
```

```
target = 8
print("Is", target, "present in the matrix?", search_2d_matrix(matrix, target))
```

Search in a row and column wise sorted matrix

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

    rows, cols = len(matrix), len(matrix[0])
    row, col = 0, cols - 1

    while row < rows and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] < target:
            row += 1
        else:
            col -= 1

    return False

# Example usage:
matrix = [
    [1, 4, 7, 11],
    [2, 5, 8, 12],
    [3, 6, 9, 16],
    [10, 13, 14, 17]
]
target = 8
print("Is", target, "present in the matrix?", search_sorted_matrix(matrix, target))
```

Find Peak Element (2D Matrix)

```
def find_peak(matrix):
    def find_peak_in_column(column):
        max_row = 0
        max_val = matrix[0][column]
        for i in range(1, len(matrix)):
            if matrix[i][column] > max_val:
                max_val = matrix[i][column]
                max_row = i
        return max_row

    def binary_search(left, right):
```

```python
        if left == right:
            return find_peak_in_column(left)

        mid = (left + right) // 2
        max_row = find_peak_in_column(mid)

        if mid > 0 and matrix[max_row][mid - 1] > matrix[max_row][mid]:
            return binary_search(left, mid - 1)
        elif mid < len(matrix[0]) - 1 and matrix[max_row][mid + 1] > matrix[max_row][mid]:
            return binary_search(mid + 1, right)
        else:
            return matrix[max_row][mid]

    if not matrix or not matrix[0]:
        return None

    return binary_search(0, len(matrix[0]) - 1)

# Example usage:
matrix = [
    [1, 3, 5, 8],
    [10, 11, 12, 6],
    [9, 7, 10, 14],
    [20, 21, 23, 24]
]

peak = find_peak(matrix)
print("Peak element in the matrix:", peak)
```

Matrix Median

```python
def matrix_median(matrix):
    # Flatten the matrix into a 1D array
    flattened = [element for row in matrix for element in row]

    # Sort the array
    flattened.sort()

    # Find the length of the flattened array
    length = len(flattened)

    # Check if the length is odd or even
    if length % 2 == 0:
        # If even, return the average of the two middle elements
        middle_index = length // 2
        median = (flattened[middle_index - 1] + flattened[middle_index]) / 2
```

```
    else:
        # If odd, return the middle element
        median = flattened[length // 2]

    return median

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

print("Matrix Median:", matrix_median(matrix))
```

Strings

Remove outermost Paranthesis

```
def remove_outer_parentheses(expression):
    if expression[0] == '(' and expression[-1] == ')':
        return expression[1:-1]
    else:
        return expression

# Example usage:
expression = "(a + b) * (c - d)"
result = remove_outer_parentheses(expression)
print(result)  # Output: "a + b) * (c - d"
```

Reverse words in a given string / Palindrome Check

```
def is_palindrome(word):
    # Convert the word to lowercase to handle case-insensitivity
    word = word.lower()
    # Check if the word is equal to its reverse
    return word == word[::-1]

# Example usage:
word = "radar"
if is_palindrome(word):
    print(f"{word} is a palindrome.")
else:
    print(f"{word} is not a palindrome.")
```

Largest odd number in a string

```
import re

def largest_odd_number(string):
    # Extract all numbers from the string
    numbers = re.findall(r'\d+', string)

    # Filter out odd numbers and convert them to integers
    odd_numbers = [int(num) for num in numbers if int(num) % 2 != 0]

    if not odd_numbers:
        return "No odd numbers found."

    # Find the largest odd number
    largest_odd = max(odd_numbers)

    return largest_odd

# Example usage:
string = "There are 12345 odd numbers in this 67890 string."
result = largest_odd_number(string)
print("Largest odd number:", result)  # Output: 12345
```

Longest Common Prefix

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

    # Sort the strings to find the shortest string
    strings.sort()
    shortest = strings[0]
    longest = strings[-1]

    # Compare characters of the shortest and longest strings
    for i, char in enumerate(shortest):
        if char != longest[i]:
            return shortest[:i]

    return shortest

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

```
print("Longest common prefix:", result)  # Output: "fl"
```

Isomorphic String

```
def is_isomorphic(s, t):
    if len(s) != len(t):
        return False

    # Dictionary to store mapping of characters
    mapping = {}

    # Set to store mapped characters
    mapped_chars = set()

    for i in range(len(s)):
        char_s = s[i]
        char_t = t[i]

        # If character in s is not mapped yet
        if char_s not in mapping:
            # Check if character in t is already mapped
            if char_t in mapped_chars:
                return False
            # Map character in s to character in t
            mapping[char_s] = char_t
            mapped_chars.add(char_t)
        # If character in s is already mapped, check if mapping matches
        elif mapping[char_s] != char_t:
            return False

    return True

# Example usage:
s1 = "egg"
t1 = "add"
print(is_isomorphic(s1, t1))  # Output: True

s2 = "foo"
t2 = "bar"
print(is_isomorphic(s2, t2))  # Output: False
```

Check whether one string is a rotation of another

```
def is_rotation(s1, s2):
    # Check if the lengths of the strings are the same and not empty
```

```python
    if len(s1) != len(s2) or len(s1) == 0:
        return False

    # Concatenate s1 with itself
    s1_double = s1 + s1

    # Check if s2 is a substring of the concatenated string
    return s2 in s1_double

# Example usage:
s1 = "waterbottle"
s2 = "erbottlewat"
print(is_rotation(s1, s2))  # Output: True

s3 = "hello"
s4 = "world"
print(is_rotation(s3, s4))  # Output: False
```

Check if two strings are anagram of each other

```python
def are_anagrams(s1, s2):
    # Remove spaces and convert both strings to lowercase
    s1 = s1.replace(" ", "").lower()
    s2 = s2.replace(" ", "").lower()

    # Check if the sorted versions of the strings are the same
    return sorted(s1) == sorted(s2)

# Example usage:
s1 = "listen"
s2 = "silent"
print(are_anagrams(s1, s2))  # Output: True

s3 = "hello"
s4 = "world"
print(are_anagrams(s3, s4))  # Output: False
```

Sort Characters by frequency

```python
def frequency_sort(s):
    # Count the frequency of each character
    char_frequency = {}
    for char in s:
        char_frequency[char] = char_frequency.get(char, 0) + 1
```

```
    # Sort characters based on their frequency in descending order
    sorted_chars = sorted(char_frequency.items(), key=lambda x: x[1], reverse=True)

    # Build the sorted string
    sorted_string = ""
    for char, freq in sorted_chars:
        sorted_string += char * freq

    return sorted_string

# Example usage:
s = "tree"
print(frequency_sort(s))  # Output: "eert" or "eetr"

s = "cccaaa"
print(frequency_sort(s))  # Output: "cccaaa" or "aaaccc"

s = "Aabb"
print(frequency_sort(s))  # Output: "bbAa" or "bbaA"
```

Maximum Nesting Depth of Paranthesis

```
def max_depth(s):
    max_depth = 0
    current_depth = 0

    for char in s:
        if char == '(':
            current_depth += 1
            max_depth = max(max_depth, current_depth)
        elif char == ')':
            current_depth -= 1

    return max_depth

# Example usage:
s = "(1+(2*3)+((8)/4))+1"
print(max_depth(s))  # Output: 3
```

Roman Number to Integer

```
def roman_to_int(s: str) -> int:
    roman_to_value = {
        'I': 1, 'V': 5, 'X': 10, 'L': 50,
        'C': 100, 'D': 500, 'M': 1000
    }
```

```python
    total = 0
    prev_value = 0

    for char in reversed(s):
        value = roman_to_value[char]
        if value >= prev_value:
            total += value
        else:
            total -= value
        prev_value = value

    return total

# Example usage
print(roman_to_int('III'))     # 3
print(roman_to_int('IV'))      # 4
print(roman_to_int('IX'))      # 9
print(roman_to_int('LVIII'))   # 58
print(roman_to_int('MCMXCIV')) # 1994
```

Integer to roman

```python
def int_to_roman(num: int) -> str:
    value_to_roman = [
        (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')
    ]

    roman = []

    for value, symbol in value_to_roman:
        while num >= value:
            roman.append(symbol)
            num -= value

    return ''.join(roman)

# Example usage
print(int_to_roman(3))     # 'III'
print(int_to_roman(4))     # 'IV'
print(int_to_roman(9))     # 'IX'
print(int_to_roman(58))    # 'LVIII'
print(int_to_roman(1994))  # 'MCMXCIV'
```

Implement Atoi

```
def my_atoi(s: str) -> int:
    s = s.strip()  # remove leading and trailing whitespaces
    if not s:
        return 0

    sign = 1
    start_index = 0
    if s[0] in ['-', '+']:
        if s[0] == '-':
            sign = -1
        start_index += 1

    total = 0
    for i in range(start_index, len(s)):
        if not s[i].isdigit():
            break
        total = total * 10 + int(s[i])

    total *= sign

    # Clamp to 32-bit integer range
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
    if total > INT_MAX:
        return INT_MAX
    if total < INT_MIN:
        return INT_MIN

    return total

# Example usage
print(my_atoi("42"))          # 42
print(my_atoi("   -42"))      # -42
print(my_atoi("4193 with words"))  # 4193
print(my_atoi("words and 987"))    # 0
print(my_atoi("-91283472332"))     # -2147483648 (clamped to INT_MIN)
```

Count Number of Substrings

```
def count_all_substrings(s: str) -> int:
    n = len(s)
```

```python
    return n * (n + 1) // 2

# Example usage
print(count_all_substrings("abc"))  # Output: 6 (a, b, c, ab, bc, abc)
```

```python
def count_distinct_substrings(s: str) -> int:
    n = len(s)
    substrings = set()

    for i in range(n):
        for j in range(i + 1, n + 1):
            substrings.add(s[i:j])

    return len(substrings)

# Example usage
print(count_distinct_substrings("abc"))  # Output: 6 (a, b, c, ab, bc, abc)
```

Longest Palindromic Substring[Do it without DP]

```python

def longest_palindromic_substring(s: str) -> str:
    if len(s) == 0:
        return ""

    def expand_around_center(left: int, right: int) -> str:
        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
        odd_palindrome = expand_around_center(i, i)
        if len(odd_palindrome) > len(longest):
            longest = odd_palindrome

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

    return longest
```

```python
# Example usage
print(longest_palindromic_substring("babad"))  # Output: "bab" or "aba"
print(longest_palindromic_substring("cbbd"))   # Output: "bb"
```

Sum of Beauty of all substring

```python
from collections import defaultdict, Counter

def beauty_of_substring(s: str) -> int:
    def calculate_beauty(freq: Counter) -> int:
        max_freq = max(freq.values())
        min_freq = min(freq.values())
        return max_freq - min_freq

    n = len(s)
    total_beauty = 0

    # Iterate over all possible starting points of substrings
    for i in range(n):
        freq = defaultdict(int)
        # Extend the substring from the starting point
        for j in range(i, n):
            freq[s[j]] += 1
            if len(freq) > 1:  # Calculate beauty only if there are at least two distinct characters
                total_beauty += calculate_beauty(freq)

    return total_beauty

# Example usage
print(beauty_of_substring("aabcb"))  # Example output, the sum of beauty of all substrings
```

Reverse every word in python

```python
def reverse_words_in_string(s: str) -> str:
    # Split the string into words
    words = s.split()

    # Reverse each word
    reversed_words = [word[::-1] for word in words]

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

```python
    return reversed_string

# Example usage
print(reverse_words_in_string("Hello World"))  # Output: "olleH dlroW"
print(reverse_words_in_string("Python is fun"))  # Output: "nohtyP si nuf"
```

Linked List

Inserting a node in LinkedList

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def insert_at_position(self, position, data):
        new_node = Node(data)
        if position == 0:
            new_node.next = self.head
            self.head = new_node
            return
        current = self.head
        for i in range(position - 1):
            if current is None:
                raise IndexError("Position out of bounds")
            current = current.next
        new_node.next = current.next
```

```
        current.next = new_node

    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage:
ll = LinkedList()
ll.insert_at_beginning(3)
ll.insert_at_beginning(2)
ll.insert_at_beginning(1)
ll.insert_at_end(4)
ll.insert_at_position(2, 2.5)
ll.print_list()
```

Deleting a node in LinkedList

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def insert_at_position(self, position, data):
```

```python
        new_node = Node(data)
        if position == 0:
            new_node.next = self.head
            self.head = new_node
            return
        current = self.head
        for i in range(position - 1):
            if current is None:
                raise IndexError("Position out of bounds")
            current = current.next
        new_node.next = current.next
        current.next = new_node

    def delete_at_beginning(self):
        if self.head is None:
            print("The list is empty, nothing to delete.")
            return
        self.head = self.head.next

    def delete_at_end(self):
        if self.head is None:
            print("The list is empty, nothing to delete.")
            return
        if self.head.next is None:
            self.head = None
            return
        second_last = self.head
        while second_last.next.next:
            second_last = second_last.next
        second_last.next = None

    def delete_at_position(self, position):
        if self.head is None:
            print("The list is empty, nothing to delete.")
            return
        if position == 0:
            self.head = self.head.next
            return
        current = self.head
        for i in range(position - 1):
            if current is None or current.next is None:
                raise IndexError("Position out of bounds")
            current = current.next
        current.next = current.next.next

    def print_list(self):
        current = self.head
```

```
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage:
ll = LinkedList()
ll.insert_at_beginning(3)
ll.insert_at_beginning(2)
ll.insert_at_beginning(1)
ll.insert_at_end(4)
ll.insert_at_position(2, 2.5)
ll.print_list()

ll.delete_at_beginning()
ll.print_list()

ll.delete_at_end()
ll.print_list()

ll.delete_at_position(1)
```

Find the length of the linkedlist

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def length(self):
        count = 0
        current_node = self.head
```

```
    while current_node:
        count += 1
        current_node = current_node.next
    return count

# Example usage
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)

print("Length of the linked list:", linked_list.length())
```

Search an element in the LL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def search(self, target):
        current_node = self.head
        while current_node:
            if current_node.data == target:
                return True  # Element found
            current_node = current_node.next
        return False  # Element not found

# Example usage
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
```

```
  linked_list.append(3)

  # Search for elements
  print("Is 2 present in the linked list?", linked_list.search(2))  # Output:
True
  print("Is 4 present in the linked list?", linked_list.search(4))  # Output:
False
```

Insert a node in DLL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        new_node.prev = last_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

    def insert_after(self, prev_node, data):
        if prev_node is None:
            print("Previous node cannot be None.")
            return
        new_node = Node(data)
        new_node.next = prev_node.next
        if prev_node.next:
```

```python
            prev_node.next.prev = new_node
        prev_node.next = new_node
        new_node.prev = prev_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()

# Example usage
dll = DoublyLinkedList()

# Append nodes
dll.append(1)
dll.append(3)
dll.append(4)

# Display initial DLL
print("Original DLL:")
dll.display()  # Output: 1 3 4

# Insert at the beginning
dll.prepend(0)
print("DLL after prepend:")
dll.display()  # Output: 0 1 3 4

# Insert after a specific node (insert 2 after node with data=1)
node_to_insert_after = dll.head.next  # node with data=1
dll.insert_after(node_to_insert_after, 2)
print("DLL after insert after node with data=1:")
dll.display()  # Output: 0 1 2 3 4
```

Delete a node in DLL

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
```

```python
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        new_node.prev = last_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

    def delete(self, key):
        current = self.head

        # Case 1: Node to be deleted is the head node
        if current is not None and current.data == key:
            if current.next is not None:
                current.next.prev = None
            self.head = current.next
            current = None
            return

        # Case 2: Node to be deleted is not the head node
        while current is not None:
            if current.data == key:
                if current.next is not None:
                    current.next.prev = current.prev
                if current.prev is not None:
                    current.prev.next = current.next
                current = None
                return
            current = current.next

        # Case 3: Key not found
        if current is None:
            print(f"Node with key {key} not found.")

    def display(self):
        current = self.head
        while current:
```

```
            print(current.data, end=" ")
            current = current.next
        print()

# Example usage
dll = DoublyLinkedList()

# Append nodes
dll.append(1)
dll.append(2)
dll.append(3)

# Display initial DLL
print("Original DLL:")
dll.display()  # Output: 1 2 3

# Delete nodes
dll.delete(2)
print("DLL after deleting node with data=2:")
dll.display()  # Output: 1 3

dll.delete(1)
print("DLL after deleting node with data=1:")
dll.display()  # Output: 3

dll.delete(3)
print("DLL after deleting node with data=3:")
dll.display()  # Output: (empty)

dll.delete(4)  # Node with key 4 not found.
```

Reverse a DLL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
```

```
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        new_node.prev = last_node

    def reverse(self):
        current = self.head
        prev_node = None
        while current:
            # Swap prev and next pointers of the current node
            next_node = current.next
            current.next = prev_node
            current.prev = next_node
            # Move prev_node and current node one step forward
            prev_node = current
            current = next_node
        # Update head to point to the last node (prev_node)
        self.head = prev_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()

# Example usage
dll = DoublyLinkedList()

# Append nodes
dll.append(1)
dll.append(2)
dll.append(3)
dll.append(4)

# Display original DLL
print("Original DLL:")
dll.display()  # Output: 1 2 3 4

# Reverse DLL
dll.reverse()
print("Reversed DLL:")
dll.display()  # Output: 4 3 2 1
```

Middle of a LinkedList [TortoiseHare Method]

```
class ListNode:
```

```python
    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

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

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

middle_node = find_middle(head)
```

```python
if middle_node:
    print(f"Middle node value: {middle_node.val}")
else:
    print("Middle node not found (empty list or single node)")
```

Reverse a LinkedList [Iterative]

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

def reverse_linked_list(head):
    prev = None
    current = head

    while current:
        next_node = current.next  # Store the next node
        current.next = prev       # Reverse the current node's pointer
        prev = current            # Move prev and current one step forward
        current = next_node

    return prev  # Prev will be the new head of the reversed list

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")
```

```python
# Example usage
values = [1, 2, 3, 4, 5]
head = create_linked_list(values)

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

reversed_head = reverse_linked_list(head)

print("Reversed linked list:")
print_linked_list(reversed_head)
```

Reverse a LL [Recursive]

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

def reverse_linked_list(head):
    if head is None or head.next is None:
        return head

    # Recursively reverse the rest of the list
    reversed_head = reverse_linked_list(head.next)

    # Reverse the current node's pointer
    head.next.next = head
    head.next = None

    return reversed_head

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head
```

```python
# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

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

reversed_head = reverse_linked_list(head)

print("Reversed linked list:")
print_linked_list(reversed_head)
```

Detect a loop in LL

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

def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next  # Start fast pointer one step ahead

    while fast and fast.next:
        if slow == fast:
            return True
        slow = slow.next        # Move slow pointer one step
        fast = fast.next.next   # Move fast pointer two steps

    return False

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

```python
    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

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

# Creating a loop for testing
head.next.next.next.next.next = head.next

has_cycle_result = has_cycle(head)
if has_cycle_result:
    print("Linked list has a cycle.")
else:
    print("Linked list does not have a cycle.")
```

Find the starting point in LL

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

def detect_cycle_start(head):
    if not head or not head.next:
        return None

    # Step 1: Detect the cycle using Tortoise and Hare algorithm
    slow = head
    fast = head

    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  # No cycle

    # Step 2: Find the length of the cycle
    length = 1
    ptr = fast.next

    while ptr != fast:
        length += 1
        ptr = ptr.next

    # Step 3: Find the starting point of the cycle
    ptr1 = head
    ptr2 = head

    # Move ptr2 ahead by 'length' nodes
    for _ in range(length):
        ptr2 = ptr2.next

    # Move both pointers until they meet at the starting point
    while ptr1 != ptr2:
        ptr1 = ptr1.next
        ptr2 = ptr2.next

    return ptr1

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

    head = ListNode(values[0])
    current = head
    nodes = {head}  # To detect cycles, we will store nodes in a set

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

        if current in nodes:
            return current  # Return the node where cycle starts

        nodes.add(current)

    return head

# Example usage
values = [1, 2, 3, 4, 5]
```

```python
head = create_linked_list(values)

# Creating a cycle for testing
cycle_start_node = head.next  # Creating cycle at node with value 2
current = head
while current.next:
    current = current.next
current.next = cycle_start_node

starting_node = detect_cycle_start(head)
if starting_node:
    print(f"Starting node of the cycle: {starting_node.val}")
else:
    print("No cycle detected.")
```

Length of Loop in LL

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

def find_length_of_loop(head):
    if not head or not head.next:
        return 0

    # Step 1: Detect the cycle using Tortoise and Hare algorithm
    slow = head
    fast = head

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

        if slow == fast:
            break

    if not fast or not fast.next:
        return 0  # No cycle

    # Step 2: Find the length of the loop
    ptr = slow  # Start ptr at the meeting point

    length = 1
    while ptr.next != slow:
        length += 1
```

```python
        ptr = ptr.next

    return length

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

    head = ListNode(values[0])
    current = head
    nodes = {head}  # To detect cycles, we will store nodes in a set

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

        if current in nodes:
            return current  # Return the node where cycle starts

        nodes.add(current)

    return head

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

# Creating a cycle for testing
cycle_start_node = head.next  # Creating cycle at node with value 2
current = head
while current.next:
    current = current.next
current.next = cycle_start_node

length_of_loop = find_length_of_loop(head)
print(f"Length of the loop: {length_of_loop}")
```

Check if LL is palindrome or not

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

def is_palindrome(head):
    if not head or not head.next:
        return True

    # Step 1: Find the middle of the linked list using slow and fast
pointers
    slow = head
    fast = head

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

    # Now slow points to the middle node
    # Step 2: Reverse the second half of the linked list
    second_half = reverse_linked_list(slow)

  # Step 3: Compare the first half (head to slow) with the reversed second
half   (second_half)
    first_half = head
    while second_half:
        if first_half.val != second_half.val:
            return False
        first_half = first_half.next
        second_half = second_half.next

    return True

def reverse_linked_list(head):
    prev = None
    current = head

    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    return prev

# Function to create a linked list from a list of values
def create_linked_list(values):
    if not values:
```

```python
        return None

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage
values1 = [1, 2, 3, 2, 1]
head1 = create_linked_list(values1)

values2 = [1, 2, 3, 4, 5]
head2 = create_linked_list(values2)

print("Linked list 1:")
print_linked_list(head1)
print("Is palindrome:", is_palindrome(head1))  # Output: True

print("\nLinked list 2:")
print_linked_list(head2)
print("Is palindrome:", is_palindrome(head2))  # Output: False
```

Segrregate odd and even nodes in LL

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

def segregate_odd_even(head):
    if not head or not head.next:
        return head
```

```python
    odd_head = ListNode(0)  # Dummy node for the head of odd list
    even_head = ListNode(0) # Dummy node for the head of even list
    odd_tail = odd_head     # Pointer to the last node in odd list
    even_tail = even_head   # Pointer to the last node in even list

    current = head
    while current:
        if current.val % 2 == 0:
            even_tail.next = current
            even_tail = even_tail.next
        else:
            odd_tail.next = current
            odd_tail = odd_tail.next
        current = current.next

    # Connect the end of odd list to the start of even list
    odd_tail.next = even_head.next
    # Set the end of even list to None to terminate the list
    even_tail.next = None

    # Return the head of the segregated list (skip the dummy node)
    return odd_head.next

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

```python
head = create_linked_list(values)

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

segregated_head = segregate_odd_even(head)

print("\nSegregated linked list:")
print_linked_list(segregated_head)
```

Remove Nth node from the back of the LL

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

def remove_nth_from_end(head, n):
    if not head:
        return None

    # Create a dummy node to handle edge cases (like removing the head)
    dummy = ListNode(0)
    dummy.next = head
    slow = dummy
    fast = dummy

    # Move 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 nth node from the end
    slow.next = slow.next.next

    return dummy.next  # Return the modified linked list

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

```python
    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

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

n = 2
head = remove_nth_from_end(head, n)

print(f"\nLinked list after removing {n}th node from the end:")
print_linked_list(head)
```

Delete the middle node of LL

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

def delete_middle_node(head):
    if not head or not head.next:
        return None

    dummy = ListNode(0)
    dummy.next = head
    slow = dummy
    fast = dummy
```

```python
    # Move fast pointer two steps ahead and slow pointer one step ahead
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Now slow points to the middle node to be deleted
    middle_node = slow.next
    slow.next = middle_node.next  # Skip over the middle node

    return dummy.next  # Return the modified linked list

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

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

head = delete_middle_node(head)

print("\nLinked list after deleting the middle node:")
print_linked_list(head)
```

Sort LL

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

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

    # Step 1: Find the middle of the linked list using slow and fast 
pointers
    middle = find_middle(head)
    left_half = head
    right_half = middle.next
    middle.next = None  # Split the linked list into two halves

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

    # Step 3: Merge the sorted halves
    sorted_list = merge(left_sorted, right_sorted)

    return sorted_list

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

    slow = head
    fast = head

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

    return slow

def merge(left, right):
    dummy = ListNode(0)
    current = dummy

    while left and right:
        if left.val <= right.val:
            current.next = left
            left = left.next
        else:
            current.next = right
```

```
            right = right.next

        current = current.next

    # Append the remaining nodes of left or right sublist
    if left:
        current.next = left
    if right:
        current.next = right

    return dummy.next

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

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

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

sorted_head = merge_sort(head)

print("\nSorted linked list:")
print_linked_list(sorted_head)
```

Sort a LL of 0's 1's and 2's by changing links

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

def sort_012(head):
    if not head or not head.next:
        return head

    # Initialize three dummy nodes for each category (0, 1, 2)
    dummy_0 = ListNode(0)
    dummy_1 = ListNode(0)
    dummy_2 = ListNode(0)

    zero_tail = dummy_0
    one_tail = dummy_1
    two_tail = dummy_2

    current = head

    # Traverse the original list and append nodes to their respective
categories
    while current:
        if current.val == 0:
            zero_tail.next = current
            zero_tail = zero_tail.next
        elif current.val == 1:
            one_tail.next = current
            one_tail = one_tail.next
        else:
            two_tail.next = current
            two_tail = two_tail.next
        current = current.next

    # Connect the end of each category to the start of the next category
    zero_tail.next = dummy_1.next if dummy_1.next else dummy_2.next if
dummy_2.next else None
    one_tail.next = dummy_2.next if dummy_2.next else None
    two_tail.next = None

    # Return the sorted linked list starting from dummy_0.next
    return dummy_0.next

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

```
    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage
values = [1, 2, 0, 1, 2, 0, 1]
head = create_linked_list(values)

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

sorted_head = sort_012(head)

print("\nSorted linked list:")
print_linked_list(sorted_head)
```

Find the intersection point of Y LL

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

def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None

    # Function to calculate the length of a linked list
    def get_length(head):
        length = 0
        current = head
        while current:
```

```python
            length += 1
            current = current.next
        return length

    # Calculate lengths of both linked lists
    lenA = get_length(headA)
    lenB = get_length(headB)

    # Calculate the difference in lengths
    diff = abs(lenA - lenB)

    # Adjust the starting points of longer linked list
    if lenA > lenB:
        for _ in range(diff):
            headA = headA.next
    else:
        for _ in range(diff):
            headB = headB.next

    # Find intersection point
    while headA and headB:
        if headA == headB:
            return headA
        headA = headA.next
        headB = headB.next

    return None  # No intersection found

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Example usage
# Create linked list 1: 1 -> 2 -> 3 -> 4 -> 5 -> 6
list1_values = [1, 2, 3, 4, 5, 6]
list1_head = create_linked_list(list1_values)
```

```python
# Create linked list 2: 9 -> 8 -> 7 -> 4 -> 5 -> 6
list2_values = [9, 8, 7]
list2_head = create_linked_list(list2_values)

# Make them intersect at node with value 4
intersecting_node = list1_head.next.next.next
list2_current = list2_head
while list2_current.next:
    list2_current = list2_current.next
list2_current.next = intersecting_node

# Find intersection point
intersection_node = get_intersection_node(list1_head, list2_head)

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

Add 1 to a number represented by LL

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

def add_one(head):
    # Step 1: Reverse the linked list
    def reverse_list(head):
        prev = None
        current = head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        return prev

    reversed_head = reverse_list(head)

    # Step 2: Add 1 to the reversed linked list
    current = reversed_head
    carry = 1

    while current:
        total = current.val + carry
        current.val = total % 10
```

```python
        carry = total // 10
        if carry == 0:
            break
        if not current.next:
            current.next = ListNode(carry)
            break
        current = current.next

    # Step 3: Reverse the list back to get the final result
    result_head = reverse_list(reversed_head)
    return result_head

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

    head = ListNode(values[0])
    current = head

    for value in values[1:]:
        new_node = ListNode(value)
        current.next = new_node
        current = new_node

    return head

# Function to print elements of a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage
values = [9, 9, 9]
head = create_linked_list(values)

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

result_head = add_one(head)

print("\nLinked list after adding 1:")
print_linked_list(result_head)
```

Add 2 numbers in LL

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

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

    while l1 or l2 or carry:
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Calculate sum of current digits and carry
        total_sum = val1 + val2 + carry

        # Determine new digit and carry
        carry = total_sum // 10
        current.next = ListNode(total_sum % 10)

        # Move to next nodes
        current = current.next
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None

    return dummy_head.next

# Helper function to create a linked list from a list of values
def create_linked_list(values):
    if not values:
        return None
    head = ListNode(values[0])
    current = head
    for val in values[1:]:
        current.next = ListNode(val)
        current = current.next
    return head

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

# Example usage:
```

```python
if __name__ == "__main__":
    # Create two linked lists
    list1 = create_linked_list([2, 4, 3])  # represents number 342
    list2 = create_linked_list([5, 6, 4])  # represents number 465

    # Print the input linked lists
    print("Input:")
    print("List 1:")
    print_linked_list(list1)
    print("List 2:")
    print_linked_list(list2)

    # Add the two linked lists
    result = addTwoNumbers(list1, list2)

    # Print the result linked list
    print("\nOutput:")
    print_linked_list(result)
```

Delete all occurrences of a key in DLL

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

def deleteKey(head, key):
    # Handle the case where head itself needs to be deleted
    while head and head.data == key:
        head = head.next
        if head:
            head.prev = None

    current = head
    prev = None

    while current:
        if current.data == key:
            if prev:
                prev.next = current.next
                if current.next:
                    current.next.prev = prev
            current = current.next
        else:
            prev = current
```

```
            current = current.next

    return head

def printDLL(head):
    current = head
    while current:
        print(current.data, end=" ")
        current = current.next
    print()

# Example usage:
if __name__ == "__main__":
    # Example Doubly Linked List: 1 <-> 2 <-> 3 <-> 2 <-> 4 <-> 2 <-> None
    head = Node(1)
    head.next = Node(2)
    head.next.prev = head
    head.next.next = Node(3)
    head.next.next.prev = head.next
    head.next.next.next = Node(2)
    head.next.next.next.prev = head.next.next
    head.next.next.next.next = Node(4)
    head.next.next.next.next.prev = head.next.next.next
    head.next.next.next.next.next = Node(2)
    head.next.next.next.next.next.prev = head.next.next.next.next

    print("Original Doubly Linked List:")
    printDLL(head)

    key_to_delete = 2
    head = deleteKey(head, key_to_delete)

    print(f"Doubly Linked List after deleting {key_to_delete}:")
    printDLL(head)
```

Find pairs with given sum in DLL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

def insert_at_end(head, data):
    new_node = Node(data)
    if head is None:
        head = new_node
```

```python
    else:
        current = head
        while current.next:
            current = current.next
        current.next = new_node
        new_node.prev = current
    return head

def find_pairs_with_sum(head, target_sum):
    if head is None or head.next is None:
        print("DLL does not have enough nodes to form pairs.")
        return

    left = head
    right = head

    # Move right pointer to the end of DLL
    while right.next:
        right = right.next

    found_pair = False

    while left != right and left.prev != right:
        current_sum = left.data + right.data

        if current_sum == target_sum:
            print(f"Pair found: ({left.data}, {right.data})")
            found_pair = True
            left = left.next
            right = right.prev
        elif current_sum < target_sum:
            left = left.next
        else:
            right = right.prev

    if not found_pair:
        print("No pairs found with the given sum.")

# Example usage:
if __name__ == "__main__":
    # Create a DLL
    head = None
    data_list = [1, 2, 4, 5, 6, 7, 8]

    for data in data_list:
        head = insert_at_end(head, data)

    target_sum = 9
```

```
    find_pairs_with_sum(head, target_sum)
```

Remove duplicates from sorted DLL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

def insert_at_end(head, data):
    new_node = Node(data)
    if head is None:
        head = new_node
    else:
        current = head
        while current.next:
            current = current.next
        current.next = new_node
        new_node.prev = current
    return head

def remove_duplicates(head):
    if head is None or head.next is None:
        return head

    current = head
    while current.next:
        if current.data == current.next.data:
            # Remove the duplicate node
            next_next = current.next.next
            current.next = next_next
            if next_next:
                next_next.prev = current
        else:
            current = current.next

    return head

def print_dll(head):
    current = head
    while current:
        print(current.data, end=" ")
        current = current.next
    print()

# Example usage:
```

```python
if __name__ == "__main__":
    # Create a sorted DLL with duplicates
    head = None
    data_list = [1, 1, 2, 3, 3, 3, 4, 5, 5]

    for data in data_list:
        head = insert_at_end(head, data)

    print("Original DLL:")
    print_dll(head)

    head = remove_duplicates(head)

    print("DLL after removing duplicates:")
    print_dll(head)
```

Reverse LL in group of given size K

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def insert_at_end(head, data):
    new_node = Node(data)
    if head is None:
        head = new_node
    else:
        current = head
        while current.next:
            current = current.next
        current.next = new_node
    return head

def reverse_in_groups(head, k):
      if head is None or k <= 1:
          return head

      # Helper function to reverse a linked list within a given range
      def reverse_group(start_node, end_node):
          prev = None
          current = start_node
          next = None
          while current != end_node:
              next = current.next
              current.next = prev
              prev = current
```

```python
            current = next
        return prev

    # Initialize pointers
    current = head
    new_head = None
    prev_tail = None

    while current:
        group_start = current
        count = 1

        # Move current to the end of the current group of size k
        while count < k and current.next:
            current = current.next
            count += 1

        group_end = current
        next_start = current.next if current else None

        # Reverse the current group
        reversed_head = reverse_group(group_start, next_start)

        # Connect reversed group to the previous tail or new head
        if prev_tail:
            prev_tail.next = reversed_head
        else:
            new_head = reversed_head

        # Update prev_tail to the end of the reversed group
        prev_tail = group_start
        current = next_start

    return new_head

def print_linked_list(head):
    current = head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")

# Example usage:
if __name__ == "__main__":
    head = None
    data_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

    for data in data_list:
```

```
        head = insert_at_end(head, data)

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

    k = 3
    head = reverse_in_groups(head, k)

    print(f"Linked List after reversing in groups of {k}:")
    print_linked_list(head)
```

Rotate a LL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def insert_at_end(head, data):
    new_node = Node(data)
    if head is None:
        head = new_node
    else:
        current = head
        while current.next:
            current = current.next
        current.next = new_node
    return head

def rotate_linked_list(head, k):
    if head is None or k == 0:
        return head

    # Find the length of the Linked List
    current = head
    length = 1
    while current.next:
        current = current.next
        length += 1

    # Adjust k to be within the range of 0 to length-1
    k = k % length

    if k == 0:
        return head
```

```
    # Find the (length - k)th node (new_tail) and (length - k - 1)th node
(new_head)
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next

    new_head = new_tail.next
    current.next = head
    new_tail.next = None

    return new_head

def print_linked_list(head):
    current = head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")

# Example usage:
if __name__ == "__main__":
    head = None
    data_list = [1, 2, 3, 4, 5]

    for data in data_list:
        head = insert_at_end(head, data)

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

    k = 2
    head = rotate_linked_list(head, k)

    print(f"Linked List after rotating by {k} positions:")
    print_linked_list(head)
```

Recursion

Recursive Implementation of atoi()

```
def atoi_recursive(s):
    # Base case: if the string is empty, return 0
    if not s:
        return 0

    # Extract the first character and convert it to integer
```

```
    first_char = s[0]
    digit = ord(first_char) - ord('0')

    # Recursive call to process the rest of the string
    remaining_value = atoi_recursive(s[1:])

    # Calculate the integer value
    return digit + remaining_value * 10

# Example usage:
s = "12345"
result = atoi_recursive(s)
print(f"The integer value of '{s}' is: {result}")
```

Pow(x, n)

```
def pow_recursive(x, n):
    # Base case: if n is 0, return 1 (x^0 = 1)
    if n == 0:
        return 1
    # Base case: if n is 1, return x (x^1 = x)
    elif n == 1:
        return x
    # Recursive case for even n
    elif n % 2 == 0:
        half_pow = pow_recursive(x, n // 2)
        return half_pow * half_pow
    # Recursive case for odd n
    else:
        half_pow = pow_recursive(x, (n - 1) // 2)
        return half_pow * half_pow * x

# Example usage:
x = 2
n = 5
result = pow_recursive(x, n)
print(f"{x}^{n} = {result}")
```

Count Good numbers

```
def count_good_numbers(n):
    # Base cases
    if n == 0:
        return 0
    if n == 1:
        return 2  # Possible numbers are '2' and '5'
```

```python
    # Memoization dictionary to store already computed results
    memo = {}

    def recursive_count(length):
        if length == 0:
            return 1  # One way to form an empty number (by doing nothing)

        # Check memoization table
        if length in memo:
            return memo[length]

        # Calculate the number of good numbers of length `length`
        # Each position can be either '2' or '5'
        result = 0
        result += 2 * recursive_count(length - 1)  # '2' at current position
        result += 2 * recursive_count(length - 1)  # '5' at current position

        # Memoize the result
        memo[length] = result
        return result

    return recursive_count(n)

# Example usage:
n = 3
print(f"Number of good numbers of length {n}: {count_good_numbers(n)}")
```

Sort a stack using recursion

```python
def sort_stack(stack):
    if not stack:
        return

    temp_stack = []
    while stack:
        # Pop the top element from the stack
        value = stack.pop()

        # Insert value into the sorted position in temp_stack
        insert_sorted(temp_stack, value)

    # Move elements from temp_stack back to stack
    while temp_stack:
        stack.append(temp_stack.pop())

def insert_sorted(stack, value):
```

```
    if not stack or stack[-1] <= value:
        stack.append(value)
    else:
        # Pop the top element from stack and recurse
        top = stack.pop()
        insert_sorted(stack, value)
        stack.append(top)

# Example usage:
stack = [3, 1, 4, 2, 5]
print("Original Stack:", stack)
sort_stack(stack)
print("Sorted Stack:", stack)
```

Reverse a stack using recursion

```
def reverse_stack(stack):
    if not stack:
        return

    # Function to insert an element at the bottom of the stack
    def insert_at_bottom(stack, item):
        if not stack:
            stack.append(item)
        else:
            # Pop all elements from stack and push them back after item
            temp = stack.pop()
            insert_at_bottom(stack, item)
            stack.append(temp)

    # Pop the top element from the stack
    top = stack.pop()

    # Recursively reverse the remaining stack
    reverse_stack(stack)

    # Insert the top element at the bottom of the reversed stack
    insert_at_bottom(stack, top)

# Example usage:
stack = [1, 2, 3, 4, 5]
print("Original Stack:", stack)
reverse_stack(stack)
print("Reversed Stack:", stack)
```

Generate all binary strings

```python
def generate_binary_strings(n):
    # Helper function to generate binary strings recursively
    def generate_helper(current_string, length):
        # Base case: if current_string reaches desired length, print it
        if length == 0:
            print(current_string)
            return

        # Recursive case: add '0' and '1' to current_string and recurse
        generate_helper(current_string + '0', length - 1)
        generate_helper(current_string + '1', length - 1)

    # Start recursion with an empty string and length n
    generate_helper("", n)

# Example usage:
n = 3
print(f"All binary strings of length {n}:")
generate_binary_strings(n)
```

Generate Paranthesis

```python
def generate_parenthesis(n):
    result = []

    # Helper function to generate valid parentheses recursively
    def generate(current, open_count, close_count):
        if len(current) == 2 * n:
            result.append(current)
            return

        if open_count < n:
            generate(current + '(', open_count + 1, close_count)

        if close_count < open_count:
            generate(current + ')', open_count, close_count + 1)

    # Start recursion with an empty string and counts of open and close
parentheses
    generate("", 0, 0)

    return result

# Example usage:
n = 3
print(f"All valid parentheses of length {2*n}:")
print(generate_parenthesis(n))
```

Print all subsequences/Power Set

```python
def generate_subsequences(input_string):
    # Helper function to generate subsequences recursively
    def generate_helper(current, index):
        if index == len(input_string):
            result.append(current)
            return
        # Exclude the current character
        generate_helper(current, index + 1)
        # Include the current character
        generate_helper(current + input_string[index], index + 1)

    result = []
    generate_helper("", 0)
    return result

# Example usage:
input_string = "abc"
print(f"All subsequences of '{input_string}':")
subsequences = generate_subsequences(input_string)
print(subsequences)
```

Count all subsequences with sum K

```python
def count_subsequences_with_sum_k(arr, k):
    def count_helper(index, current_sum):
        # Base case: if we have reached the end of the array
        if index == len(arr):
            return 1 if current_sum == k else 0

        # Include the current element in the sum
        include = count_helper(index + 1, current_sum + arr[index])

        # Exclude the current element from the sum
        exclude = count_helper(index + 1, current_sum)

        # Return the count of both included and excluded subsequences
        return include + exclude

    return count_helper(0, 0)

# Example usage:
arr = [1, 2, 3]
```

```
  k = 3
  print(f"Number of subsequences with sum {k}:
{count_subsequences_with_sum_k(arr, k)}")
```

Check if there exists a subsequence with sum K

```
  def has_subsequence_with_sum_k(arr, k):
      def helper(index, current_sum):
          # Base case: if we have reached the end of the array
          if index == len(arr):
              return current_sum == k

          # Check if the current_sum already equals k
          if current_sum == k:
              return True

          # Include the current element in the sum and recurse
          include = helper(index + 1, current_sum + arr[index])

          # Exclude the current element from the sum and recurse
          exclude = helper(index + 1, current_sum)

 #Return true if either inclusion or exclusion yields a subsequence with sum k
          return include or exclude

      return helper(0, 0)

  # Example usage:
  arr = [1, 2, 3]
  k = 5
  print(f"Does there exist a subsequence with sum {k}?
{has_subsequence_with_sum_k(arr, k)}")
```

Combination Sum

```
  def combination_sum(candidates, target):
      def backtrack(start, path, target):
          if target == 0:
              result.append(path)
              return
          if target < 0:
              return

          for i in range(start, len(candidates)):
              # Include the number candidates[i] in the combination
              backtrack(i, path + [candidates[i]], target - candidates[i])
```

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

# Example usage:
candidates = [2, 3, 6, 7]
target = 7
print(f"Combinations that sum to {target}: {combination_sum(candidates,
target)}")
```

Combination Sum-II

```
def combination_sum_ii(candidates, target):
    def backtrack(start, path, target):
        if target == 0:
            result.append(path)
            return
        if target < 0:
            return

        for i in range(start, len(candidates)):
            # Skip duplicates
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            # Include the number candidates[i] in the combination and move
to the next index
            backtrack(i + 1, path + [candidates[i]], target - candidates[i])

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

# Example usage:
candidates = [10, 1, 2, 7, 6, 1, 5]
target = 8
print(f"Combinations that sum to {target}: {combination_sum_ii(candidates,
target)}")
```

Subset Sum-I

```
def subset_sum(arr, target):
    def backtrack(start, current_sum, path):
        # If the current sum equals the target, add the current path to the
results
```

```python
        if current_sum == target:
            result.append(path)
            return
        # If the current sum exceeds the target, no need to continue
        if current_sum > target:
            return

        for i in range(start, len(arr)):
            # Include the current element in the path and recurse
            backtrack(i + 1, current_sum + arr[i], path + [arr[i]])

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

# Example usage:
arr = [1, 2, 3, 4, 5]
target = 5
print(f"Subsets that sum to {target}: {subset_sum(arr, target)}")
```

Subset Sum-II

```python
def subset_sum_ii(candidates, target):
    def backtrack(start, current_sum, path):
# If the current sum equals the target, add the current path to the results
        if current_sum == target:
            result.append(path)
            return
        # If the current sum exceeds the target, no need to continue
        if current_sum > target:
            return

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

    candidates.sort()  # Sort to handle duplicates
    result = []
    backtrack(0, 0, [])
    return result

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

```
  target = 8
  print(f"Unique subsets that sum to {target}: {subset_sum_ii(candidates,
target)}")
```

Combination Sum – III

```
  def combination_sum_iii(k, n):
      def backtrack(start, path, target):
          # If the combination is of length k and the target is reached
          if len(path) == k and target == 0:
              result.append(path)
              return
          # If the combination is of length k but the target is not reached,
or if the target goes negative
          if len(path) == k or target < 0:
              return

          for i in range(start, 10):
              # Include the number i in the combination and recurse
              backtrack(i + 1, path + [i], target - i)

      result = []
      backtrack(1, [], n)
      return result

  # Example usage:
  k = 3
  n = 7
  print(f"Combinations of {k} numbers that sum to {n}: {combination_sum_iii(k,
n)}")
```

Letter Combinations of a Phone number

```
  def letter_combinations(digits):
      if not digits:
          return []

      # Mapping of digits to corresponding letters
      phone_map = {
          '2': 'abc', '3': 'def', '4': 'ghi',
          '5': 'jkl', '6': 'mno', '7': 'pqrs',
          '8': 'tuv', '9': 'wxyz'
      }

      def backtrack(index, path):
```

```python
        # If the current combination is the same length as digits, we have a
complete   combination
        if index == len(digits):
            combinations.append("".join(path))
            return

        # Get the letters that the current digit maps to, and loop through
them
        possible_letters = phone_map[digits[index]]
        for letter in possible_letters:
            # Add the letter to the current combination
            path.append(letter)
            # Move on to the next digit
            backtrack(index + 1, path)
            # Backtrack by removing the letter before moving on to the next
            path.pop()

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

# Example usage:
digits = "23"
print(f"Letter combinations for {digits}: {letter_combinations(digits)}")
```

Palindrome Partitioning

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

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

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

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

```python
# Example usage:
s = "aab"
print(f"All palindrome partitions of '{s}': {partition(s)}")
```

Word Search

```python
def exist(board, word):
    def dfs(i, j, k):
        # Base case: If all characters in word are found
        if k == len(word):
            return True

        # Check boundaries and if board[i][j] matches current character in
word
       if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or
board[i][j] != word  [k]:
            return False

        # Mark board[i][j] as visited (using a placeholder character)
        temp = board[i][j]
        board[i][j] = '#'

        # Explore adjacent cells (up, down, left, right)
        found = (dfs(i + 1, j, k + 1) or
                 dfs(i - 1, j, k + 1) or
                 dfs(i, j + 1, k + 1) or
                 dfs(i, j - 1, k + 1))

        # Restore the original character on the board
        board[i][j] = temp

        return found

    # Iterate through the board to find the first character of the word
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == word[0] and dfs(i, j, 0):
                return True

    return False

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

```
word1 = "ABCCED"
word2 = "SEE"
word3 = "ABCB"

print(f"Does '{word1}' exist in the board? {exist(board, word1)}")
print(f"Does '{word2}' exist in the board? {exist(board, word2)}")
print(f"Does '{word3}' exist in the board? {exist(board, word3)}")
```

N Queen

```
def solve_n_queens(n):
    def is_safe(row, col):
        # Check if there's a queen in the same column or diagonal
        for r in range(row):
            if board[r] == col or abs(board[r] - col) == row - r:
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append([''.join(['Q' if col == board[row] else '.' for
col in range(n)]) for row in range(n)])
            return
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                backtrack(row + 1)

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

# Example usage:
n = 4
solutions = solve_n_queens(n)
for i, solution in enumerate(solutions):
    print(f"Solution {i + 1}:")
    for row in solution:
        print(row)
    print()
```

Rat in a Maze

```
def solve_maze(maze):
    if not maze:
```

```python
        return []

    n = len(maze)
    result = []

    def backtrack(x, y, path):
        # Base case: Reached the bottom-right corner of the maze
        if x == n-1 and y == n-1:
            result.append(path[:])
            return

        # Check all possible moves (down, right, up, left)
        moves = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n and maze[nx][ny] == 1:
                # Mark the current cell as visited
                maze[x][y] = 0
                path.append((nx, ny))
                backtrack(nx, ny, path)
                # Backtrack by removing the last move
                path.pop()
                # Unmark the current cell
                maze[x][y] = 1

    # Start from the top-left corner (0, 0)
    if maze[0][0] == 1:
        backtrack(0, 0, [(0, 0)])

    return result

# Example usage:
maze = [
    [1, 0, 0, 0],
    [1, 1, 0, 1],
    [0, 1, 0, 0],
    [1, 1, 1, 1]
]
paths = solve_maze(maze)
for i, path in enumerate(paths):
    print(f"Path {i + 1}: {path}")
```

Word Break

```python
def word_break(s, word_dict):
    n = len(s)
```

```
      # Create a dp array where dp[i] will be True if substring s[:i] can be
segmented into words from word_dict
      dp = [False] * (n + 1)
      dp[0] = True  # Empty string is always breakable

      for i in range(1, n + 1):
          for j in range(i):
              if dp[j] and s[j:i] in word_dict:
                  dp[i] = True
                  break

      return dp[n]

  # Example usage:
  s = "leetcode"
  word_dict = ["leet", "code"]
  print(f"Can '{s}' be segmented into words from the dictionary?
{word_break(s, word_dict)}")

  s = "applepenapple"
  word_dict = ["apple", "pen"]
  print(f"Can '{s}' be segmented into words from the dictionary?
{word_break(s, word_dict)}")

  s = "catsandog"
  word_dict = ["cats", "dog", "sand", "and", "cat"]
  print(f"Can '{s}' be segmented into words from the dictionary?
{word_break(s, word_dict)}")
```

M Coloring Problem

```
  def is_safe(graph, v, color, c):
      # Check if there's an adjacent vertex with the same color
      for i in range(len(graph)):
          if graph[v][i] == 1 and color[i] == c:
              return False
      return True

  def graph_coloring(graph, m, v, color):
      # Base case: if all vertices are assigned a color
      if v == len(graph):
          return True

      for c in range(1, m + 1):
          if is_safe(graph, v, color, c):
              color[v] = c
              if graph_coloring(graph, m, v + 1, color):
```

```
                return True
            color[v] = 0

    return False

# Example usage:
graph = [
    [0, 1, 1, 1],
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 0]
]
m = 3  # Number of colors
color = [0] * len(graph)

if graph_coloring(graph, m, 0, color):
    print(f"Graph can be colored using at most {m} colors.")
    print(f"Colors assigned to vertices: {color}")
else:
    print("Graph cannot be colored using at most", m, "colors.")
```

Sudoko Solver

```
def is_safe(board, row, col, num):
    # Check if the number is already present in the current row
    if num in board[row]:
        return False

    # Check if the number is already present in the current column
    for r in range(9):
        if board[r][col] == num:
            return False

    # Check if the number is already present in the current 3x3 box
    box_row = (row // 3) * 3
    box_col = (col // 3) * 3
    for r in range(box_row, box_row + 3):
        for c in range(box_col, box_col + 3):
            if board[r][c] == num:
                return False

    return True

def solve_sudoku(board):
    def backtrack(board, row, col):
        if row == 9:
            return True  # Entire board has been filled
```

```python
        next_row = row if col < 8 else row + 1
        next_col = (col + 1) % 9

        if board[row][col] != 0:
            return backtrack(board, next_row, next_col)

        for num in range(1, 10):
            if is_safe(board, row, col, num):
                board[row][col] = num
                if backtrack(board, next_row, next_col):
                    return True
                board[row][col] = 0

        return False

    backtrack(board, 0, 0)

# 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]
]

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

solve_sudoku(board)

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

Expression Add Operators

```python
def add_operators(num, target):
    def backtrack(index, path, current_value, last_operand):
        if index == len(num):
            if current_value == target:
```

```python
                result.append(path)
            return

        for i in range(index, len(num)):
            # Avoid leading zeros in operands (except single '0')
            if i > index and num[index] == '0':
                break

            operand_str = num[index:i + 1]
            operand = int(operand_str)

            if index == 0:
                backtrack(i + 1, operand_str, operand, operand)
            else:
              backtrack(i + 1, path + '+' + operand_str, current_value +
operand,    operand)
              backtrack(i + 1, path + '-' + operand_str, current_value -
operand,    -operand)
              backtrack(i + 1, path + '*' + operand_str, current_value -
last_operand +    last_operand * operand, last_operand * operand)

    result = []
    if not num:
        return result
    backtrack(0, '', 0, 0)
    return result

# Example usage:
num = "123"
target = 6
print(add_operators(num, target))

num = "232"
target = 8
print(add_operators(num, target))

num = "105"
target = 5
print(add_operators(num, target))

num = "00"
target = 0
print(add_operators(num, target))

num = "3456237490"
target = 9191
print(add_operators(num, target))
```

Bit Manipulation

Check if the i-th bit is set or not

```
def is_ith_bit_set(num, i):
    # Create a bitmask where only the i-th bit is set
    bitmask = 1 << i
    # Perform bitwise AND operation to check if the i-th bit is set
    if (num & bitmask) != 0:
        return True
    else:
        return False

# Example usage:
num = 5  # Binary: 101
print(f"Binary representation of {num}: {bin(num)}")

# Check if specific bits are set
for i in range(3):  # Checking the 0th, 1st, and 2nd bits
    if is_ith_bit_set(num, i):
        print(f"The {i}-th bit is set.")
    else:
        print(f"The {i}-th bit is not set.")
```

Check if a number is odd or not

```
def is_odd_bitwise(num):
    if num & 1 != 0:
        return True
    else:
        return False

# Example usage:
number = 9

if is_odd_bitwise(number):
    print(f"{number} is odd.")
else:
    print(f"{number} is not odd (i.e., it is even).")
```

Check if a number is power of 2 or not

```
def is_power_of_two(num):
    if num <= 0:
        return False
```

```python
    # A power of two has exactly one bit set in its binary representation.
    # Using num & (num - 1) checks if only one bit is set.
    return (num & (num - 1)) == 0

# Example usage:
number = 16

if is_power_of_two(number):
    print(f"{number} is a power of 2.")
else:
    print(f"{number} is not a power of 2.")
```

Count the number of set bits

```python
def count_set_bits(num):
    count = 0
    while num:
        count += num & 1
        num >>= 1
    return count

# Example usage:
number = 23

print(f"Number of set bits in {number}: {count_set_bits(number)}")
```

Set/Unset the rightmost unset bit

```python
def set_rightmost_unset_bit(num):
    return num | (num + 1)

# Example usage:
number = 23

print(f"Original number: {number}")
modified_number = set_rightmost_unset_bit(number)
print(f"Number after setting rightmost unset bit: {modified_number}")
```

```python
def unset_rightmost_set_bit(num):
    return num & (num - 1)

# Example usage:
number = 23
```

```python
print(f"Original number: {number}")
modified_number = unset_rightmost_set_bit(number)
print(f"Number after unsetting rightmost set bit: {modified_number}")
```

Swap two numbers

```python
def swap_numbers(a, b):
    print(f"Original values: a = {a}, b = {b}")
    # Step 1: a becomes a ^ b
    a = a ^ b
    # Step 2: b becomes a ^ b ^ b = a
    b = a ^ b
    # Step 3: a becomes a ^ b ^ a = b
    a = a ^ b
    print(f"Swapped values: a = {a}, b = {b}")
    return a, b

# Example usage
a, b = 5, 10
a, b = swap_numbers(a, b)
```

Divide two integers without using multiplication, division and mod operator.

```python
def divide(dividend, divisor):
    # Handle special cases
    if divisor == 0:
        raise ValueError("Cannot divide by zero.")
    if dividend == 0:
        return 0
    if dividend == -2147483648 and divisor == -1:
        return 2147483647  # Overflow case

    # Determine the sign of the result
    negative = (dividend < 0) ^ (divisor < 0)

    # Work with absolute values
    dividend, divisor = abs(dividend), abs(divisor)

    quotient = 0
    # The highest bit we will check is 31, since Python integers are
unbounded
    # and we need to ensure we handle up to 32-bit signed integers
    for i in range(31, -1, -1):
        if (dividend >> i) >= divisor:
            quotient += 1 << i
            dividend -= divisor << i
```

```python
    return -quotient if negative else quotient

# Example usage
dividend = 43
divisor = 8
result = divide(dividend, divisor)
print(f"Result of {dividend} divided by {divisor} is {result}")
```

Count number of bits to be flipped to convert A to B

```python
def count_bits_to_flip(a, b):
    # Step 1: XOR of a and b
    xor_result = a ^ b

    # Step 2: Count the number of 1s in the xor_result
    count = 0
    while xor_result:
        count += xor_result & 1
        xor_result >>= 1

    return count

# Example usage
a = 29  # Binary: 11101
b = 15  # Binary: 01111
result = count_bits_to_flip(a, b)
print(f"Number of bits to be flipped to convert {a} to {b} is {result}")
```

Find the number that appears odd number of times

```python
def find_odd_occurrence(arr):
    result = 0
    for number in arr:
        result ^= number
    return result

# Example usage
arr = [1, 2, 3, 2, 3, 1, 3]
odd_occurrence = find_odd_occurrence(arr)
print(f"The number that appears an odd number of times is {odd_occurrence}")
```

Power Set

```python
def power_set(s):
    n = len(s)
    power_set_result = []

    # There are 2^n possible subsets
    for i in range(2**n):
        subset = []
        for j in range(n):
            # Check if jth bit in the i is set. If set then add s[j] to
subset.
            if (i & (1 << j)) > 0:
                subset.append(s[j])
        power_set_result.append(subset)

    return power_set_result

# Example usage
s = [1, 2, 3]
result = power_set(s)
print("Power set:")
for subset in result:
    print(subset)
```

Find xor of numbers from L to R

```python
def xor_upto(n):
    if n % 4 == 0:
        return n
    elif n % 4 == 1:
        return 1
    elif n % 4 == 2:
        return n + 1
    elif n % 4 == 3:
        return 0

def xor_L_to_R(L, R):
    return xor_upto(R) ^ xor_upto(L - 1)

# Example usage:
L = 5
R = 8
result = xor_L_to_R(L, R)
print("XOR from {} to {} is: {}".format(L, R, result))
```

Find the two numbers appearing odd number of times

```python
def find_two_odd_numbers(arr):
    # Step 1: XOR all elements in the array to get xor_all
    xor_all = 0
    for num in arr:
        xor_all ^= num

    # Step 2: Find a set bit in xor_all (we use the rightmost set bit)
    rightmost_set_bit = xor_all & -xor_all

    # Step 3: Divide elements into two groups and XOR each group
    num1 = 0
    num2 = 0
    for num in arr:
        if num & rightmost_set_bit:
            num1 ^= num
        else:
            num2 ^= num

    return num1, num2

# Example usage:
arr = [4, 2, 4, 5, 2, 3, 3, 1]
result = find_two_odd_numbers(arr)
print("The two numbers appearing odd number of times are:", result)
```

Print Prime Factors of a Number using bit manipulation

```python
def print_prime_factors(n):
    # Handle factors of 2 separately
    while n % 2 == 0:
        print(2, end=' ')
        n //= 2

    # Handle odd factors from 3 onwards
    for i in range(3, int(n**0.5) + 1, 2):
        while n % i == 0:
            print(i, end=' ')
            n //= i

    # If n is a prime number greater than 2
    if n > 2:
        print(n)

# Example usage:
```

```
number = 315
print("Prime factors of", number, "are:")
print_prime_factors(number)
```

All Divisors of a Number

```
def find_all_divisors_bitwise(n):
    divisors = set()
    bit_position = 0
    while (1 << bit_position) <= n:
        if (n & (1 << bit_position)) != 0:
            divisor = 1 << bit_position
            divisors.add(divisor)
            if divisor != n // divisor:
                divisors.add(n // divisor)
        bit_position += 1
    return sorted(divisors)

# Example usage:
number = 36
print(f"All divisors of {number}:")
divisors = find_all_divisors_bitwise(number)
print(divisors)
```

Sieve of Eratosthenes

```
def sieve_of_eratosthenes(limit):
  # Calculate the number of bits needed (limit + 1 bits, but bitarray index
starts from   0)
    size = (limit // 32) + 1
    bitarray = [0] * size

    def set_bit(x):
        bitarray[x // 32] |= (1 << (x % 32))

    def clear_bit(x):
        bitarray[x // 32] &= ~(1 << (x % 32))

    def is_bit_set(x):
        return bitarray[x // 32] & (1 << (x % 32))

    # 0 and 1 are not prime numbers
    set_bit(0)
    set_bit(1)
```

```
    # Sieve of Eratosthenes algorithm
    for i in range(2, int(limit**0.5) + 1):
        if not is_bit_set(i):
            for multiple in range(i * i, limit + 1, i):
                set_bit(multiple)

    # Collect all primes
    primes = [i for i in range(2, limit + 1) if not is_bit_set(i)]
    return primes

# Example usage:
limit = 50
primes = sieve_of_eratosthenes(limit)
print(f"Primes up to {limit}: {primes}")
```

Find Prime Factorisation of a Number using Sieve

```
def sieve_with_spf(limit):
    # Array to store the smallest prime factor for every number
    spf = list(range(limit + 1))  # Initialize spf[i] = i

    # Sieve of Eratosthenes to fill the spf array
    for i in range(2, int(limit**0.5) + 1):
        if spf[i] == i:  # i is a prime number
            for j in range(i * i, limit + 1, i):
                if spf[j] == j:
                    spf[j] = i
    return spf

def prime_factorization(n, spf):
    factors = []
    while n != 1:
        factors.append(spf[n])
        n //= spf[n]
    return factors

# Example usage:
limit = 100  # You can set this to any number based on your needs
spf = sieve_with_spf(limit)

number = 84
print(f"Prime factorization of {number}:")
factors = prime_factorization(number, spf)
print(factors)
```

Power(n, x)

```python
def power(n, x):
    result = 1
    base = n

    while x > 0:
        # If the current bit is set, multiply the result by the base
        if x & 1:
            result *= base

        # Square the base for the next bit
        base *= base

        # Shift the exponent to the right by 1 bit
        x >>= 1

    return result

# Example usage:
n = 2
x = 10
print(f"{n}^{x} =", power(n, x))
```

Stack & Queue

Implement Stack using Arrays

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

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

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

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

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.stack[-1]
```

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

    def __str__(self):
        return str(self.stack)

# Example usage:
if __name__ == "__main__":
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(3)
    print(f"Stack: {s}")
    print(f"Top element: {s.peek()}")
    print(f"Popped element: {s.pop()}")
    print(f"Stack after pop: {s}")
    print(f"Is stack empty? {s.is_empty()}")
    print(f"Size of stack: {s.size()}")
```

Implement Queue using Arrays

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

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

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.queue[0]

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

    def __str__(self):
        return str(self.queue)
```

```python
# Example usage:
if __name__ == "__main__":
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print(f"Queue: {q}")
    print(f"Front element: {q.peek()}")
    print(f"Dequeued element: {q.dequeue()}")
    print(f"Queue after dequeue: {q}")
    print(f"Is queue empty? {q.is_empty()}")
    print(f"Size of queue: {q.size()}")
```

Implement Stack using Queue

```python
from queue import Queue

class StackUsingQueues:
    def __init__(self):
        self.queue1 = Queue()
        self.queue2 = Queue()

    def is_empty(self):
        return self.queue1.empty()

    def push(self, item):
        # Always push to queue2
        self.queue2.put(item)

        # Push all elements of queue1 to queue2
        while not self.queue1.empty():
            self.queue2.put(self.queue1.get())

        # Swap the names of queue1 and queue2
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.queue1.get()

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.queue1.queue[0]
```

```
    def size(self):
        return self.queue1.qsize()

    def __str__(self):
        return str(list(self.queue1.queue))

# Example usage:
if __name__ == "__main__":
    stack = StackUsingQueues()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(f"Stack: {stack}")
    print(f"Top element: {stack.peek()}")
    print(f"Popped element: {stack.pop()}")
    print(f"Stack after pop: {stack}")
    print(f"Is stack empty? {stack.is_empty()}")
    print(f"Size of stack: {stack.size()}")
```

Implement Queue using Stack

```
class QueueUsingStacks:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

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

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

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

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

    def peek(self):
        if not self.stack2:
            while self.stack1:
                self.stack2.append(self.stack1.pop())
```

```python
        if not self.stack2:
            raise IndexError("peek from empty queue")
        return self.stack2[-1]

# Example usage
queue = QueueUsingStacks()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())  # Output: 1
queue.enqueue(4)
print(queue.dequeue())  # Output: 2
print(queue.peek())     # Output: 3
print(queue.dequeue())  # Output: 3
print(queue.dequeue())  # Output: 4
```

Implement stack using Linkedlist

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

class StackUsingLinkedList:
    def __init__(self):
        self.top = None
        self.size = 0

    def push(self, value):
        new_node = Node(value)
        new_node.next = self.top
        self.top = new_node
        self.size += 1

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        popped_value = self.top.value
        self.top = self.top.next
        self.size -= 1
        return popped_value

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.top.value
```

```python
    def is_empty(self):
        return self.top is None

    def get_size(self):
        return self.size

# Example usage
if __name__ == "__main__":
    stack = StackUsingLinkedList()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.pop())  # Output: 3
    print(stack.peek()) # Output: 2
    print(stack.pop())  # Output: 2
    print(stack.get_size())  # Output: 1
    print(stack.pop())  # Output: 1
    print(stack.is_empty())  # Output: True
```

Implement queue using Linkedlist

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

class QueueUsingLinkedList:
    def __init__(self):
        self.front = None  # Front of the queue (oldest element)
        self.rear = None   # Rear of the queue (newest element)
        self.size = 0      # Size of the queue

    def enqueue(self, value):
        new_node = Node(value)
        if self.is_empty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        removed_value = self.front.value
        self.front = self.front.next
```

```
        if self.front is None:
            self.rear = None
        self.size -= 1
        return removed_value

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.front.value

    def is_empty(self):
        return self.front is None

    def get_size(self):
        return self.size

# Example usage
if __name__ == "__main__":
    queue = QueueUsingLinkedList()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.dequeue())  # Output: 1
    print(queue.peek())     # Output: 2
    print(queue.dequeue())  # Output: 2
    print(queue.get_size()) # Output: 1
    print(queue.dequeue())  # Output: 3
    print(queue.is_empty()) # Output: True
```

Check for balanced paranthesis

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

    for char in expression:
        if char in mapping.values():  # if it's an opening bracket
            stack.append(char)
        elif char in mapping:  # if it's a closing bracket
            if not stack or stack[-1] != mapping[char]:
                return False
            stack.pop()

    return len(stack) == 0

# Example usage
if __name__ == "__main__":
```

```python
    test_cases = ["{[()]}", "[()]{}", "{[(])}]", "(", ")", ""]
    for expression in test_cases:
        print(f"{expression} is balanced: {is_balanced(expression)}")
```

Implement Min Stack

```python
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, value):
        self.stack.append(value)
        if not self.min_stack or value <= self.min_stack[-1]:
            self.min_stack.append(value)

    def pop(self):
        if not self.stack:
            raise IndexError("pop from empty stack")
        popped_value = self.stack.pop()
        if popped_value == self.min_stack[-1]:
            self.min_stack.pop()
        return popped_value

    def top(self):
        if not self.stack:
            raise IndexError("top from empty stack")
        return self.stack[-1]

    def get_min(self):
        if not self.min_stack:
            raise IndexError("get_min from empty stack")
        return self.min_stack[-1]

# Example usage
if __name__ == "__main__":
    min_stack = MinStack()
    min_stack.push(3)
    min_stack.push(1)
    min_stack.push(5)
    min_stack.push(2)

    print("Current stack:", min_stack.stack)
    print("Min stack:", min_stack.min_stack)

    print("Top element:", min_stack.top())    # Output: 2
    print("Minimum element:", min_stack.get_min())    # Output: 1
```

```
    min_stack.pop()
    print("After popping one element:")
    print("Current stack:", min_stack.stack)
    print("Min stack:", min_stack.min_stack)
```

Infix to Postfix Conversion using Stack

```
def precedence(op):
    if op in ('+', '-'):
        return 1
    if op in ('*', '/'):
        return 2
    if op == '^':
        return 3
    return 0

def infix_to_postfix(expression):
    stack = []
    output = []

    for token in expression:
        if token.isalnum():  # If the token is an operand (considering only
alphanumeric characters here)
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Pop the left parenthesis
        else:  # The token is an operator
            while stack and precedence(stack[-1]) >= precedence(token):
                output.append(stack.pop())
            stack.append(token)

    # Pop all the operators left in the stack
    while stack:
        output.append(stack.pop())

    return ''.join(output)

# Example usage
infix_expr = "A+B*C-(D/E^F)*G"
postfix_expr = infix_to_postfix(infix_expr)
print("Infix Expression: ", infix_expr)
print("Postfix Expression: ", postfix_expr)
```

Prefix to Infix Conversion

```
def is_operator(c):
    return c in ['+', '-', '*', '/', '^']

def prefix_to_infix(prefix_expr):
    stack = []

    # Traverse the prefix expression from right to left
    for c in reversed(prefix_expr):
        if is_operator(c):
            # Pop two operands from the stack
            op1 = stack.pop()
            op2 = stack.pop()
            # Form the infix expression
            new_expr = '(' + op1 + c + op2 + ')'
            # Push the resulting string back to the stack
            stack.append(new_expr)
        else:
            # Push the operand to the stack
            stack.append(c)

    # The final element in the stack is the infix expression
    return stack.pop()

# Example usage
prefix_expr = "*-A/BC-/AKL"
infix_expr = prefix_to_infix(prefix_expr)
print("Prefix Expression: ", prefix_expr)
print("Infix Expression: ", infix_expr)
```

Prefix to Postfix Conversion

```
def is_operator(c):
    return c in ['+', '-', '*', '/', '^']

def prefix_to_postfix(prefix_expr):
    stack = []

    # Traverse the prefix expression from right to left
    for c in reversed(prefix_expr):
        if is_operator(c):
            # Pop two operands from the stack
            op1 = stack.pop()
            op2 = stack.pop()
```

```python
            # Form the postfix expression
            new_expr = op1 + op2 + c
            # Push the resulting string back to the stack
            stack.append(new_expr)
        else:
            # Push the operand to the stack
            stack.append(c)

    # The final element in the stack is the postfix expression
    return stack.pop()

# Example usage
prefix_expr = "*-A/BC-/AKL"
postfix_expr = prefix_to_postfix(prefix_expr)
print("Prefix Expression: ", prefix_expr)
print("Postfix Expression: ", postfix_expr)
```

Postfix to Prefix Conversion

```python
def is_operator(c):
    return c in ['+', '-', '*', '/', '^']

def postfix_to_prefix(postfix_expr):
    stack = []

    # Traverse the postfix expression from left to right
    for c in postfix_expr:
        if is_operator(c):
            # Pop two operands from the stack
            op2 = stack.pop()
            op1 = stack.pop()
            # Form the prefix expression
            new_expr = c + op1 + op2
            # Push the resulting string back to the stack
            stack.append(new_expr)
        else:
            # Push the operand to the stack
            stack.append(c)

    # The final element in the stack is the prefix expression
    return stack.pop()

# Example usage
postfix_expr = "ABC/-AK/L-*"
prefix_expr = postfix_to_prefix(postfix_expr)
print("Postfix Expression: ", postfix_expr)
print("Prefix Expression: ", prefix_expr)
```

Postfix to Infix

```python
def is_operator(c):
    return c in ['+', '-', '*', '/', '^']

def postfix_to_infix(postfix_expr):
    stack = []

    # Traverse the postfix expression from left to right
    for c in postfix_expr:
        if is_operator(c):
            # Pop two operands from the stack
            op2 = stack.pop()
            op1 = stack.pop()
            # Form the infix expression
            new_expr = f'({op1}{c}{op2})'
            # Push the resulting string back to the stack
            stack.append(new_expr)
        else:
            # Push the operand to the stack
            stack.append(c)

    # The final element in the stack is the infix expression
    return stack.pop()

# Example usage
postfix_expr = "ABC/-AK/L-*"
infix_expr = postfix_to_infix(postfix_expr)
print("Postfix Expression: ", postfix_expr)
print("Infix Expression: ", infix_expr)
```

Convert Infix To Prefix Notation

```python
def precedence(op):
    if op in ('+', '-'):
        return 1
    if op in ('*', '/'):
        return 2
    if op == '^':
        return 3
    return 0

def infix_to_postfix(expression):
    stack = []
    output = []
```

```python
    for token in expression:
        if token.isalnum():  # If the token is an operand (considering only alphanumeric characters here)
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Pop the left parenthesis
        else:  # The token is an operator
            while stack and precedence(stack[-1]) >= precedence(token):
                output.append(stack.pop())
            stack.append(token)

    # Pop all the operators left in the stack
    while stack:
        output.append(stack.pop())

    return ''.join(output)

def infix_to_prefix(expression):
    # Step 1: Reverse the infix expression
    expression = expression[::-1]

    # Step 2: Replace '(' with ')' and vice versa
    expression = expression.replace('(', 'temp')
    expression = expression.replace(')', '(')
    expression = expression.replace('temp', ')')

    # Step 3: Get the postfix expression of the modified expression
    postfix_expr = infix_to_postfix(expression)

    # Step 4: Reverse the postfix expression to get the prefix expression
    prefix_expr = postfix_expr[::-1]

    return prefix_expr

# Example usage
infix_expr = "A+B*C-(D/E^F)*G"
prefix_expr = infix_to_prefix(infix_expr)
print("Infix Expression: ", infix_expr)
print("Prefix Expression: ", prefix_expr)
```

Next Greater Element

```python
def next_greater_elements(arr):
    stack = []
    result = [-1] * len(arr)  # Initialize the result list with -1

    # Traverse the array from right to left
    for i in range(len(arr) - 1, -1, -1):
        # Pop elements from the stack if they are less than or equal to the
current element
        while stack and stack[-1] <= arr[i]:
            stack.pop()

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

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

    return result

# Example usage
arr = [4, 5, 2, 25]
result = next_greater_elements(arr)
print("Array:", arr)
print("Next Greater Elements:", result)
```

Next Greater Element 2

```python
def next_greater_elements_circular(nums):
    n = len(nums)
    result = [-1] * n
    stack = []

    for i in range(2 * n):
        # We use modulo to wrap around the index
        while stack and nums[stack[-1]] < nums[i % n]:
            result[stack.pop()] = nums[i % n]
        if i < n:
            stack.append(i % n)

    return result

# Example usage
arr = [1, 2, 1]
result = next_greater_elements_circular(arr)
print("Array:", arr)
print("Next Greater Elements:", result)
```

Next Smaller Element

```
def next_smaller_elements(arr):
    stack = []
    result = [-1] * len(arr)  # Initialize the result list with -1

    # Traverse the array from right to left
    for i in range(len(arr) - 1, -1, -1):
        # Pop elements from the stack if they are greater than or equal to
the current element
        while stack and stack[-1] >= arr[i]:
            stack.pop()

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

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

    return result

# Example usage
arr = [4, 8, 5, 2, 25]
result = next_smaller_elements(arr)
print("Array:", arr)
print("Next Smaller Elements:", result)
```

Number of NGEs to the right

```
def number_of_nges_to_the_right(arr):
    stack = []
    count = [0] * len(arr)  # Initialize the count list with 0s

    # Traverse the array from right to left
    for i in range(len(arr) - 1, -1, -1):
        # Count elements that are greater than the current element
        while stack and arr[stack[-1]] <= arr[i]:
            stack.pop()

        # If the stack is not empty, count how many elements are greater
        count[i] = len(stack)

        # Push the current element index onto the stack
        stack.append(i)
```

```python
    return count

# Example usage
arr = [4, 5, 2, 25]
result = number_of_nges_to_the_right(arr)
print("Array:", arr)
print("Number of NGEs to the right:", result)
```

Trapping Rainwater

```python
def trap(height):
    n = len(height)
    if n == 0:
        return 0

    left_max = [0] * n
    right_max = [0] * n

    # Compute left_max array
    left_max[0] = height[0]
    for i in range(1, n):
        left_max[i] = max(left_max[i - 1], height[i])

    # Compute right_max array
    right_max[n - 1] = height[n - 1]
    for i in range(n - 2, -1, -1):
        right_max[i] = max(right_max[i + 1], height[i])

    # Calculate trapped water
    water_trapped = 0
    for i in range(n):
        water_trapped += min(left_max[i], right_max[i]) - height[i]

    return water_trapped

# Example usage:
heights = [0,1,0,2,1,0,1,3,2,1,2,1]
print("Heights:", heights)
print("Trapped Water:", trap(heights))
```

Sum of subarray minimum

```python
def sum_subarray_minimums(arr):
    n = len(arr)
    MOD = 10**9 + 7
```

```
    # Arrays to store the indices of next smaller element to left and right
    left = [-1] * n
    right = [n] * n

    # Use stacks to find next smaller element to left (NSE)
    stack = []
    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)

    # Clear stack for use in finding next smaller element to right (PSE)
    stack = []
    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)

    # Calculate sum of minimum elements in all subarrays
    result = 0
    for i in range(n):
        # Calculate the contribution of arr[i] as minimum element
        left_count = i - left[i]
        right_count = right[i] - i
        result += arr[i] * left_count * right_count
        result %= MOD

    return result

# Example usage:
arr = [3, 1, 2, 4]
print("Array:", arr)
print("Sum of minimum elements of all subarrays:",
sum_subarray_minimums(arr))
```

Asteroid Collision

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

    for asteroid in asteroids:
        # Handle collision for asteroids moving to the left
        while stack and asteroid < 0 and stack[-1] > 0:
            if stack[-1] == -asteroid:
```

```python
                stack.pop()  # Both asteroids explode
                break
            elif stack[-1] > -asteroid:
                break  # Current asteroid explodes
            else:
                stack.pop()  # Top of stack asteroid explodes

        else:
            stack.append(asteroid)

    return stack

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

asteroids = [8, -8]
print("\nOriginal asteroids:", asteroids)
print("After collision:", asteroid_collision(asteroids))

asteroids = [10, 2, -5]
print("\nOriginal asteroids:", asteroids)
print("After collision:", asteroid_collision(asteroids))

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

Sum of subarray ranges

```python
def sum_subarray_ranges(arr):
    n = len(arr)
    total_sum = sum(arr)
    result = 0

    for i in range(n):
        result += arr[i] * (i + 1) * (n - i)

    return result

# Example usage:
arr = [1, 2, 3]
print("Array:", arr)
print("Sum of all subarray ranges:", sum_subarray_ranges(arr))
```

Remove k Digits

```
def removeKdigits(num, k):
    stack = []

    for digit in num:
        while k > 0 and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)

    # If we have remaining k to remove, pop from the stack
    while k > 0:
        stack.pop()
        k -= 1

    # Construct the result number
    result = ''.join(stack).lstrip('0')

    # If result is empty, return '0'
    return result if result else '0'

# Example usage:
num = "1432219"
k = 3
print("Original number:", num)
print("After removing", k, "digits:", removeKdigits(num, k))

num = "10200"
k = 1
print("\nOriginal number:", num)
print("After removing", k, "digits:", removeKdigits(num, k))

num = "10"
k = 2
print("\nOriginal number:", num)
print("After removing", k, "digits:", removeKdigits(num, k))

num = "1234567890"
k = 9
print("\nOriginal number:", num)
print("After removing", k, "digits:", removeKdigits(num, k))
```

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
    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 = [2, 1, 5, 6, 2, 3]
print("Histogram:", histogram)
print("Largest Rectangle Area:", largestRectangleArea(histogram))
```

Maximal Rectangles

```
def maximalRectangle(matrix):
    if not matrix or not matrix[0]:
        return 0

    rows = len(matrix)
    cols = len(matrix[0])
    heights = [0] * (cols + 1)  # extra space to handle the last column
    max_area = 0

    for row in matrix:
        # Calculate heights for the current row
```

```python
        for i in range(cols):
            heights[i] = heights[i] + 1 if row[i] == '1' else 0

        # Calculate the largest rectangle area in the histogram formed by
heights
        stack = []
        for i in range(len(heights)):
            while stack and heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]
                w = i if not stack else i - stack[-1] - 1
                max_area = max(max_area, h * w)
            stack.append(i)

    return max_area

# Example usage:
matrix = [
    ["1","0","1","0","0"],
    ["1","0","1","1","1"],
    ["1","1","1","1","1"],
    ["1","0","0","1","0"]
]
print("Binary Matrix:")
for row in matrix:
    print(row)
print("\nMaximal Rectangle Area:", maximalRectangle(matrix))
```

Sliding Window maximum

```python
from collections import deque

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

    # Deque to store indices of array elements
    deque_idx = deque()
    result = []

    for i in range(n):
    # Remove indices from the front of deque if they are out of current
window range
        if deque_idx and deque_idx[0] == i - k:
            deque_idx.popleft()
```

```python
        # Remove indices from the back of deque while they are smaller than
current element
        while deque_idx and nums[deque_idx[-1]] <= nums[i]:
            deque_idx.pop()

        # Add current index at the back of deque
        deque_idx.append(i)

        # Add maximum element to the result for each window of size k
        if i >= k - 1:
            result.append(nums[deque_idx[0]])

    return result

# Example usage:
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
print("Input Array:", nums)
print("Sliding Window Maximum (k =", k, "):", maxSlidingWindow(nums, k))
```

Stock span problem

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

    for i in range(n):
        # Pop elements from stack while stack is not empty and
prices[stack[-1]] <= prices[i]
        while stack and prices[stack[-1]] <= prices[i]:
            stack.pop()

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

        # Push current day index onto stack
        stack.append(i)

    return span

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

The Celebrity Problem

```python
def findCelebrity(n, matrix):
    left, right = 0, n - 1

    # Step 1: Find potential celebrity using two-pointer approach
    while left < right:
        if matrix[left][right]:
            left += 1  # left cannot be the celebrity
        else:
            right -= 1  # right cannot be the celebrity

    potential_celebrity = left

    # Step 2: Verify potential celebrity
    for i in range(n):
        if i != potential_celebrity:
            if matrix[i][potential_celebrity] or not
matrix[potential_celebrity][i]:
                return -1  # potential_celebrity is not a celebrity

    return potential_celebrity

# Example usage:
n = 4
matrix = [
    [False, False, True, False],
    [False, False, True, False],
    [False, False, False, False],
    [False, False, True, False]
]

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

Sliding Window & Two Pointer

Longest Substring Without Repeating Characters

```python
def length_of_longest_substring(s):
    char_map = {}
    max_length = 0
    start = 0
    left = 0
```

```python
    for right in range(len(s)):
        if s[right] in char_map and char_map[s[right]] >= left:
            left = char_map[s[right]] + 1

        char_map[s[right]] = right
        current_length = right - left + 1

        if current_length > max_length:
            max_length = current_length
            start = left

    return max_length

# Example usage:
input_string = "abcabcbb"
print(length_of_longest_substring(input_string))  # Output: 3 (for "abc")
```

Max Consecutive Ones III

```python
def longest_ones(nums, k):
    left = 0
    zero_count = 0
    max_length = 0

    for right in range(len(nums)):
        if nums[right] == 0:
            zero_count += 1

        while zero_count > k:
            if nums[left] == 0:
                zero_count -= 1
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
nums = [1,1,0,0,1,1,1,0,1,1]
k = 2
print(longest_ones(nums, k))  # Output: 6 (flipping the two '0's to '1's
gives [1,1,1,1,1,1,1,1,1,1])
```

Fruit Into Baskets

```python
def totalFruit(tree):
    fruit_map = {}
    max_fruits = 0
    left = 0

    for right in range(len(tree)):
        fruit = tree[right]
        if fruit in fruit_map:
            fruit_map[fruit] += 1
        else:
            fruit_map[fruit] = 1

        while len(fruit_map) > 2:
            left_fruit = tree[left]
            fruit_map[left_fruit] -= 1
            if fruit_map[left_fruit] == 0:
                del fruit_map[left_fruit]
            left += 1

        max_fruits = max(max_fruits, right - left + 1)

    return max_fruits

# Example usage:
tree = [1,2,1,3,4,3,5,1,2]
print(totalFruit(tree))  # Output: 5 (the longest subarray with at most two
different types of fruits is [3,4,3,5,1])
```

Longest repeating character replacement

```python
def longest_repeating_character_replacement(s, k):
    char_count = [0] * 26
    max_count = 0
    max_length = 0
    left = 0

    for right in range(len(s)):
        char_count[ord(s[right]) - ord('A')] += 1
        max_count = max(max_count, char_count[ord(s[right]) - ord('A')])

        if (right - left + 1) - max_count > k:
            char_count[ord(s[left]) - ord('A')] -= 1
            left += 1

        max_length = max(max_length, right - left + 1)
```

```python
    return max_length

# Example usage:
s = "ABAB"
k = 2
print(longest_repeating_character_replacement(s, k))  # Output: 4 (replace the two 'A's with 'B's to get "BBBB")
```

Binary subarray with sum

```python
def num_subarrays_with_sum(nums, goal):
    prefix_count = {0: 1}  # initialize prefix_count with {0:1} to account for subarrays starting from index 0
    prefix_sum = 0
    count = 0

    for num in nums:
        prefix_sum += num
        needed_prefix = prefix_sum - goal
        if needed_prefix in prefix_count:
            count += prefix_count[needed_prefix]

        if prefix_sum in prefix_count:
            prefix_count[prefix_sum] += 1
        else:
            prefix_count[prefix_sum] = 1

    return count

# Example usage:
nums = [1,0,1,0,1]
goal = 2
print(num_subarrays_with_sum(nums, goal))  # Output: 4 (the subarrays are [1,0,1], [1,0,1,0], [0,1,0,1], [1,0,1])
```

Count number of nice subarrays

```python
def count_nice_subarrays(nums, k):
    def transform_array(nums):
        return [1 if num % 2 == 1 else 0 for num in nums]

    transformed_nums = transform_array(nums)
    left = 0
    count_odd = 0
    count_nice = 0
```

```
    for right in range(len(transformed_nums)):
        if transformed_nums[right] == 1:
            count_odd += 1

        while count_odd > k:
            if transformed_nums[left] == 1:
                count_odd -= 1
            left += 1

        if count_odd == k:
            count_nice += 1

    return count_nice

# Example usage:
nums = [1,1,2,1,1]
k = 3
print(count_nice_subarrays(nums, k))  # Output: 2 (the nice subarrays are
[1,1,2,1] and [2,1,1])
```

Number of substring containing all three characters

```
def numberOfSubstrings(s):
    char_count = {'a': 0, 'b': 0, 'c': 0}
    left = 0
    count = 0

    for right in range(len(s)):
        char_count[s[right]] += 1

        while char_count['a'] > 0 and char_count['b'] > 0 and
char_count['c'] > 0:
            count += len(s) - right
            char_count[s[left]] -= 1
            left += 1

    return count

# Example usage:
s = "abcabc"
print(numberOfSubstrings(s))  # Output: 10 (the substrings are "abc",
"abca", "abcab", "abcabc", "bcab", "bcabc", "cab", "cabc", "ab", "abc")
```

Maximum point you can obtain from cards

```python
def maxScore(cardPoints, k):
    n = len(cardPoints)
    total_sum = sum(cardPoints)
    window_size = n - k

    # Compute the sum of the first window of size (n - k)
    current_sum = sum(cardPoints[:window_size])
    min_sum = current_sum

    # Sliding the window across the array to find the minimum sum
    for i in range(window_size, n):
        current_sum += cardPoints[i] - cardPoints[i - window_size]
        min_sum = min(min_sum, current_sum)

    # Maximum points is total sum minus the minimum sum found
    return total_sum - min_sum

# Example usage:
cardPoints = [1, 2, 3, 4, 5, 6, 1]
k = 3
print(maxScore(cardPoints, k))  # Output: 12 (pick cards from the beginning
[1, 2, 3] and from the end [1])
```

Longest Substring with At Most K Distinct Characters

```python
def lengthOfLongestSubstringKDistinct(s, k):
    char_count = {}
    distinct_count = 0
    max_length = 0
    left = 0

    for right in range(len(s)):
        # Update char_count for s[right]
        if s[right] in char_count:
            char_count[s[right]] += 1
        else:
            char_count[s[right]] = 1
            distinct_count += 1

        # Shrink the window if distinct_count exceeds k
        while distinct_count > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                distinct_count -= 1
                del char_count[s[left]]
            left += 1
```

```
        # Update max_length if current window size is larger
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
s = "eceba"
k = 2
print(lengthOfLongestSubstringKDistinct(s, k))  # Output: 3 (the longest
substring with at most 2 distinct characters is "ece")
```

Subarray with k different integers

```
def subarraysWithKDistinct(A, K):
    def atMostK(A, K):
        count_map = {}
        left = 0
        result = 0
        for right in range(len(A)):
            if A[right] in count_map:
                count_map[A[right]] += 1
            else:
                count_map[A[right]] = 1

            while len(count_map) > K:
                count_map[A[left]] -= 1
                if count_map[A[left]] == 0:
                    del count_map[A[left]]
                left += 1

            # All subarrays ending at right and having at most K distinct
elements
            result += right - left + 1

        return result

    return atMostK(A, K) - atMostK(A, K - 1)

# Example usage:
A = [1, 2, 1, 2, 3]
K = 2
print(subarraysWithKDistinct(A, K))  # Output: 7 (subarrays with exactly 2
distinct integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2])
```

Minimum Window Substring

```python
import collections

def minWindow(s, t):
    if not s or not t:
        return ""

    target_map = collections.Counter(t)
    required_chars = len(target_map)

    left, right = 0, 0
    formed = 0
    window_map = {}
    min_length = float('inf')
    min_window = ""

    while right < len(s):
        char = s[right]
        window_map[char] = window_map.get(char, 0) + 1

        if char in target_map and window_map[char] == target_map[char]:
            formed += 1

        while formed == required_chars and left <= right:
            current_length = right - left + 1
            if current_length < min_length:
                min_length = current_length
                min_window = s[left:right + 1]

            left_char = s[left]
            window_map[left_char] -= 1
            if left_char in target_map and window_map[left_char] <
target_map[left_char]:
                formed -= 1

            left += 1

        right += 1

    return min_window

# Example usage:
s = "ADOBECODEBANC"
t = "ABC"
print(minWindow(s, t))  # Output: "BANC"
```

Minimum Window Subsequence

```python
def minWindowSubsequence(S, T):
    n, m = len(S), len(T)
    dp = [[-1] * m for _ in range(n)]

    # Initialize dp table for the first character of T
    if S[0] == T[0]:
        dp[0][0] = 0

    # Fill dp table
    for i in range(1, n):
        if S[i] == T[0]:
            dp[i][0] = i
        else:
            dp[i][0] = dp[i-1][0]

    for j in range(1, m):
        last_match = -1
        for i in range(n):
            if S[i] == T[j]:
                dp[i][j] = dp[last_match][j-1] if last_match != -1 else i
                last_match = dp[last_match][j-1] if last_match != -1 else i
            else:
                dp[i][j] = dp[i-1][j]

    # Find minimum window from the last row of dp table
    min_length = float('inf')
    start = -1

    for i in range(n):
        if dp[i][m-1] != -1:
            current_length = i - dp[i][m-1] + 1
            if current_length < min_length:
                min_length = current_length
                start = dp[i][m-1]

    if min_length == float('inf'):
        return ""
    else:
        return S[start:start + min_length]

# Example usage:
S = "abcdebdde"
T = "bde"
print(minWindowSubsequence(S, T))  # Output: "bcde"
```

Heaps

Introduction to Priority Queues using Binary Heaps

```python
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, item):
        # Append the new element to the end of the heap
        self.heap.append(item)
        i = len(self.heap) - 1

        # Fix the min-heap property if it's violated
        while i != 0 and self.heap[self.parent(i)] > self.heap[i]:
            # Swap parent and current node
            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 len(self.heap) == 0:
            return None

        # Remove the root (minimum element)
        min_element = self.heap[0]
        last_element = self.heap.pop()

        if len(self.heap) > 0:
            # Move the last element to the root and heapify
            self.heap[0] = last_element
            self.min_heapify(0)

        return min_element

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

```
        # Compare with left child
        if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
            smallest = left

        # Compare with right child
        if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
            smallest = right

        # If the smallest is not the current node, swap and continue
heapifying
        if smallest != i:
            self.heap[i], self.heap[smallest] = self.heap[smallest],
self.heap[i]
            self.min_heapify(smallest)

    def peek_min(self):
        if len(self.heap) > 0:
            return self.heap[0]
        return None

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

    def is_empty(self):
        return len(self.heap) == 0
```

Min Heap and Max Heap Implementation

```
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, item):
        self.heap.append(item)
        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[self.parent(i)], self.heap[i] = self.heap[i],
self.heap[self.parent(i)]
            i = self.parent(i)

    def extract_min(self):
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        min_element = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return min_element

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

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

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

    def is_empty(self):
        return len(self.heap) == 0
```

Check if an array represents a min-heap or not

```
def is_min_heap(arr):
    n = len(arr)
    # Check every node (from 0 to n//2 - 1) as those are the internal nodes
```

```
    for i in range(n // 2):
        left_child = 2 * i + 1
        right_child = 2 * i + 2

        # Check if left child exists and if it's smaller than parent
        if left_child < n and arr[left_child] < arr[i]:
            return False

        # Check if right child exists and if it's smaller than parent
        if right_child < n and arr[right_child] < arr[i]:
            return False

    return True

# Example usage:
arr1 = [1, 3, 5, 8, 10, 6]
arr2 = [2, 3, 6, 8, 10, 5]

print(is_min_heap(arr1))  # Output: True (arr1 represents a min-heap)
print(is_min_heap(arr2))  # Output: False (arr2 does not represent a min-
heap)
```

Convert min Heap to max Heap

```
def max_heapify(arr, n, i):
   largest = i
   left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)

def min_heap_to_max_heap(arr):
    n = len(arr)

    # Start from the last non-leaf node and heapify each node
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

# Example usage:
```

```python
min_heap = [1, 3, 5, 8, 10, 6]
print("Min-Heap before conversion:", min_heap)

min_heap_to_max_heap(min_heap)
print("Max-Heap after conversion:", min_heap)
```

Kth largest element in an array [use priority queue]

```python
import heapq

def find_kth_largest(nums, k):
    # Min-heap implementation using heapq in Python (which is a min-heap by default)
    min_heap = []

    # Populate the heap with the first K elements
    for num in nums[:k]:
        heapq.heappush(min_heap, num)

    # Traverse through the rest of the elements in the array
    for num in nums[k:]:
        # If current element is larger than the root (smallest element in heap), replace it
        if num > min_heap[0]:
            heapq.heapreplace(min_heap, num)

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

# Example usage:
nums = [3, 2, 1, 5, 6, 4]  # Sample array
k = 2  # Kth largest element we want to find

result = find_kth_largest(nums, k)
print(f"The {k}th largest element in the array is: {result}")  # Output: 5
```

Kth smallest element in an array [use priority queue]

```python
import heapq

def find_kth_smallest(nums, k):
    # Max-heap implementation using negative values for heapq to simulate max-heap behavior
    max_heap = []
```

```
    # Populate the heap with the first K elements (negate them to simulate
max-heap)
    for num in nums[:k]:
        heapq.heappush(max_heap, -num)

    # Traverse through the rest of the elements in the array
    for num in nums[k:]:
        # If current element is smaller than the largest element in heap,
replace it
        if num < -max_heap[0]:
            heapq.heapreplace(max_heap, -num)

    # The root of the max-heap (negated) is the Kth smallest element
    return -max_heap[0]

# Example usage:
nums = [3, 2, 1, 5, 6, 4]  # Sample array
k = 2  # Kth smallest element we want to find

result = find_kth_smallest(nums, k)
print(f"The {k}th smallest element in the array is: {result}")  # Output: 2
```

Sort K sorted array

```
import heapq

def sort_k_sorted_array(arr, k):
    n = len(arr)
    sorted_arr = []
    min_heap = []

    # Populate min-heap with the first K+1 elements
    for i in range(min(k + 1, n)):
        heapq.heappush(min_heap, arr[i])

    index = 0
    # Process remaining elements in the array
    for i in range(k + 1, n):
        sorted_arr.append(heapq.heappop(min_heap))
        heapq.heappush(min_heap, arr[i])

    # Extract remaining elements from the min-heap
    while min_heap:
        sorted_arr.append(heapq.heappop(min_heap))

    return sorted_arr
```

```python
# Example usage:
arr = [6, 5, 3, 2, 8, 10, 9]
k = 2
sorted_array = sort_k_sorted_array(arr, k)
print("Sorted K-sorted array:", sorted_array)
```

Merge M sorted Lists

```python
import heapq

def merge_m_sorted_lists(lists):
    min_heap = []
    merged_result = []

    # Initialize the min-heap with the first element from each list
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(min_heap, (lst[0], i, 0))  # (element,
list_index, index_in_list)

    while min_heap:
        element, list_index, index_in_list = heapq.heappop(min_heap)
        merged_result.append(element)

        # Move to the next element in the same list
        next_index = index_in_list + 1
        if next_index < len(lists[list_index]):
            heapq.heappush(min_heap, (lists[list_index][next_index],
list_index, next_index))

    return merged_result

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

merged_result = merge_m_sorted_lists(lists)
print("Merged sorted lists:", merged_result)
```

Replace each array element by its corresponding rank

```python
def replace_by_rank(arr):
    # Create a sorted copy of the array
```

```python
    sorted_arr = sorted(arr)

    # Create a dictionary to store element -> rank mapping
    rank_map = {}

    # Assign ranks based on sorted order
    rank = 1
    for num in sorted_arr:
        if num not in rank_map:
            rank_map[num] = rank
            rank += 1

    # Replace each element in the original array with its rank
    for i in range(len(arr)):
        arr[i] = rank_map[arr[i]]

    return arr

# Example usage:
arr = [10, 8, 15, 12, 6, 20]
result = replace_by_rank(arr)
print("Original Array:", arr)
print("Array after replacing with ranks:", result)
```

Task Scheduler

```python
import heapq
from collections import defaultdict

def least_interval(tasks, n):
    # Step 1: Count frequency of each task
    freq = defaultdict(int)
    for task in tasks:
        freq[task] += 1

    # Step 2: Create max-heap (negative frequencies for max-heap simulation)
    max_heap = []
    for task, f in freq.items():
        heapq.heappush(max_heap, (-f, task))

    total_time = 0
    while max_heap:
        # Step 3: Execute tasks in order of their frequency (max heap)
        k = n + 1  # cooldown interval + 1 (slots available)
        temp = []

        for _ in range(k):
```

```
            total_time += 1
            if max_heap:
                f, task = heapq.heappop(max_heap)
                if f < -1:
                    temp.append((f + 1, task))  # decrease frequency and add
back to heap

        # Step 4: Push back tasks that are not yet done
        for item in temp:
            heapq.heappush(max_heap, item)

        # Step 5: If max_heap is empty, we've finished scheduling all tasks
        if not max_heap:
            break

        # Step 6: If k > 0 but temp was exhausted, add idle time
        total_time += k - len(temp)  # if temp is not exhausted, then k is
enough for the current cycle

    return total_time

# Example usage:
tasks = ["A", "A", "A", "B", "B", "B"]
n = 2
result = least_interval(tasks, n)
print("Total time:", result)
```

Hands of Straights

```
from collections import defaultdict

def isNStraightHand(hand, W):
    if len(hand) % W != 0:
        return False

    count = defaultdict(int)
    for card in hand:
        count[card] += 1

    hand.sort()

    for card in hand:
        if count[card] > 0:
            for i in range(W):
                if count[card + i] <= 0:
                    return False
                count[card + i] -= 1
```

```
    return True

# Example usage:
hand = [1, 2, 3, 6, 2, 3, 4, 7, 8]
W = 3
print(isNStraightHand(hand, W))  # Output: True
```

Connect `n` ropes with minimal cost

```
import heapq

def min_cost_to_connect_ropes(ropes):
    if not ropes:
        return 0

    # Min-heap initialization
    heapq.heapify(ropes)

    total_cost = 0

    # Continue until only one rope is left in the heap
    while len(ropes) > 1:
        # Extract the two smallest ropes
        first = heapq.heappop(ropes)
        second = heapq.heappop(ropes)

        # Combine them
        combined_cost = first + second
        total_cost += combined_cost

        # Insert the combined rope back into the heap
        heapq.heappush(ropes, combined_cost)

    return total_cost

# Example usage:
ropes = [8, 4, 6, 12]
result = min_cost_to_connect_ropes(ropes)
print("Minimum cost to connect all ropes:", result)
```

Kth largest element in a stream of running integers

```
import heapq

class KthLargestInStream:
```

```
    def __init__(self, k):
        self.max_heap = []
        self.k = k

    def add(self, num):
        # If max_heap has less than k elements, add num directly
        if len(self.max_heap) < self.k:
            heapq.heappush(self.max_heap, num)
        else:
            # If num is larger than the smallest element (root) of max_heap
            if num > self.max_heap[0]:
                heapq.heapreplace(self.max_heap, num)

    def get_kth_largest(self):
        return self.max_heap[0]

# Example usage:
kth_largest_stream = KthLargestInStream(3)
nums = [4, 2, 6, 1, 5, 3]
for num in nums:
    kth_largest_stream.add(num)

print("Kth largest element:", kth_largest_stream.get_kth_largest())  #
Output should be 4
```

Maximum Sum Combination

```
import heapq

def max_sum_combination(A, B, k):
    # Sort arrays A and B in descending order
    A.sort(reverse=True)
    B.sort(reverse=True)

    max_heap = []
    seen_pairs = set()
    result = []

    # Initialize with the maximum sum combination from the largest elements
    heapq.heappush(max_heap, (-(A[0] + B[0]), 0, 0))
    seen_pairs.add((0, 0))

    # Process k maximum sums
    while len(result) < k:
        neg_sum, i, j = heapq.heappop(max_heap)
        result.append(-neg_sum)
```

```python
        # Generate next potential pairs (i+1, j) and (i, j+1) if not already
seen
        if i + 1 < len(A) and (i + 1, j) not in seen_pairs:
            heapq.heappush(max_heap, (-(A[i + 1] + B[j]), i + 1, j))
            seen_pairs.add((i + 1, j))
        if j + 1 < len(B) and (i, j + 1) not in seen_pairs:
            heapq.heappush(max_heap, (-(A[i] + B[j + 1]), i, j + 1))
            seen_pairs.add((i, j + 1))

    return result

# Example usage:
A = [4, 2, 5, 1]
B = [8, 0, 3, 5]
k = 3
print("Maximum sum combinations:", max_sum_combination(A, B, k))  # Output:
[13, 12, 10]
```

Find Median from Data Stream

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.left = []  # Max-heap (negated values)
        self.right = []  # Min-heap
        self.count = 0

    def addNum(self, num: int) -> None:
        heapq.heappush(self.left, -num)
        heapq.heappush(self.right, -heapq.heappop(self.left))

        if len(self.left) < len(self.right):
            heapq.heappush(self.left, -heapq.heappop(self.right))

        self.count += 1

    def findMedian(self) -> float:
        if self.count % 2 == 1:
            return -self.left[0]
        else:
            return (-self.left[0] + self.right[0]) / 2

# Example usage:
median_finder = MedianFinder()
nums = [1, 3, 2, 5, 4]
for num in nums:
```

```
    median_finder.addNum(num)
    print("Median after adding", num, "is:", median_finder.findMedian())
```

K most frequent elements

```
import heapq
from collections import Counter

def topKFrequent(nums, k):
    # Step 1: Count frequencies using Counter
    freq_map = Counter(nums)

    # Step 2: Use a min-heap to keep track of K most frequent elements
    min_heap = []

    for num, freq in freq_map.items():
        if len(min_heap) < k:
            heapq.heappush(min_heap, (freq, num))
        else:
            if freq > min_heap[0][0]:
                heapq.heapreplace(min_heap, (freq, num))

    # Step 3: Extract top K elements from the heap
    result = []
    while min_heap:
        result.append(heapq.heappop(min_heap)[1])

    return result

# Example usage:
nums = [1, 1, 1, 2, 2, 3]
k = 2
print("Top", k, "frequent elements:", topKFrequent(nums, k))  # Output: [1,
2]
```

Greedy Algorithms

Assign Cookies

```
def findContentChildren(g, s):
    g.sort()
    s.sort()

    i, j = 0, 0
```

```python
    satisfied = 0

    while i < len(g) and j < len(s):
        if s[j] >= g[i]:
            satisfied += 1
            i += 1
        j += 1

    return satisfied

g = [1, 2, 3]
s = [1, 1, 2]
print(findContentChildren(g, s))  # Output: 2
```

Fractional Knapsack Problem

```python
def fractional_knapsack(W, items):
    # Compute value-to-weight ratio for each item
    for item in items:
        item['ratio'] = item['value'] / item['weight']

    # Sort items by ratio (descending order)
    items.sort(key=lambda x: x['ratio'], reverse=True)

    total_value = 0.0
    remaining_capacity = W

    for item in items:
        if remaining_capacity <= 0:
            break

        if item['weight'] <= remaining_capacity:
            total_value += item['value']
            remaining_capacity -= item['weight']
        else:
            total_value += item['ratio'] * remaining_capacity
            remaining_capacity = 0

    return total_value

# Example usage:
items = [
    {'weight': 10, 'value': 60},
    {'weight': 20, 'value': 100},
    {'weight': 30, 'value': 120}
]
```

```
knapsack_capacity = 50
max_value = fractional_knapsack(knapsack_capacity, items)
print("Maximum value in knapsack:", max_value)  # Output: 80.0
```

Greedy algorithm to find minimum number of coins

```
def min_coins_greedy(amount, coins):
    coins.sort(reverse=True)  # Sort coins in descending order
    coin_count = 0

    for coin in coins:
        if amount <= 0:
            break
        coin_count += amount // coin
        amount %= coin

    return coin_count

# Example usage:
coins = [25, 10, 5, 1]
amount = 30
min_coins = min_coins_greedy(amount, coins)
print(f"Minimum number of coins needed for {amount} cents: {min_coins}")  #
Output: 2
```

Lemonade Change

```
def lemonadeChange(bills):
    # Initialize bill counts
    count5 = 0
    count10 = 0

    for bill in bills:
        if bill == 5:
            count5 += 1
        elif bill == 10:
            if count5 > 0:
                count5 -= 1
                count10 += 1
            else:
                return False
        elif bill == 20:
            if count10 > 0 and count5 > 0:
                count10 -= 1
                count5 -= 1
            elif count5 >= 3:
```

```python
                count5 -= 3
            else:
                return False

    return True

# Example usage:
bills = [5, 5, 5, 10, 20]
print(lemonadeChange(bills))  # Output: True

bills = [5, 5, 10, 10, 20]
print(lemonadeChange(bills))  # Output: False
```

Valid Paranthesis Checker

```python
def isValid(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping.values():
            stack.append(char)
        elif char in mapping.keys():
            if stack and stack[-1] == mapping[char]:
                stack.pop()
            else:
                return False
        else:
            return False

    return len(stack) == 0

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

N meetings in one room

```python
def max_meetings(start_time, end_time):
    # Combine start and end times
    meetings = list(zip(start_time, end_time))

    # Sort meetings by end time
```

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

    # Select meetings greedily
    selected_meetings = []
    end_time = 0

    for meeting in meetings:
        if meeting[0] >= end_time:
            selected_meetings.append(meeting)
            end_time = meeting[1]

    return selected_meetings

# Example usage:
start_time = [1, 3, 0, 5, 8, 5, 8]
end_time = [2, 4, 6, 7, 9, 9, 10]

selected_meetings = max_meetings(start_time, end_time)
print("Selected meetings:", selected_meetings)
print("Number of meetings:", len(selected_meetings))  # Output: 4
```

Jump Game

```python
def canJump(nums):
    max_reach = 0
    n = len(nums)

    for i in range(n):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + nums[i])
        if max_reach >= n - 1:
            return True

    return True  # This line is actually redundant because max_reach should reach n - 1 before the loop ends

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

nums = [3, 2, 1, 0, 4]
print(canJump(nums))  # Output: False
```

Jump Game 2

```python
def jump(nums):
    n = len(nums)
    if n == 1:
        return 0

    jumps = 0
    max_reach = 0
    end = 0

    for i in range(n - 1):
        max_reach = max(max_reach, i + nums[i])

        if i == end:
            jumps += 1
            end = max_reach

            if end >= n - 1:
                break

    return jumps

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

nums = [2, 3, 0, 1, 4]
print(jump(nums))  # Output: 2
```

Minimum number of platforms required for a railway

```python
def minimumPlatforms(arrival, departure):
    n = len(arrival)
    if n != len(departure):
        return -1

    # Sort arrival and departure times
    arrival.sort()
    departure.sort()

    platforms_needed = 0
    max_platforms = 0

    # Two pointers to track arrival and departure times
    i = 0  # pointer for arrival
    j = 0  # pointer for departure

    while i < n and j < n:
```

```python
        if arrival[i] <= departure[j]:
            platforms_needed += 1
            max_platforms = max(max_platforms, platforms_needed)
            i += 1
        else:
            platforms_needed -= 1
            j += 1

    return max_platforms

# Example usage:
arrival = [900, 940, 950, 1100, 1500, 1800]
departure = [910, 1200, 1120, 1130, 1900, 2000]
print(minimumPlatforms(arrival, departure))  # Output: 3
```

Job sequencing Problem

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

    # Find max deadline
    max_deadline = max(jobs, key=lambda x: x[1])[1]

    # Initialize time_slots
    time_slots = [-1] * (max_deadline + 1)
    total_profit = 0
    scheduled_jobs = []

    # Schedule jobs
    for job_id, deadline, profit in jobs:
        # Find the latest available time slot before the deadline
        for j in range(deadline, 0, -1):
            if time_slots[j] == -1:
                time_slots[j] = job_id
                total_profit += profit
                scheduled_jobs.append(job_id)
                break

    return scheduled_jobs, total_profit

# Example usage:
jobs = [
    (1, 4, 20),
    (2, 1, 10),
    (3, 1, 40),
    (4, 1, 30)
```

```
]

scheduled_jobs, total_profit = jobSequencing(jobs)
print("Scheduled Jobs:", scheduled_jobs)  # Output: [3, 1, 4]
print("Total Profit:", total_profit)      # Output: 90
```

Candy

```
def candy(ratings):
    n = len(ratings)
    if n == 0:
        return 0

    # Initialize candies array with all 1's
    candies = [1] * n

    # Left to right pass
    for i in range(1, n):
        if ratings[i] > ratings[i-1]:
            candies[i] = candies[i-1] + 1

    # Right to left pass
    for i in range(n-2, -1, -1):
        if ratings[i] > ratings[i+1] and candies[i] <= candies[i+1]:
            candies[i] = candies[i+1] + 1

    # Calculate total candies
    total_candies = sum(candies)

    return total_candies

# Example usage:
ratings = [1, 0, 2]
print(candy(ratings))  # Output: 5
```

Program for Shortest Job First (or SJF) CPU Scheduling

```
def sjf(processes, n):
    # Sort processes based on burst time
    processes.sort(key=lambda x: x[1])  # x[1] is burst time

    # Initialize waiting time and total waiting time
    waiting_time = [0] * n
    total_waiting_time = 0

    # Calculate waiting time for each process
```

```python
    for i in range(1, n):
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1][1]
        total_waiting_time += waiting_time[i]

    # Calculate average waiting time
    average_waiting_time = total_waiting_time / n

    # Print the results
    print("Process\tBurst Time\tWaiting Time")
    for i in range(n):
        print(f"{processes[i][0]}\t{processes[i][1]}\t\t{waiting_time[i]}")

    print(f"\nAverage Waiting Time: {average_waiting_time}")

# Example usage:
if __name__ == "__main__":
    # Example processes: (process_id, burst_time)
    processes = [
        (1, 6),
        (2, 8),
        (3, 7),
        (4, 3),
    ]
    n = len(processes)
    sjf(processes, n)
```

Program for Least Recently Used (LRU) Page Replacement Algorithm

```python
from collections import deque

def lru_page_replacement(pages, page_frames):
    queue = deque()
    page_set = set()
    page_faults = 0

    for page in pages:
        if page in page_set:
            # Page already in memory, update its recent usage
            queue.remove(page)
            queue.append(page)
        else:
            # Page not in memory, handle page fault
            if len(queue) == page_frames:
                # Remove least recently used page
                removed_page = queue.popleft()
                page_set.remove(removed_page)
            # Add new page to memory
```

```python
            queue.append(page)
            page_set.add(page)
            page_faults += 1

    return page_faults

# Example usage:
if __name__ == "__main__":
    pages = [1, 3, 0, 3, 5, 6, 3]
    page_frames = 3
    page_faults = lru_page_replacement(pages, page_frames)
    print("Total Page Faults:", page_faults)  # Output: 5
```

Insert Interval

```python
def insertInterval(intervals, newInterval):
    result = []
    i = 0
    n = len(intervals)

    # Add all intervals that come before newInterval
    while i < n and intervals[i][1] < newInterval[0]:
        result.append(intervals[i])
        i += 1

    # Merge intervals that overlap with newInterval
    while i < n and intervals[i][0] <= newInterval[1]:
        newInterval[0] = min(newInterval[0], intervals[i][0])
        newInterval[1] = max(newInterval[1], intervals[i][1])
        i += 1

    result.append(newInterval)

    # Add remaining intervals
    while i < n:
        result.append(intervals[i])
        i += 1

    return result

# Example usage:
intervals = [[1,3], [6,9]]
newInterval = [2,5]
print(insertInterval(intervals, newInterval))  # Output: [[1, 5], [6, 9]]
```

Merge Intervals

```
def mergeIntervals(intervals):
    # Sort intervals by the start time
    intervals.sort(key=lambda x: x[0])

    merged_intervals = []

    for interval in intervals:
        if not merged_intervals or merged_intervals[-1][1] < interval[0]:
            # No overlap, add the interval directly
            merged_intervals.append(interval)
        else:
            # Overlap exists, merge the intervals
            merged_intervals[-1][1] = max(merged_intervals[-1][1],
interval[1])

    return merged_intervals

# Example usage:
intervals = [[1,3], [2,6], [8,10], [15,18]]
print(mergeIntervals(intervals))  # Output: [[1, 6], [8, 10], [15, 18]]
```

Non-overlapping Intervals

```
def eraseOverlapIntervals(intervals):
    if not intervals:
        return 0

    # Sort intervals by end time
    intervals.sort(key=lambda x: x[1])

    # Initialize variables
    end = intervals[0][1]
    count = 0

    # Iterate through intervals
    for i in range(1, len(intervals)):
        if intervals[i][0] < end:  # Overlapping intervals
            count += 1
        else:  # Non-overlapping interval found
            end = intervals[i][1]

    return count

# Example usage:
intervals = [[1,2], [2,3], [3,4], [1,3]]
```

```
print(eraseOverlapIntervals(intervals))  # Output: 1
```

Binary Trees

Binary Tree Representation In Python

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node:
            self._inorder(node.left, result)
            result.append(node.val)
            self._inorder(node.right, result)

# Example usage:
if __name__ == "__main__":
```

```python
    tree = BinaryTree()
    tree.insert(5)
    tree.insert(3)
    tree.insert(7)
    tree.insert(2)
    tree.insert(4)

    print("Inorder traversal:", tree.inorder_traversal())  # Output: [2, 3, 4, 5, 7]
```

Binary Tree Traversals in Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node:
```

```
            self._inorder(node.left, result)
            result.append(node.val)
            self._inorder(node.right, result)

    def preorder_traversal(self):
        result = []
        self._preorder(self.root, result)
        return result

    def _preorder(self, node, result):
        if node:
            result.append(node.val)
            self._preorder(node.left, result)
            self._preorder(node.right, result)

    def postorder_traversal(self):
        result = []
        self._postorder(self.root, result)
        return result

    def _postorder(self, node, result):
        if node:
            self._postorder(node.left, result)
            self._postorder(node.right, result)
            result.append(node.val)

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(5)
    tree.insert(3)
    tree.insert(7)
    tree.insert(2)
    tree.insert(4)

    print("Inorder traversal:", tree.inorder_traversal())    # Output: [2,
3, 4, 5, 7]
    print("Preorder traversal:", tree.preorder_traversal())  # Output: [5,
3, 2, 4, 7]
    print("Postorder traversal:", tree.postorder_traversal())# Output: [2,
4, 3, 7, 5]
```

Height of a Binary Tree

```
class TreeNode:
    def __init__(self, value):
        self.val = value
```

```python
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def height(self):
        return self._height_recursive(self.root)

    def _height_recursive(self, node):
        if node is None:
            return 0
        else:
            left_height = self._height_recursive(node.left)
            right_height = self._height_recursive(node.right)
            return max(left_height, right_height) + 1

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(5)
    tree.insert(3)
    tree.insert(7)
    tree.insert(2)
    tree.insert(4)

    print("Height of the binary tree:", tree.height())  # Output: 3
```

Check if the Binary tree is height-balanced or not

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def is_balanced(self):
        return self._is_balanced_recursive(self.root)[0]

    def _is_balanced_recursive(self, node):
        if node is None:
            return True, 0

        left_balanced, left_height = self._is_balanced_recursive(node.left)
        right_balanced, right_height =
self._is_balanced_recursive(node.right)

        current_balanced = left_balanced and right_balanced and
abs(left_height -   right_height) <= 1
        current_height = max(left_height, right_height) + 1

        return current_balanced, current_height

# Example usage:
if __name__ == "__main__":
```

```python
    tree = BinaryTree()
    tree.insert(5)
    tree.insert(3)
    tree.insert(7)
    tree.insert(2)
    tree.insert(4)

    print("Is the binary tree balanced?", tree.is_balanced())  # Output:
True
```

Diameter of Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def diameter_of_binary_tree(self):
        return self._diameter_recursive(self.root)[0]

    def _diameter_recursive(self, node):
        if node is None:
            return 0, 0
```

```
        left_diameter, left_height = self._diameter_recursive(node.left)
        right_diameter, right_height = self._diameter_recursive(node.right)

        current_height = max(left_height, right_height) + 1
        current_diameter = max(left_height + right_height, left_diameter,
right_diameter)

        return current_diameter, current_height

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
    tree.insert(5)

    print("Diameter of the binary tree:", tree.diameter_of_binary_tree())  #
Output: 3
```

Maximum path sum

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

class BinaryTree:
    def __init__(self):
        self.root = None
        self.max_sum = float('-inf')  # Initialize max_sum to negative
infinity

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
```

```python
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def max_path_sum(self):
        self._max_path_sum_recursive(self.root)
        return self.max_sum

    def _max_path_sum_recursive(self, node):
        if node is None:
            return 0

        left_sum = max(self._max_path_sum_recursive(node.left), 0)
        right_sum = max(self._max_path_sum_recursive(node.right), 0)

        current_sum = node.val + left_sum + right_sum

        self.max_sum = max(self.max_sum, current_sum)

        return node.val + max(left_sum, right_sum)

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)

    print("Maximum path sum in the binary tree:", tree.max_path_sum())  #
Output: 6
```

Check if two trees are identical or not

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
```

```python
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

def are_identical(root1, root2):
    if root1 is None and root2 is None:
        return True
    if root1 is not None and root2 is not None:
        return (root1.val == root2.val and
                are_identical(root1.left, root2.left) and
                are_identical(root1.right, root2.right))
    return False

# Example usage:
if __name__ == "__main__":
    tree1 = BinaryTree()
    tree1.insert(1)
    tree1.insert(2)
    tree1.insert(3)

    tree2 = BinaryTree()
    tree2.insert(1)
    tree2.insert(2)
    tree2.insert(3)

    print("Are the two binary trees identical?", are_identical(tree1.root,
tree2.root))  # Output: True

    tree3 = BinaryTree()
    tree3.insert(1)
    tree3.insert(3)
    tree3.insert(2)

    print("Are the two binary trees identical?", are_identical(tree1.root,
tree3.root))  # Output: False
```

Zig Zag Traversal of Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def zigzag_traversal(self):
        if not self.root:
            return []

        result = []
        current_level = [self.root]
        next_level = []
        left_to_right = True

        while current_level:
            level_values = []
            while current_level:
                node = current_level.pop()
                level_values.append(node.val)

                if left_to_right:
                    if node.left:
                        next_level.append(node.left)
                    if node.right:
```

```python
                        next_level.append(node.right)
                else:
                    if node.right:
                        next_level.append(node.right)
                    if node.left:
                        next_level.append(node.left)

            result.append(level_values)
            current_level, next_level = next_level, current_level
            left_to_right = not left_to_right

        return result

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
    tree.insert(5)
    tree.insert(6)
    tree.insert(7)

    print("Zig Zag Traversal of the binary tree:", tree.zigzag_traversal())
    # Output: [[1], [3, 2], [4, 5, 6, 7]]
```

Boundary Traversal of Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
```

```
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def boundary_traversal(self):
        if not self.root:
            return []

        result = []

        def is_leaf(node):
            return node.left is None and node.right is None

        def add_left_boundary(node):
            while node:
                if not is_leaf(node):
                    result.append(node.val)
                if node.left:
                    node = node.left
                else:
                    node = node.right

        def add_leaves(node):
            if node is None:
                return
            if is_leaf(node):
                result.append(node.val)
            add_leaves(node.left)
            add_leaves(node.right)

        def add_right_boundary(node):
            stack = []
            while node:
                if not is_leaf(node):
                    stack.append(node.val)
                if node.right:
                    node = node.right
                else:
                    node = node.left
            while stack:
                result.append(stack.pop())
```

```
        if not is_leaf(self.root):
            result.append(self.root.val)

        add_left_boundary(self.root.left)
        add_leaves(self.root)
        add_right_boundary(self.root.right)

        return result

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(20)
    tree.insert(8)
    tree.insert(22)
    tree.insert(4)
    tree.insert(12)
    tree.insert(10)
    tree.insert(14)
    tree.insert(25)

    print("Boundary Traversal of the binary tree:",
tree.boundary_traversal())
    # Output: [20, 8, 4, 10, 14, 25, 22]
```

Vertical Order Traversal of Binary Tree

```
from collections import defaultdict, deque

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
```

```python
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

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

        column_table = defaultdict(list)
        queue = deque([(self.root, 0, 0)])  # (node, column, level)

        while queue:
            node, column, level = queue.popleft()
            if node is not None:
                column_table[column].append((level, node.val))
                queue.append((node.left, column - 1, level + 1))
                queue.append((node.right, column + 1, level + 1))

        sorted_columns = sorted(column_table.keys())
        result = []
        for column in sorted_columns:
            column_table[column].sort(key=lambda x: (x[0], x[1]))  # Sort by
level, then value
            column_values = [val for level, val in column_table[column]]
            result.append(column_values)

        return result

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(3)
    tree.insert(9)
    tree.insert(20)
    tree.insert(15)
    tree.insert(7)

    print("Vertical Order Traversal of the binary tree:",
tree.vertical_order_traversal())
    # Output: [[9], [3, 15], [20], [7]]
```

Top View of Binary Tree

```
from collections import deque, defaultdict

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

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

        # Dictionary to store the top view of the binary tree
        top_view_map = {}
        # Queue for level order traversal; stores (node, horizontal
distance)
        queue = deque([(self.root, 0)])

        while queue:
            node, hd = queue.popleft()
            # If the horizontal distance is encountered for the first time
            if hd not in top_view_map:
                top_view_map[hd] = node.val
```

```
            # Enqueue left and right children with their respective
horizontal distances
            if node.left:
                queue.append((node.left, hd - 1))
            if node.right:
                queue.append((node.right, hd + 1))

        # Extracting the values in order of their horizontal distances
        sorted_hd_keys = sorted(top_view_map.keys())
        return [top_view_map[hd] for hd in sorted_hd_keys]

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(20)
    tree.insert(8)
    tree.insert(22)
    tree.insert(4)
    tree.insert(12)
    tree.insert(10)
    tree.insert(14)
    tree.insert(25)

    print("Top View of the binary tree:", tree.top_view())
    # Output: [4, 8, 20, 22, 25]
```

Bottom View of Binary Tree

```
from collections import deque, defaultdict

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
```

```
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

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

        # Dictionary to store the bottom view of the binary tree
        bottom_view_map = {}
        # Queue for level order traversal; stores (node, horizontal
distance)
        queue = deque([(self.root, 0)])

        while queue:
            node, hd = queue.popleft()
            # Update the bottom view map with the current node
            bottom_view_map[hd] = node.val

            # Enqueue left and right children with their respective
horizontal distances
            if node.left:
                queue.append((node.left, hd - 1))
            if node.right:
                queue.append((node.right, hd + 1))

        # Extracting the values in order of their horizontal distances
        sorted_hd_keys = sorted(bottom_view_map.keys())
        return [bottom_view_map[hd] for hd in sorted_hd_keys]

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(20)
    tree.insert(8)
    tree.insert(22)
    tree.insert(4)
    tree.insert(12)
    tree.insert(10)
    tree.insert(14)
    tree.insert(25)
```

```python
    print("Bottom View of the binary tree:", tree.bottom_view())
    # Output: [4, 10, 12, 14, 25]
```

Right/Left View of Binary Tree

```python
from collections import deque

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def right_view(self):
        if not self.root:
            return []

        right_view_result = []
        queue = deque([(self.root, 0)])  # (node, level)
        last_level = -1

        while queue:
            node, level = queue.popleft()
            if level != last_level:
```

```
                right_view_result.append(node.val)
                last_level = level

            if node.right:
                queue.append((node.right, level + 1))
            if node.left:
                queue.append((node.left, level + 1))

        return right_view_result

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
    tree.insert(5)
    tree.insert(6)
    tree.insert(7)

    print("Right View of the binary tree:", tree.right_view())
    # Output: [1, 3, 7]
```

Symmetric Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
```

```python
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def is_symmetric(self):
        if not self.root:
            return True
        return self._is_mirror(self.root.left, self.root.right)

    def _is_mirror(self, left, right):
        if left is None and right is None:
            return True
        if left is None or right is None:
            return False
        return (left.val == right.val) and self._is_mirror(left.left,
right.right) and self._is_mirror(left.right, right.left)

if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.root.left = TreeNode(2)
    tree.root.right = TreeNode(2)
    tree.root.left.left = TreeNode(3)
    tree.root.left.right = TreeNode(4)
    tree.root.right.left = TreeNode(4)
    tree.root.right.right = TreeNode(3)

    print("Is the binary tree symmetric?:", tree.is_symmetric())
    # Output: True
```

Root to Node Path in Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
```

```
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def find_path(self, target):
        path = []
        if self._find_path_recursive(self.root, target, path):
            return path
        else:
            return None

    def _find_path_recursive(self, node, target, path):
        if not node:
            return False

        path.append(node.val)

        if node.val == target:
            return True

        if ((node.left and self._find_path_recursive(node.left, target, path))
or
            (node.right and self._find_path_recursive(node.right, target,
path))):
            return True

        path.pop()
        return False

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
```

```python
    tree.insert(5)
    tree.insert(6)
    tree.insert(7)

    target = 5
    path = tree.find_path(target)

    if path:
        print(f"Path from root to node {target}:", path)
    else:
        print(f"Node {target} not found in the tree.")
    # Output: Path from root to node 5: [1, 2, 5]
```

LCA in Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def find_lca(self, root, p, q):
        # Base case
        if not root:
            return None
```

```python
        # If either p or q is the root, then root is LCA
        if root.val == p or root.val == q:
            return root

        # Recur for left and right subtrees
        left_lca = self.find_lca(root.left, p, q)
        right_lca = self.find_lca(root.right, p, q)

        # If both left and right calls return non-None, root is LCA
        if left_lca and right_lca:
            return root

        # Otherwise, return non-None child
        return left_lca if left_lca else right_lca

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(20)
    tree.insert(8)
    tree.insert(22)
    tree.insert(4)
    tree.insert(12)
    tree.insert(10)
    tree.insert(14)

    lca = tree.find_lca(tree.root, 10, 14)
    if lca:
        print("LCA of 10 and 14 is:", lca.val)
    else:
        print("LCA not found.")
    # Output: LCA of 10 and 14 is: 12

    lca = tree.find_lca(tree.root, 14, 8)
    if lca:
        print("LCA of 14 and 8 is:", lca.val)
    else:
        print("LCA not found.")
    # Output: LCA of 14 and 8 is: 8

    lca = tree.find_lca(tree.root, 10, 22)
    if lca:
        print("LCA of 10 and 22 is:", lca.val)
    else:
        print("LCA not found.")
    # Output: LCA of 10 and 22 is: 20
```

Maximum width of a Binary Tree

```python
from collections import deque

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

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

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

        while queue:
            level_length = len(queue)
            _, first_index = queue[0]
            for i in range(level_length):
                node, index = queue.popleft()
                if node.left:
                    queue.append((node.left, 2 * index + 1))
                if node.right:
```

```python
                    queue.append((node.right, 2 * index + 2))
            _, last_index = queue[-1] if queue else (None, 0)
            max_width = max(max_width, last_index - first_index + 1)

        return max_width

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(1)
    tree.insert(3)
    tree.insert(2)
    tree.root.left.left = TreeNode(5)
    tree.root.left.right = TreeNode(3)
    tree.root.right.right = TreeNode(9)

    print("Maximum width of the binary tree:", tree.max_width())
    # Output: Maximum width of the binary tree: 4
```

Check for Children Sum Property

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
```

```python
                self._insert_recursive(node.right, value)

    def check_children_sum_property(self, node=None):
        if node is None:
            node = self.root

        # Base case: if the node is None or a leaf node
        if node is None or (node.left is None and node.right is None):
            return True

        # Initialize sum of children to 0
        left_val = 0
        right_val = 0

        # Get the value of left child
        if node.left:
            left_val = node.left.val

        # Get the value of right child
        if node.right:
            right_val = node.right.val

        # Check if the current node value is equal to the sum of its
children's values
        if node.val == left_val + right_val:
            # Recursively check for left and right subtrees
            return (self.check_children_sum_property(node.left) and
                    self.check_children_sum_property(node.right))

        return False

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(10)
    tree.root.left = TreeNode(8)
    tree.root.right = TreeNode(2)
    tree.root.left.left = TreeNode(3)
    tree.root.left.right = TreeNode(5)
    tree.root.right.right = TreeNode(2)

    print("Does the tree satisfy the Children Sum Property?:",
tree.check_children_sum_property())
    # Output: Does the tree satisfy the Children Sum Property?: True
```

Print all the Nodes at a distance of K in a Binary Tree

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def print_k_distance_nodes(self, target, k):
        def subtree_nodes_at_distance_k(node, k):
            if node is None or k < 0:
                return
            if k == 0:
                result.append(node.val)
                return
            subtree_nodes_at_distance_k(node.left, k - 1)
            subtree_nodes_at_distance_k(node.right, k - 1)

        def find_nodes(node, target, k):
            if node is None:
                return -1

            if node.val == target:
                subtree_nodes_at_distance_k(node, k)
                return 0

            left_dist = find_nodes(node.left, target, k)
            if left_dist != -1:
```

```
                if left_dist + 1 == k:
                    result.append(node.val)
                else:
                    subtree_nodes_at_distance_k(node.right, k - left_dist - 2)
                return left_dist + 1

            right_dist = find_nodes(node.right, target, k)
            if right_dist != -1:
                if right_dist + 1 == k:
                    result.append(node.val)
                else:
                    subtree_nodes_at_distance_k(node.left, k - right_dist - 2)
                return right_dist + 1

            return -1

        result = []
        find_nodes(self.root, target, k)
        return result

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(20)
    tree.insert(8)
    tree.insert(22)
    tree.root.left.left = TreeNode(4)
    tree.root.left.right = TreeNode(12)
    tree.root.left.right.left = TreeNode(10)
    tree.root.left.right.right = TreeNode(14)

    target = 8
    k = 2
    result = tree.print_k_distance_nodes(target, k)
    print(f"Nodes at distance {k} from node {target}:", result)
    # Output: Nodes at distance 2 from node 8: [10, 14, 22]
```

Minimum time taken to BURN the Binary Tree from a Node

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

class BinaryTree:
    def __init__(self):
```

```python
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def create_parent_map(self, node, parent_map, parent=None):
        if node:
            parent_map[node] = parent
            self.create_parent_map(node.left, parent_map, node)
            self.create_parent_map(node.right, parent_map, node)

    def burn_tree(self, target):
        if not self.root:
            return 0

        # Step 1: Create a parent map using DFS
        parent_map = {}
        self.create_parent_map(self.root, parent_map)

        # Step 2: Use BFS to simulate the burning process
        from collections import deque

        # Find the target node
        target_node = None
        def find_target(node, target):
            if not node:
                return None
            if node.val == target:
                return node
            left_result = find_target(node.left, target)
            if left_result:
                return left_result
            return find_target(node.right, target)
```

```python
        target_node = find_target(self.root, target)
        if not target_node:
            return -1  # Target node not found

        # Initialize BFS
        queue = deque([target_node])
        visited = set([target_node])
        time = 0

        while queue:
            size = len(queue)
            for _ in range(size):
                current = queue.popleft()
                # Visit left child
                if current.left and current.left not in visited:
                    visited.add(current.left)
                    queue.append(current.left)
                # Visit right child
                if current.right and current.right not in visited:
                    visited.add(current.right)
                    queue.append(current.right)
                # Visit parent
                parent = parent_map.get(current)
                if parent and parent not in visited:
                    visited.add(parent)
                    queue.append(parent)
            if queue:
                time += 1

        return time

# Example usage:
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(10)
    tree.insert(20)
    tree.insert(30)
    tree.insert(40)
    tree.insert(50)
    tree.root.left = TreeNode(15)
    tree.root.right = TreeNode(25)
    tree.root.left.left = TreeNode(12)
    tree.root.left.right = TreeNode(18)
    tree.root.right.right = TreeNode(35)

    target = 15
```

```
    print(f"Minimum time to burn the tree from node {target}:
{tree.burn_tree(target)}")
    # Output: Minimum time to burn the tree from node 15: 3
```

Count total Nodes in a COMPLETE Binary Tree

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

class CompleteBinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.val:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:  # value >= node.val
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def count_nodes(self):
        def get_tree_height(node):
            height = 0
            while node:
                height += 1
                node = node.left
            return height

        def count_nodes_recursive(node):
            if not node:
                return 0

            left_height = get_tree_height(node.left)
```

```
            right_height = get_tree_height(node.right)

            if left_height == right_height:
                return (1 << left_height) + count_nodes_recursive(node.right)
            else:
                return (1 << right_height) + count_nodes_recursive(node.left)

        return count_nodes_recursive(self.root)

# Example usage:
if __name__ == "__main__":
    tree = CompleteBinaryTree()
    tree.insert(10)
    tree.insert(5)
    tree.insert(15)
    tree.insert(2)
    tree.insert(7)
    tree.insert(12)
    tree.insert(18)
    tree.insert(1)
    tree.insert(3)

    print("Total number of nodes in the complete binary tree:",
tree.count_nodes())
    # Output: Total number of nodes in the complete binary tree: 9
```

Construct Binary Tree from inorder and preorder

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

def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None

    root_val = preorder.pop(0)  # First element in preorder is the root
    root = TreeNode(root_val)

    # Find index of root in inorder traversal
    inorder_index = inorder.index(root_val)

    # Build left subtree
    root.left = build_tree(preorder, inorder[:inorder_index])
```

```python
    # Build right subtree
    root.right = build_tree(preorder, inorder[inorder_index + 1:])

    return root

def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.val, end=' ')
        inorder_traversal(node.right)

def preorder_traversal(node):
    if node:
        print(node.val, end=' ')
        preorder_traversal(node.left)
        preorder_traversal(node.right)

# Example usage:
if __name__ == "__main__":
    # Example inputs
    inorder = [9, 3, 15, 20, 7]
    preorder = [3, 9, 20, 15, 7]

    # Build the tree
    root = build_tree(preorder, inorder)

    # Output the inorder and preorder traversals of the constructed tree
    print("Inorder traversal:")
    inorder_traversal(root)
    print("\nPreorder traversal:")
    preorder_traversal(root)
```

Construct the Binary Tree from Postorder and Inorder Traversal

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

def build_tree(postorder, inorder):
    if not postorder or not inorder:
        return None

    root_val = postorder.pop()  # Last element in postorder is the root
    root = TreeNode(root_val)
```

```python
    # Find index of root in inorder traversal
    inorder_index = inorder.index(root_val)

    # Build right subtree
    root.right = build_tree(postorder, inorder[inorder_index + 1:])

    # Build left subtree
    root.left = build_tree(postorder, inorder[:inorder_index])

    return root

def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.val, end=' ')
        inorder_traversal(node.right)

def postorder_traversal(node):
    if node:
        postorder_traversal(node.left)
        postorder_traversal(node.right)
        print(node.val, end=' ')

# Example usage:
if __name__ == "__main__":
    # Example inputs
    inorder = [9, 3, 15, 20, 7]
    postorder = [9, 15, 7, 20, 3]

    # Build the tree
    root = build_tree(postorder, inorder)

    # Output the inorder and postorder traversals of the constructed tree
    print("Inorder traversal:")
    inorder_traversal(root)
    print("\nPostorder traversal:")
    postorder_traversal(root)
```

Serialize and deserialize Binary Tree

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

```python
def serialize(root):
    def serialize_helper(node):
        if not node:
            return '# '
        serialized_string = str(node.val) + ' '
        serialized_string += serialize_helper(node.left)
        serialized_string += serialize_helper(node.right)
        return serialized_string

    if not root:
        return ''

    return serialize_helper(root)

def deserialize(data):
    def deserialize_helper(tokens):
        if tokens[0] == '#':
            tokens.pop(0)  # Consume the '#'
            return None

        # Create current node
        root = TreeNode(int(tokens.pop(0)))
        root.left = deserialize_helper(tokens)
        root.right = deserialize_helper(tokens)
        return root

    if not data:
        return None

    tokens = data.split()
    return deserialize_helper(tokens)

# Example usage:
if __name__ == "__main__":
    # Example binary tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.right.left = TreeNode(4)
    root.right.right = TreeNode(5)

    # Serialize the binary tree
    serialized_tree = serialize(root)
    print("Serialized tree:", serialized_tree)

    # Deserialize the serialized tree
    deserialized_tree = deserialize(serialized_tree)
```

```
    # Test deserialized tree by printing inorder traversal
    def inorder_traversal(node):
        if node:
            inorder_traversal(node.left)
            print(node.val, end=' ')
            inorder_traversal(node.right)

    print("Inorder traversal of deserialized tree:")
    inorder_traversal(deserialized_tree)
```

Morris Preorder Traversal of a Binary Tree

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

def morris_preorder(root):
    current = root
    while current:
        if current.left is None:
            print(current.val, end=' ')
            current = current.right
        else:
            # Find the predecessor (rightmost node in the left subtree)
            predecessor = current.left
            while predecessor.right and predecessor.right != current:
                predecessor = predecessor.right

            if predecessor.right is None:
                # Make the current node as the right child of predecessor
                predecessor.right = current
                print(current.val, end=' ')  # Visit before going left
                current = current.left
            else:
                # Restore the tree structure
                predecessor.right = None
                current = current.right

# Example usage:
if __name__ == "__main__":
    # Example binary tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
```

```python
    root.left.right = TreeNode(5)

    print("Morris preorder traversal:")
    morris_preorder(root)
```

Morris Inorder Traversal of a Binary Tree

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

def morris_inorder(root):
    current = root
    while current:
        if current.left is None:
            print(current.val, end=' ')
            current = current.right
        else:
            # Find the predecessor (rightmost node in the left subtree)
            predecessor = current.left
            while predecessor.right and predecessor.right != current:
                predecessor = predecessor.right

            if predecessor.right is None:
                # Make the current node as the right child of predecessor
                predecessor.right = current
                current = current.left
            else:
                # Restore the tree structure
                predecessor.right = None
                print(current.val, end=' ')
                current = current.right

# Example usage:
if __name__ == "__main__":
    # Example binary tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)

    print("Morris inorder traversal:")
    morris_inorder(root)
```

Flatten Binary Tree to LinkedList

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

def flatten(root):
    if not root:
        return

    # Flatten the left and right subtrees recursively
    flatten(root.left)
    flatten(root.right)

    # Store the flattened right subtree
    right_subtree = root.right

    # Move the entire left subtree to the right subtree
    root.right = root.left
    root.left = None

    # Traverse to the end of the new right subtree and attach the original
right subtree
    current = root
    while current.right:
        current = current.right
    current.right = right_subtree

def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=' ')
        current = current.right
    print()

# Example usage:
if __name__ == "__main__":
    # Example 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 the binary tree
    flatten(root)
```

```python
# Print the flattened linked list
print("Flattened linked list (preorder):")
print_linked_list(root)
```

Binary Search Tree

Search in a Binary Search Tree

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        # Base case: node is null or key is present at node
        if node is None or node.key == key:
            return node

        # Key is greater than node's key
        if key > node.key:
```

```python
            return self._search(node.right, key)

        # Key is smaller than node's key
        return self._search(node.left, key)

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

# Search for a key
key_to_search = 40
result = bst.search(key_to_search)

if result:
    print(f"Key {key_to_search} found in the BST.")
else:
    print(f"Key {key_to_search} not found in the BST.")
```

Find Min/Max in BST

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
```

```python
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None or node.key == key:
            return node
        if key > node.key:
            return self._search(node.right, key)
        return self._search(node.left, key)

    def find_min(self):
        if self.root is None:
            return None
        return self._find_min(self.root)

    def _find_min(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def find_max(self):
        if self.root is None:
            return None
        return self._find_max(self.root)

    def _find_max(self, node):
        current = node
        while current.right is not None:
            current = current.right
        return current

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)
```

```python
# Find the minimum value
min_node = bst.find_min()
if min_node:
    print(f"The minimum value in the BST is {min_node.key}.")
else:
    print("The BST is empty.")

# Find the maximum value
max_node = bst.find_max()
if max_node:
    print(f"The maximum value in the BST is {max_node.key}.")
else:
    print("The BST is empty.")
```

Ceil in a Binary Search Tree

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)
```

```python
    def _search(self, node, key):
        if node is None or node.key == key:
            return node
        if key > node.key:
            return self._search(node.right, key)
        return self._search(node.left, key)

    def find_ceil(self, key):
        return self._find_ceil(self.root, key)

    def _find_ceil(self, node, key):
        if node is None:
            return None

        # If node's key is equal to the key
        if node.key == key:
            return node

        # If node's key is smaller, ceil must be in the right subtree
        if node.key < key:
            return self._find_ceil(node.right, key)

        # If node's key is greater, ceil could be in the left subtree or it
could be the   current node
        left_ceil = self._find_ceil(node.left, key)
        return left_ceil if left_ceil else node

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

# Find the ceil value
key_to_search = 65
ceil_node = bst.find_ceil(key_to_search)

if ceil_node:
    print(f"The ceil value in the BST for {key_to_search} is
{ceil_node.key}.")
else:
    print(f"There is no ceil value in the BST for {key_to_search}.")
```

Floor in a Binary Search Tree

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None or node.key == key:
            return node
        if key > node.key:
            return self._search(node.right, key)
        return self._search(node.left, key)

    def find_floor(self, key):
        return self._find_floor(self.root, key)

    def _find_floor(self, node, key):
        if node is None:
            return None

        # If node's key is equal to the key
        if node.key == key:
```

```
            return node

        # If node's key is greater, floor must be in the left subtree
        if node.key > key:
            return self._find_floor(node.left, key)

        # If node's key is smaller, floor could be in the right subtree or
it could be the current node
        right_floor = self._find_floor(node.right, key)
        return right_floor if right_floor else node

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

# Find the floor value
key_to_search = 65
floor_node = bst.find_floor(key_to_search)

if floor_node:
    print(f"The floor value in the BST for {key_to_search} is
{floor_node.key}.")
else:
    print(f"There is no floor value in the BST for {key_to_search}.")
```

Insert a given Node in Binary Search Tree

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
```

```python
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def inorder(self, node, res):
        if node:
            self.inorder(node.left, res)
            res.append(node.key)
            self.inorder(node.right, res)

    def display(self):
        res = []
        self.inorder(self.root, res)
        return res

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

# Display the BST in inorder traversal (should be sorted)
print("Inorder traversal of the BST:", bst.display())
```

Delete a Node in Binary Search Tree

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

class BinarySearchTree:
```

```python
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete(self, node, key):
        if node is None:
            return node

        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            # Node with only one child or no child
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left

            # Node with two children: Get the inorder successor
            temp = self._min_value_node(node.right)
            node.key = temp.key
            node.right = self._delete(node.right, temp.key)

        return node

    def _min_value_node(self, node):
        current = node
```

```
        while current.left is not None:
            current = current.left
        return current

    def inorder(self, node, res):
        if node:
            self.inorder(node.left, res)
            res.append(node.key)
            self.inorder(node.right, res)

    def display(self):
        res = []
        self.inorder(self.root, res)
        return res

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

print("Inorder traversal before deletion:", bst.display())

# Delete a node
key_to_delete = 50
bst.delete(key_to_delete)

print("Inorder traversal after deletion:", bst.display())
```

Find K-th smallest/largest element in BST

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
```

```
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def kth_smallest(self, k):
        self.k = k
        self.count = 0
        self.result = None
        self._inorder(self.root)
        return self.result

    def _inorder(self, node):
        if node is None or self.result is not None:
            return
        self._inorder(node.left)
        self.count += 1
        if self.count == self.k:
            self.result = node.key
            return
        self._inorder(node.right)

    def kth_largest(self, k):
        self.k = k
        self.count = 0
        self.result = None
        self._reverse_inorder(self.root)
        return self.result

    def _reverse_inorder(self, node):
        if node is None or self.result is not None:
            return
        self._reverse_inorder(node.right)
        self.count += 1
        if self.count == self.k:
            self.result = node.key
            return
```

```python
        self._reverse_inorder(node.left)

# Example usage:
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

k = 3
kth_smallest = bst.kth_smallest(k)
kth_largest = bst.kth_largest(k)

print(f"The {k}-th smallest element in the BST is {kth_smallest}.")
print(f"The {k}-th largest element in the BST is {kth_largest}.")
```

Check if a tree is a BST or BT

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        # For a general binary tree, we can insert arbitrarily.
        # For simplicity, we'll use level order insertion here.
        queue = [node]
        while queue:
            current = queue.pop(0)
            if not current.left:
                current.left = TreeNode(key)
                break
            else:
```

```python
                queue.append(current.left)
            if not current.right:
                current.right = TreeNode(key)
                break
            else:
                queue.append(current.right)

def is_bst(node, min_key=float('-inf'), max_key=float('inf')):
    if node is None:
        return True
    if not (min_key < node.key < max_key):
        return False
    return (is_bst(node.left, min_key, node.key) and
            is_bst(node.right, node.key, max_key))

def is_binary_tree(node):
    # Any tree with nodes is a binary tree by definition
    return node is not None

# Example usage:
bt = BinaryTree()
bt.insert(10)
bt.insert(5)
bt.insert(20)
bt.insert(3)
bt.insert(7)
bt.insert(15)
bt.insert(25)

print("Is the tree a binary tree?", is_binary_tree(bt.root))  # Should be
True for any non-empty tree
print("Is the tree a binary search tree?", is_bst(bt.root))  # Depends on
the structure and values
```

LCA in Binary Search Tree

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
```

```
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def find_lca(self, node, p, q):
        # If the BST is empty, return None
        if node is None:
            return None

        # If both p and q are smaller than the node's key, LCA lies in the
left subtree
        if p < node.key and q < node.key:
            return self.find_lca(node.left, p, q)

        # If both p and q are greater than the node's key, LCA lies in the
right subtree
        if p > node.key and q > node.key:
            return self.find_lca(node.right, p, q)

        # If we reach here, then node is the LCA of p and q
        return node

# Example usage:
bst = BinarySearchTree()
bst.insert(20)
bst.insert(8)
bst.insert(22)
bst.insert(4)
bst.insert(12)
bst.insert(10)
bst.insert(14)

p = 10
q = 14
lca = bst.find_lca(bst.root, p, q)
```

```python
  if lca:
      print(f"The LCA of {p} and {q} is {lca.key}.")
  else:
      print(f"There is no common ancestor for {p} and {q} in the BST.")
```

Construct a BST from a preorder traversal

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

   class BinarySearchTree:
       def __init__(self):
           self.root = None

       def construct_bst_from_preorder(self, preorder):
           if not preorder:
               return None
           self.index = 0
           return self._construct_bst_from_preorder(preorder, float('-inf'),
float('inf'))

       def _construct_bst_from_preorder(self, preorder, min_val, max_val):
           if self.index >= len(preorder):
               return None

           key = preorder[self.index]

           if key < min_val or key > max_val:
               return None

           # Construct the current node and move to the next element
           node = TreeNode(key)
           self.index += 1

           # All elements in the left subtree must be smaller than the current
node's key
           node.left = self._construct_bst_from_preorder(preorder, min_val,
key)

           # All elements in the right subtree must be greater than the
current node's key
           node.right = self._construct_bst_from_preorder(preorder, key,
max_val)
```

```python
        return node

    def inorder(self, node, res):
        if node:
            self.inorder(node.left, res)
            res.append(node.key)
            self.inorder(node.right, res)

    def display(self):
        res = []
        self.inorder(self.root, res)
        return res

# Example usage:
bst = BinarySearchTree()
preorder = [10, 5, 1, 7, 40, 50]
bst.root = bst.construct_bst_from_preorder(preorder)

# Display the BST in inorder traversal (should be sorted)
print("Inorder traversal of the constructed BST:", bst.display())
```

Inorder Successor/Predecessor in BST

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
```

```
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def find_min(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def find_max(self, node):
        current = node
        while current.right is not None:
            current = current.right
        return current

    def find_inorder_successor(self, root, key):
        successor = None
        current = root
        while current:
            if key < current.key:
                successor = current
                current = current.left
            elif key > current.key:
                current = current.right
            else:
                if current.right:
                    successor = self.find_min(current.right)
                break
        return successor

    def find_inorder_predecessor(self, root, key):
        predecessor = None
        current = root
        while current:
            if key < current.key:
                current = current.left
            elif key > current.key:
                predecessor = current
                current = current.right
            else:
                if current.left:
                    predecessor = self.find_max(current.left)
                break
        return predecessor

# Example usage:
bst = BinarySearchTree()
```

```python
keys = [20, 8, 22, 4, 12, 10, 14]
for key in keys:
    bst.insert(key)

node_key = 10
successor = bst.find_inorder_successor(bst.root, node_key)
predecessor = bst.find_inorder_predecessor(bst.root, node_key)

if successor:
    print(f"The inorder successor of {node_key} is {successor.key}.")
else:
    print(f"There is no inorder successor of {node_key} in the BST.")

if predecessor:
    print(f"The inorder predecessor of {node_key} is {predecessor.key}.")
else:
    print(f"There is no inorder predecessor of {node_key} in the BST.")
```

Merge 2 BST's

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)
```

```python
    def inorder(self, node, res):
        if node:
            self.inorder(node.left, res)
            res.append(node.key)
            self.inorder(node.right, res)

    def to_sorted_list(self):
        res = []
        self.inorder(self.root, res)
        return res

def merge_sorted_lists(list1, list2):
    merged = []
    i, j = 0, 0

    while i < list1.length and j < list2.length:
        if list1[i] < list2[j]:
            merged.append(list1[i])
            i += 1
        else:
            merged.append(list2[j])
            j += 1

    while i < list1.length:
        merged.append(list1[i])
        i += 1

    while j < list2.length:
        merged.append(list2[j])
        j += 1

    return merged

def sorted_list_to_bst(sorted_list):
    if not sorted_list:
        return None

    mid = len(sorted_list) // 2
    node = TreeNode(sorted_list[mid])

    node.left = sorted_list_to_bst(sorted_list[:mid])
    node.right = sorted_list_to_bst(sorted_list[mid+1:])

    return node

def merge_bsts(bst1, bst2):
    list1 = bst1.to_sorted_list()
```

```
    list2 = bst2.to_sorted_list()
    merged_list = merge_sorted_lists(list1, list2)
    return sorted_list_to_bst(merged_list)

# Example usage:
bst1 = BinarySearchTree()
bst2 = BinarySearchTree()

keys1 = [20, 8, 22, 4, 12, 10, 14]
keys2 = [25, 18, 30, 15, 19, 27, 35]

for key in keys1:
    bst1.insert(key)

for key in keys2:
    bst2.insert(key)

merged_bst_root = merge_bsts(bst1, bst2)

# Helper function to print inorder traversal of the BST
def print_inorder(node):
    if node:
        print_inorder(node.left)
        print(node.key, end=' ')
        print_inorder(node.right)

print("Inorder traversal of the merged BST:")
print_inorder(merged_bst_root)
```

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

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)
```

```python
    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def find_pair_with_sum(self, root, target):
        # In-order and reverse in-order iterators
        def inorder(node):
            stack = []
            while stack or node:
                while node:
                    stack.append(node)
                    node = node.left
                node = stack.pop()
                yield node
                node = node.right

        def reverse_inorder(node):
            stack = []
            while stack or node:
                while node:
                    stack.append(node)
                    node = node.right
                node = stack.pop()
                yield node
                node = node.left

        if not root:
            return False

        it1 = inorder(root)
        it2 = reverse_inorder(root)

        left = next(it1, None)
        right = next(it2, None)

        while left is not None and right is not None and left != right:
            current_sum = left.key + right.key
            if current_sum == target:
                print(f"Pair found: ({left.key}, {right.key})")
                return True
```

```
            elif current_sum < target:
                left = next(it1, None)
            else:
                right = next(it2, None)

        print("No pair found")
        return False

# Example usage:
bst = BinarySearchTree()
keys = [20, 8, 22, 4, 12, 10, 14]
for key in keys:
    bst.insert(key)

target_sum = 18
if not bst.find_pair_with_sum(bst.root, target_sum):
    print(f"No pair with the sum {target_sum} exists in the BST.")
```

Recover BST | Correct BST with two nodes swapped

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

class Solution:
    def __init__(self):
        self.first = None
          self.second = None
          self.prev = TreeNode(float('-inf'))

      def inorder(self, root):
          if not root:
              return

          # Traverse the left subtree
          self.inorder(root.left)

          # Find nodes that are out of order
          if not self.first and self.prev.key > root.key:
              self.first = self.prev
          if self.first and self.prev.key > root.key:
              self.second = root

          # Update previous node
          self.prev = root
```

```
        # Traverse the right subtree
        self.inorder(root.right)

    def recoverTree(self, root):
        # Perform in-order traversal and find the two nodes
        self.inorder(root)

        # Swap the values of the two nodes
        if self.first and self.second:
            self.first.key, self.second.key = self.second.key,
self.first.key

# Helper function to insert nodes into the BST
def insert(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

# Helper function to do in-order traversal and print the tree
def inorder_print(root):
    if root:
        inorder_print(root.left)
        print(root.key, end=' ')
        inorder_print(root.right)

# Example usage:
root = None
keys = [10, 5, 15, 3, 7, 12, 17]
for key in keys:
    root = insert(root, key)

# Swap two nodes manually to simulate the problem
root.left.key, root.right.key = root.right.key, root.left.key

print("Inorder traversal of the tree before recovery:")
inorder_print(root)
print()

# Recover the BST
solution = Solution()
solution.recoverTree(root)

print("Inorder traversal of the tree after recovery:")
```

```
  inorder_print(root)
  print()
```

Largest BST in Binary Tree

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

class BSTInfo:
    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_in_bt(root):
    def postorder(node):
        nonlocal max_bst_size

        if not node:
            # An empty tree is a BST of size 0
            return BSTInfo(True, 0, float('inf'), float('-inf'))

        # Recursively get info from left and right subtrees
        left_info = postorder(node.left)
        right_info = postorder(node.right)

        # Check if the current node makes a valid BST
        if (left_info.is_bst and right_info.is_bst and
            left_info.max_val < node.key < right_info.min_val):
            size = left_info.size + right_info.size + 1
            min_val = min(left_info.min_val, node.key)
            max_val = max(right_info.max_val, node.key)
            max_bst_size = max(max_bst_size, size)
            return BSTInfo(True, size, min_val, max_val)
        else:
            # If it's not a BST, return the size of the largest BST found so
far
            return BSTInfo(False, max(left_info.size, right_info.size), 0,
0)

    max_bst_size = 0
    postorder(root)
    return max_bst_size
```

```python
# Helper function to insert nodes into the Binary Tree
def insert(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

# Example usage:
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)

print("Size of the largest BST in the given Binary Tree:",
largest_bst_in_bt(root))
```

Graphs

BFS

```python
from collections import deque

def bfs(graph, start):
    visited = set()  # To keep track of visited nodes
    queue = deque([start])  # Initialize a queue with the start node

    while queue:
        # Dequeue a node from the front of the queue
        node = queue.popleft()

        if node not in visited:
            # Mark the node as visited
            visited.add(node)
            print(node)  # Process the node (you can replace this with any
action)

            # Enqueue all adjacent nodes that haven't been visited
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
```

```
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

bfs(graph, 'A')
```

DFS

```
def dfs_iterative(graph, start):
    visited = set()  # To keep track of visited nodes
    stack = [start]  # Initialize a stack with the start node

    while stack:
        # Pop a node from the top of the stack
        node = stack.pop()

        if node not in visited:
            # Mark the node as visited
            visited.add(node)
            print(node)  # Process the node (you can replace this with any
action)

            # Push all adjacent nodes that haven't been visited onto the
stack
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

dfs_iterative(graph, 'A')
```

Number of provinces

```
def findCircleNum(isConnected):
    def dfs(node):
        for neighbor, connected in enumerate(isConnected[node]):
            if connected and neighbor not in visited:
                visited.add(neighbor)
                dfs(neighbor)

    visited = set()
    provinces = 0

    for i in range(len(isConnected)):
        if i not in visited:
            dfs(i)
            provinces += 1

    return provinces

# Example usage
isConnected = [
    [1, 1, 0],
    [1, 1, 0],
    [0, 0, 1]
]

print(findCircleNum(isConnected))  # Output: 2
```

Connected Components Problem in Matrix

```
def count_connected_components(matrix):
    if not matrix or not matrix[0]:
        return 0

    rows, cols = len(matrix), len(matrix[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]

    def dfs(r, c, value):
      if r < 0 or c < 0 or r >= rows or c >= cols or visited[r][c] or
matrix[r][c] !=   value:
            return
        visited[r][c] = True
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        for dr, dc in directions:
            dfs(r + dr, c + dc, value)

    component_count = 0
```

```python
    for r in range(rows):
        for c in range(cols):
            if not visited[r][c]:
                dfs(r, c, matrix[r][c])
                component_count += 1

    return component_count

# Example usage
matrix = [
    [1, 1, 0, 0, 0],
    [1, 0, 0, 1, 1],
    [0, 0, 1, 1, 0],
    [0, 0, 0, 0, 1]
]

print(count_connected_components(matrix))  # Output: 5
```

Rotten Oranges

```python
from collections import deque

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

    # Initialize the queue with all rotten oranges
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1

    # If there are no fresh oranges, return 0 immediately
    if fresh_count == 0:
        return 0

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

    # BFS to rot the fresh oranges
    while queue:
        minutes_passed += 1
        for _ in range(len(queue)):
```

```
            x, y = queue.popleft()
            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
                    fresh_count -= 1
                    queue.append((nx, ny))

    # If there are still fresh oranges left, return -1
    return minutes_passed - 1 if fresh_count == 0 else -1

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

print(orangesRotting(grid))  # Output: 4
```

Flood fill

```
def floodFill(image, sr, sc, newColor):
    if image[sr][sc] == newColor:
        return image

    rows, cols = len(image), len(image[0])
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    original_color = image[sr][sc]

    def dfs(r, c):
        if image[r][c] == original_color:
            image[r][c] = newColor
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and image[nr][nc] ==
original_color:
                    dfs(nr, nc)

    dfs(sr, sc)
    return image

# Example usage
image = [
    [1, 1, 1],
    [1, 1, 0],
    [1, 0, 1]
```

```
]
sr, sc = 1, 1
newColor = 2

print(floodFill(image, sr, sc, newColor))
```

Cycle Detection in unirected Graph (bfs)

```
from collections import defaultdict, deque

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

    visited = [-1] * n  # -1: unvisited, 0: in queue, 1: visited
    parent = [-1] * n

    for i in range(n):
        if visited[i] == -1:  # not visited
            if bfs_cycle_check(graph, i, visited, parent):
                return True
    return False

def bfs_cycle_check(graph, start, visited, parent):
    queue = deque([start])
    visited[start] = 0  # mark as in queue

    while queue:
        node = queue.popleft()
        visited[node] = 1  # mark as visited

        for neighbor in graph[node]:
            if visited[neighbor] == -1:  # not visited
                queue.append(neighbor)
                visited[neighbor] = 0  # mark as in queue
                parent[neighbor] = node
            elif visited[neighbor] == 0 and parent[node] != neighbor:
                return True  # Cycle detected
    return False

# Example usage
edges = [(0, 1), (1, 2), (2, 0)]  # Example graph with a cycle

print(hasCycle(3, edges))  # Output: True
```

Cycle Detection in undirected Graph (dfs)

```
from collections import defaultdict

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

    visited = [-1] * n  # -1: unvisited, 0: in recursion stack, 1: visited

    for i in range(n):
        if visited[i] == -1:  # not visited
            if dfs_cycle_check(graph, i, visited, -1):  # -1 as parent of starting node
                return True

    return False

def dfs_cycle_check(graph, node, visited, parent):
    visited[node] = 0  # mark as in recursion stack

    for neighbor in graph[node]:
        if visited[neighbor] == -1:  # not visited
            if dfs_cycle_check(graph, neighbor, visited, node):
                return True
        elif visited[neighbor] == 0 and neighbor != parent:
            return True  # Cycle detected

    visited[node] = 1  # mark as visited
    return False

# Example usage
edges = [(0, 1), (1, 2), (2, 0)]  # Example graph with a cycle

print(hasCycle(3, edges))  # Output: True
```

0/1 Matrix (Bfs Problem)

```
from collections import deque

def updateMatrix(matrix):
    if not matrix:
        return matrix

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

```python
    queue = deque()

    # Enqueue all cells with value 0 and mark others with -1
    for r in range(rows):
        for c in range(cols):
            if matrix[r][c] == 0:
                queue.append((r, c))
            else:
                matrix[r][c] = -1

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

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

        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and matrix[nr][nc] == -1:
                matrix[nr][nc] = matrix[r][c] + 1
                queue.append((nr, nc))

    return matrix

# Example usage
matrix = [
    [0, 0, 0],
    [0, 1, 0],
    [1, 1, 1]
]

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

Surrounded Regions (dfs)

```python
def solve(board):
    if not board or not board[0]:
        return

    rows, cols = len(board), len(board[0])

    def dfs(r, c):
        if r < 0 or r >= rows or c < 0 or c >= cols or board[r][c] != 'O':
            return
        board[r][c] = '#'
```

```python
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            dfs(nr, nc)

    # Step 1: Traverse the border and mark all connected 'O's as visited
    for r in range(rows):
        for c in [0, cols - 1]:
            if board[r][c] == 'O':
                dfs(r, c)

    for c in range(cols):
        for r in [0, rows - 1]:
            if board[r][c] == 'O':
                dfs(r, c)

    # Step 2: Mark surrounded 'O's as 'X' and restore '#' to 'O'
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == 'O':
                board[r][c] = 'X'
            elif board[r][c] == '#':
                board[r][c] = 'O'

# Example usage
board = [
    ['X', 'X', 'X', 'X'],
    ['X', 'O', 'O', 'X'],
    ['X', 'X', 'O', 'X'],
    ['X', 'O', 'X', 'X']
]

solve(board)
for row in board:
    print(row)
```

Number of Enclaves [flood fill implementation - multisource]

```python
def numEnclaves(grid):
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])

    def dfs(r, c):
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] != 1:
            return
```

```python
        grid[r][c] = 0  # Mark as visited

        # Check 4 neighbors
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    # Step 1: Perform DFS from boundary cells (first and last rows)
    for r in range(rows):
        if grid[r][0] == 1:
            dfs(r, 0)
        if grid[r][cols - 1] == 1:
            dfs(r, cols - 1)

    for c in range(cols):
        if grid[0][c] == 1:
            dfs(0, c)
        if grid[rows - 1][c] == 1:
            dfs(rows - 1, c)

    # Step 2: Count remaining '1's in the grid
    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                count += 1

    return count

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

print(numEnclaves(grid))  # Output: 1
```

Word ladder – 1

```python
from collections import deque, defaultdict

def findLadders(beginWord, endWord, wordList):
    # Step 1: Build adjacency list
    wordList = set(wordList)
```

```python
    if endWord not in wordList:
        return []

    adj_list = defaultdict(list)
    for word in wordList:
        for i in range(len(word)):
            wildcard = word[:i] + '*' + word[i+1:]
            adj_list[wildcard].append(word)

    # Step 2: BFS to find shortest path
    queue = deque([(beginWord, [beginWord])])
    visited = set()
    visited.add(beginWord)

    while queue:
        curr_word, path = queue.popleft()

        if curr_word == endWord:
            return path

        for i in range(len(curr_word)):
            wildcard = curr_word[:i] + '*' + curr_word[i+1:]
            for neighbor in adj_list[wildcard]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [neighbor]))
              # No need to remove from adj_list because we're using wildcard

    return []

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

print(findLadders(beginWord, endWord, wordList))
```

Word ladder – 2

```python
from collections import defaultdict, deque

def findLadders(beginWord, endWord, wordList):
    # Step 1: Build adjacency list using BFS
    wordList = set(wordList)
    if endWord not in wordList:
        return []
```

```python
    adj_list = defaultdict(list)
    level = {}
    queue = deque([beginWord])
    level[beginWord] = 0

    while queue:
        curr_word = queue.popleft()
        next_level = level[curr_word] + 1

        for i in range(len(curr_word)):
            for ch in 'abcdefghijklmnopqrstuvwxyz':
                if ch != curr_word[i]:
                    new_word = curr_word[:i] + ch + curr_word[i+1:]
                    if new_word in wordList:
                        adj_list[curr_word].append(new_word)
                        if new_word not in level:
                            level[new_word] = next_level
                            queue.append(new_word)

    # Step 2: DFS to find all paths
    res = []
    path = [beginWord]

    def dfs(word):
        if word == endWord:
            res.append(path[:])
            return

        if word not in adj_list:
            return

        for neighbor in adj_list[word]:
            if level[neighbor] == level[word] + 1:
                path.append(neighbor)
                dfs(neighbor)
                path.pop()

    dfs(beginWord)
    return res

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

print(findLadders(beginWord, endWord, wordList))
```

Number of Distinct Islands [dfs multisource]

```python
def numDistinctIslands(grid):
    if not grid or not grid[0]:
        return 0

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

    def dfs(r, c, shape, origin_r, origin_c):
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] != 1:
            return

        # Mark the cell as visited
        grid[r][c] = 0

        # Add the relative position to the shape
        shape.append((r - origin_r, c - origin_c))

        # Explore all 4 directions
        dfs(r + 1, c, shape, origin_r, origin_c)
        dfs(r - 1, c, shape, origin_r, origin_c)
        dfs(r, c + 1, shape, origin_r, origin_c)
        dfs(r, c - 1, shape, origin_r, origin_c)

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                shape = []
                dfs(r, c, shape, r, c)  # Start DFS from this cell
                normalized_shape = tuple(sorted(shape))  # Normalize shape

                # Add normalized shape to set of seen shapes
                seen_shapes.add(normalized_shape)

    return len(seen_shapes)

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

print(numDistinctIslands(grid))  # Output: 2
```

Bipartite Graph (DFS)

```python
def isBipartite(graph):
    def dfs(node, c):
        color[node] = c
        for neighbor in graph[node]:
            if color[neighbor] == c:
                return False
            if color[neighbor] == -1 and not dfs(neighbor, 1 - c):
                return False
        return True

    n = len(graph)
    color = [-1] * n  # -1 means uncolored, 0 and 1 are the two colors

    for i in range(n):
        if color[i] == -1:  # unvisited
            if not dfs(i, 0):
                return False

    return True

# Example usage
graph = [
    [1, 3],
    [0, 2],
    [1, 3],
    [0, 2]
]

print(isBipartite(graph))  # Output: True
```

Cycle Detection in Directed Graph (DFS)

```python
def isCyclicUtil(v, adj_list, visited, rec_stack):
    visited[v] = True
    rec_stack[v] = True

    for neighbor in adj_list[v]:
        if not visited[neighbor]:
            if isCyclicUtil(neighbor, adj_list, visited, rec_stack):
                return True
        elif rec_stack[neighbor]:
            return True

    rec_stack[v] = False
    return False
```

```python
def isCyclic(numCourses, prerequisites):
    adj_list = [[] for _ in range(numCourses)]

    for pair in prerequisites:
        adj_list[pair[1]].append(pair[0])

    visited = [False] * numCourses
    rec_stack = [False] * numCourses

    for v in range(numCourses):
        if not visited[v]:
            if isCyclicUtil(v, adj_list, visited, rec_stack):
                return True

    return False

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

print(isCyclic(numCourses, prerequisites))  # Output: True (cycle exists)
```

Topo Sort

```python
from collections import defaultdict, deque

def topoSortDFS(graph):
    def dfs(v):
        visited[v] = True
        for neighbor in graph[v]:
            if not visited[neighbor]:
                dfs(neighbor)
        stack.append(v)

    num_vertices = len(graph)
    visited = [False] * num_vertices
    stack = []

    for v in range(num_vertices):
        if not visited[v]:
            dfs(v)

    # The stack now contains vertices in topologically sorted order
    return stack[::-1]  # Return reversed stack to get the correct order

# Example usage
```

```python
graph = {
    0: [1, 2],
    1: [3],
    2: [3],
    3: []
}

print(topoSortDFS(graph))  # Output: [0, 2, 1, 3]
```

Kahn's Algorithm

```python
from collections import defaultdict, deque

def topoSortKahn(graph):
    num_vertices = len(graph)
    in_degree = [0] * num_vertices
    topo_order = []
    queue = deque()

    # Calculate in-degrees
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # Enqueue vertices with zero in-degree
    for u in range(num_vertices):
        if in_degree[u] == 0:
            queue.append(u)

    while queue:
        u = queue.popleft()
        topo_order.append(u)

        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    # Check if all vertices are in the topological order
    if len(topo_order) == num_vertices:
        return topo_order
    else:
        # Graph has a cycle
        return []

# Example usage
graph = {
```

```python
    0: [1, 2],
    1: [3],
    2: [3],
    3: []
}

print(topoSortKahn(graph))  # Output: [0, 1, 2, 3]
```

Cycle Detection in Directed Graph (BFS)

```python
from collections import defaultdict, deque

def hasCycle(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    # Build the graph and calculate in-degrees
    for course, pre in prerequisites:
        graph[pre].append(course)
        in_degree[course] += 1

    queue = deque()
    for course in range(numCourses):
        if in_degree[course] == 0:
            queue.append(course)

    count = 0
    while queue:
        course = queue.popleft()
        count += 1

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

    return count == numCourses

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

print(hasCycle(numCourses, prerequisites))  # Output: True (cycle exists)
```

Course Schedule – I

```python
from collections import defaultdict, deque

def canFinish(numCourses, prerequisites):
    # Build graph and calculate in-degrees
    graph = defaultdict(list)
    in_degree = [0] * numCourses

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

    # Initialize queue with courses having no prerequisites
    queue = deque()
    for course in range(numCourses):
        if in_degree[course] == 0:
            queue.append(course)

    # BFS/Topological sort to process courses
    count = 0
    while queue:
        course = queue.popleft()
        count += 1

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

    # If we processed all courses, return True
    return count == numCourses

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

print(canFinish(numCourses, prerequisites))  # Output: False
```

Course Schedule – II

```python
from collections import defaultdict, deque

def findOrder(numCourses, prerequisites):
    # Build graph and calculate in-degrees
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for course, prereq in prerequisites:
```

```python
        graph[prereq].append(course)
        in_degree[course] += 1

    # Initialize queue with courses having no prerequisites
    queue = deque()
    for course in range(numCourses):
        if in_degree[course] == 0:
            queue.append(course)

    # BFS/Topological sort to process courses
    topological_order = []
    while queue:
        course = queue.popleft()
        topological_order.append(course)

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

    # Check if we processed all courses
    if len(topological_order) == numCourses:
        return topological_order
    else:
        return []

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

print(findOrder(numCourses, prerequisites))  # Output: [0, 1, 2, 3]
```

Find eventual safe states

```python
def eventualSafeNodes(graph):
    n = len(graph)
    safe = [0] * n  # 0: unvisited, 1: safe, 2: unsafe

    def dfs(node):
        if safe[node] != 0:
            return safe[node] == 1

        safe[node] = 2  # Mark as visiting

        for neighbor in graph[node]:
            if not dfs(neighbor):
                return False
```

```python
        safe[node] = 1  # Mark as safe
        return True

    result = []
    for i in range(n):
        if dfs(i):
            result.append(i)

    return result

# Example usage
graph = [[1,2],[2,3],[5],[0],[5],[],[]]
print(eventualSafeNodes(graph))  # Output: [2, 4, 5, 6]
```

Alien dictionary

```python
from collections import defaultdict, deque

def alienOrder(words):
    # Step 1: Build the graph
    graph = defaultdict(set)
    in_degree = {c: 0 for word in words for c in word}

    # Step 2: Compare adjacent words to find character order
    for i in range(1, len(words)):
        w1, w2 = words[i-1], words[i]
        min_len = min(len(w1), len(w2))
        for j in range(min_len):
            if w1[j] != w2[j]:
                if w2[j] not in graph[w1[j]]:
                    graph[w1[j]].add(w2[j])
                    in_degree[w2[j]] += 1
                break

    # Step 3: Topological sorting using Kahn's Algorithm
    queue = deque([c for c in in_degree if in_degree[c] == 0])
    topo_order = []

    while queue:
        node = queue.popleft()
        topo_order.append(node)
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)
```

```python
    # Step 4: Check if we have processed all nodes
    if len(topo_order) == len(in_degree):
        return ''.join(topo_order)
    else:
        return ''

# Example usage
words = [
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]

print(alienOrder(words))  # Output: "wertf"
```

Shortest Path in UG with unit weights

```python
from collections import defaultdict, deque

def shortestPath(graph, source, destination):
    if source == destination:
        return 0

    queue = deque([source])
    distance = {source: 0}

    while queue:
        node = queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in distance:
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)

                if neighbor == destination:
                    return distance[neighbor]

    # If destination is not reachable from source
    return -1

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
```

```
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

source = 'A'
destination = 'F'

print(f"Shortest path from {source} to {destination}:", shortestPath(graph,
source,    destination))  # Output: 2
```

Shortest Path in DAG

```
from collections import defaultdict, deque

def shortestPathDAG(graph, source, num_nodes):
    # Step 1: Perform topological sorting using DFS
    topo_order = []
    visited = [False] * num_nodes

    def dfs(node):
        visited[node] = True
        for neighbor, weight in graph[node]:
            if not visited[neighbor]:
                dfs(neighbor)
        topo_order.append(node)

    for node in range(num_nodes):
        if not visited[node]:
            dfs(node)

    topo_order.reverse()

    # Step 2: Initialize distances
    distance = [float('inf')] * num_nodes
    distance[source] = 0

    # Step 3: Relax edges in topological order
    for node in topo_order:
        if distance[node] != float('inf'):
            for neighbor, weight in graph[node]:
                if distance[neighbor] > distance[node] + weight:
                    distance[neighbor] = distance[node] + weight

    return distance

# Example usage
```

```
num_nodes = 6
source = 0
graph = defaultdict(list)
graph[0] = [(1, 5), (2, 3)]
graph[1] = [(3, 6), (2, 2)]
graph[2] = [(3, 7), (4, 4)]
graph[3] = [(4, -1), (5, 1)]
graph[4] = [(5, -2)]

shortest_distances = shortestPathDAG(graph, source, num_nodes)
print("Shortest distances from source node", source)
for node, dist in enumerate(shortest_distances):
    print(f"To node {node}: {dist}")
```

Djisktra's Algorithm

```
import heapq
from collections import defaultdict

def dijkstra(graph, source):
    # Step 1: Initialize data structures
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    priority_queue = [(0, source)]  # (distance, node)
    heapq.heapify(priority_queue)

    # Step 2: Process nodes in priority order
    while priority_queue:
        current_dist, current_node = heapq.heappop(priority_queue)

        # If current distance is greater than recorded distance, skip it
        if current_dist > dist[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight

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

    return dist

# Example usage
```

```python
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('C', 5), ('D', 10)],
    'C': [('D', 3)],
    'D': []
}
source = 'A'

shortest_distances = dijkstra(graph, source)
print("Shortest distances from source node", source)
for node, dist in shortest_distances.items():
    print(f"To node {node}: {dist}")
```

Why priority Queue is used in Djisktra's Algorithm

```python
import heapq
from collections import defaultdict

def dijkstra(graph, source):
    # Initialize distances and priority queue
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    pq = [(0, source)]  # (distance, node)
    heapq.heapify(pq)

    while pq:
        current_dist, u = heapq.heappop(pq)

        # If current distance is greater than recorded distance, skip it
        if current_dist > dist[u]:
            continue

        # Explore neighbors
        for v, weight in graph[u]:
            distance = current_dist + weight

            # If found shorter path to neighbor, update distance and push to
priority queue
            if distance < dist[v]:
                dist[v] = distance
                heapq.heappush(pq, (distance, v))

    return dist

# Example usage
graph = defaultdict(list)
graph['A'] = [('B', 4), ('C', 2)]
```

```python
graph['B'] = [('C', 5), ('D', 10)]
graph['C'] = [('D', 3)]
graph['D'] = []

source = 'A'
shortest_distances = dijkstra(graph, source)
print("Shortest distances from source node", source)
for node, dist in shortest_distances.items():
    print(f"To node {node}: {dist}")
```

Shortest path in a binary maze

```python
from collections import deque

def shortestPathBinaryMaze(maze):
    if not maze or not maze[0]:
        return -1

    m, n = len(maze), len(maze[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    # BFS setup
    queue = deque([(0, 0)])
    visited = [[False] * n for _ in range(m)]
    visited[0][0] = True
    dist = [[float('inf')] * n for _ in range(m)]
    dist[0][0] = 0

    # BFS traversal
    while queue:
        x, y = queue.popleft()

        # Check all possible directions
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny] and 
maze[nx][ny] == 0:
                visited[nx][ny] = True
                dist[nx][ny] = dist[x][y] + 1
                queue.append((nx, ny))

                # Check if reached the bottom-right corner
                if (nx, ny) == (m - 1, n - 1):
                    return dist[nx][ny]

    # If bottom-right corner is unreachable
```

```
    return -1

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

print("Shortest path length in maze:", shortestPathBinaryMaze(maze))  #
Output: 7
```

Path with minimum effort

```
import heapq

def minimumEffortPath(heights):
    if not heights or not heights[0]:
        return 0

    m, n = len(heights), len(heights[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def bfs(max_effort):
        queue = [(0, 0)]  # (effort, x, y)
        visited = [[False] * n for _ in range(m)]
        visited[0][0] = True

        while queue:
            current_effort, x, y = heapq.heappop(queue)

            if x == m - 1 and y == n - 1:
                return True

            for dx, dy in directions:
                nx, ny = x + dx, y + dy

                if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny]:
                    next_effort = max(current_effort, abs(heights[x][y] -
heights[nx][ny]))

                    if next_effort <= max_effort:
                        visited[nx][ny] = True
                        heapq.heappush(queue, (next_effort, nx, ny))

        return False
```

```python
    low, high = 0, max(max(row) for row in heights)

    while low < high:
        mid = (low + high) // 2
        if bfs(mid):
            high = mid
        else:
            low = mid + 1

    return low

# Example usage
heights = [
    [1, 2, 2],
    [3, 8, 2],
    [5, 3, 5]
]

print("Minimum maximum effort path:", minimumEffortPath(heights)) #Output:2
```

Cheapest flights within k stops

```python
import heapq
from collections import defaultdict

def findCheapestPrice(n, flights, src, dst, k):
    # Build graph
    graph = defaultdict(list)
    for u, v, cost in flights:
        graph[u].append((v, cost))

    # Min-heap for Dijkstra's algorithm: (cost, node, stops)
    pq = [(0, src, 0)]  # (current_cost, current_node, current_stops)
    heapq.heapify(pq)

    # Dictionary to track the minimum cost to reach each node with given
number of stops
    visited = {}

    while pq:
        current_cost, u, stops = heapq.heappop(pq)

        if u == dst:
            return current_cost

        if stops > k:
```

```
            continue

        if (u, stops) in visited and visited[(u, stops)] < current_cost:
            continue

        visited[(u, stops)] = current_cost

        for v, cost in graph[u]:
            heapq.heappush(pq, (current_cost + cost, v, stops + 1))

    return -1

# Example usage
n = 3
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))  # Output: 200
```

Network Delay time

```
import heapq
from collections import defaultdict

def networkDelayTime(times, n, k):
    # Step 1: Build the graph
    graph = defaultdict(list)
    for u, v, time in times:
        graph[u].append((v, time))

    # Step 2: Dijkstra's Algorithm using priority queue
    pq = [(0, k)]  # (time, node)
    dist = {i: float('inf') for i in range(1, n + 1)}
    dist[k] = 0

    while pq:
        current_time, u = heapq.heappop(pq)

        if current_time > dist[u]:
            continue

        for v, time in graph[ u]:
            if dist[u] + time < dist[v]:
                dist[v] = dist[u] + time
```

```python
                heapq.heappush(pq, (dist[v], v))

    # Step 3: Find the maximum time in dist (excluding the source node k)
    max_time = max(dist.values())

    return max_time if max_time < float('inf') else -1

# Example usage
times = [[2,1,1],[2,3,1],[3,4,1]]
n = 4
k = 2
print("Network Delay Time:", networkDelayTime(times, n, k))  # Output: 2
```

.

Number of ways to arrive at destination

```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 dfs_count_paths(self, start, end, visited, path_count):
        visited[start] = True

        # If current vertex is same as destination, increment path_count
        if start == end:
            path_count[0] += 1
        else:
            # Recur for all the vertices adjacent to this vertex
            for neighbor in self.graph[start]:
                if not visited[neighbor]:
                    self.dfs_count_paths(neighbor, end, visited, path_count)

        # Mark current vertex as unvisited to allow other paths from it
        visited[start] = False

    def count_paths(self, start, end):
        visited = defaultdict(bool)
        path_count = [0]  # Using list to pass by reference

        self.dfs_count_paths(start, end, visited, path_count)

        return path_count[0]
```

```python
# Example usage:
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(3, 3)

    start_node = 2
    end_node = 3
    print(f"Number of ways to reach {end_node} from {start_node}:
{g.count_paths(start_node, end_node)}")
```

Minimum steps to reach end from start by performing multiplication and mod operations with array elements

```python
from collections import deque

def min_steps_to_reach_end(arr):
    n = len(arr)
    if n == 0:
        return -1

    queue = deque([(1, 0)])  # (current value, steps)
    visited = [False] * n
    visited[0] = True

    while queue:
        current_value, steps = queue.popleft()

        if current_value == n:
            return steps

        for i in range(n):
            if not visited[i]:
                new_value_mul = current_value * arr[i]
                new_value_mod = current_value % arr[i]

                if new_value_mul <= n:
                    if not visited[new_value_mul - 1]:
                        visited[new_value_mul - 1] = True
                        queue.append((new_value_mul, steps + 1))

                if new_value_mod <= n:
                    if not visited[new_value_mod - 1]:
```

```
                            visited[new_value_mod - 1] = True
                            queue.append((new_value_mod, steps + 1))

    return -1

# Example usage:
arr = [3, 2, 5, 8]
print(min_steps_to_reach_end(arr))  # Output: 2
```

Bellman Ford Algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.graph = []    # List to store edges

    # Function to add an edge to the graph
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # Utility function to print the solution
    def print_distances(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print(f"{i}\t\t{dist[i]}")

    def bellman_ford(self, src):
        # Step 1: Initialize distances from src to all other vertices as
INFINITE
        dist = [float("Inf")] * self.V
        dist[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(self.V - 1):
            # Update dist value and parent index of the adjacent vertices of
            # the picked vertex. Consider only those vertices which are
still in
            # queue
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[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 dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        # Print the distance array
        self.print_distances(dist)
```

Floyd Warshal Algorithm

```
  INF = float('inf')

  def floyd_warshall(graph):
      """
    Function to find the shortest paths between all pairs of vertices using
Floyd-Warshall   algorithm.

      Parameters:
      graph : list of list of numbers
        Adjacency matrix representation of the graph where graph[i][j] is the
weight of   the edge from vertex i to vertex j.
        If there is no edge, graph[i][j] should be INF (infinity), except
graph[i][i]   should be 0.

      Returns:
      dist : list of list of numbers
          A 2D list dist where dist[i][j] is the shortest distance from vertex
i to vertex j.
      """
      n = len(graph)
      dist = [row[:] for row in graph]  # Make a copy of the input graph
matrix to work with

      # Applying Floyd-Warshall algorithm
      for k in range(n):
          for i in range(n):
              for j in range(n):
                  if dist[i][j] > dist[i][k] + dist[k][j]:
                      dist[i][j] = dist[i][k] + dist[k][j]

      return dist

  # Example usage:
  if __name__ == "__main__":
      # Example graph adjacency matrix
      graph = [
```

```
        [0, 5, INF, 10],
        [INF, 0, 3, INF],
        [INF, INF, 0, 1],
        [INF, INF, INF, 0]
    ]

    # Finding shortest paths using Floyd-Warshall algorithm
    shortest_paths = floyd_warshall(graph)

    # Printing the shortest paths
    print("Shortest distances between every pair of vertices:")
    for row in shortest_paths:
        print(row)
```

Minimum Spanning Tree

```
import heapq

def prim_mst(graph):
    # Number of vertices in the graph
    V = len(graph)

    # Priority queue to store vertices along with their key value
    pq = [(0, 0)]  # (key, vertex), starting with vertex 0 with key 0
    key = [float('inf')] * V  # To store key values that will pick the
minimum weight edge
    parent = [-1] * V  # Array to store constructed MST

    # Key value for the first vertex is 0
    key[0] = 0

    # To keep track of vertices included in MST
    mst_set = [False] * V

    while pq:
        # Extract the vertex with the smallest key value
        u_key, u = heapq.heappop(pq)

        # Include the extracted vertex in MST
        mst_set[u] = True

        # Check all adjacent vertices of u
        for v, weight in graph[u]:
            # If v is not yet in MST and weight of (u,v) is smaller than
current key of v
            if not mst_set[v] and weight < key[v]:
                # Update key value and priority queue
```

```
                key[v] = weight
                heapq.heappush(pq, (key[v], v))
                parent[v] = u

    # Print the constructed MST
    print("Edge \tWeight")
    for i in range(1, V):
        print(parent[i], "-", i, "\t", graph[i][parent[i]])

# Example usage:
if __name__ == '__main__':
    # Example graph as an adjacency list
    graph = {
        0: [(1, 2), (2, 3)],      # (neighbor, weight)
        1: [(0, 2), (2, 1), (3, 1)],
        2: [(0, 3), (1, 1), (3, 4), (4, 5)],
        3: [(1, 1), (2, 4), (4, 3)],
        4: [(2, 5), (3, 3)]
    }

    # Function call to find MST
    prim_mst(graph)
```

Prim's Algorithm

```
import heapq

def prim_mst(graph):
    V = len(graph)  # Number of vertices
    pq = []         # Priority queue to store vertices with their key values
    heapq.heappush(pq, (0, 0))  # Start from vertex 0 with key 0
    key = [float('inf')] * V    # Key values to pick minimum weight edge
    parent = [-1] * V           # Array to store constructed MST
    mst_set = [False] * V       # To keep track of vertices included in MST

    key[0] = 0  # Starting vertex with key 0

    while pq:
        u_key, u = heapq.heappop(pq)  # Extract vertex with minimum key
value
        mst_set[u] = True

        for v, weight in graph[u]:
            if not mst_set[v] and weight < key[v]:
                key[v] = weight
                parent[v] = u
                heapq.heappush(pq, (key[v], v))
```

```
    # Print the edges of the MST
    print("Edge \tWeight")
    for i in range(1, V):
        print(parent[i], "-", i, "\t", graph[i][parent[i]])

if __name__ == '__main__':
    # Example graph as an adjacency list
    graph = {
        0: [(1, 2), (2, 3)],     # (neighbor, weight)
        1: [(0, 2), (2, 1), (3, 1)],
        2: [(0, 3), (1, 1), (3, 4), (4, 5)],
        3: [(1, 1), (2, 4), (4, 3)],
        4: [(2, 5), (3, 3)]
    }

    prim_mst(graph)
```

Disjoint Set [Union by Rank]

```
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * 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

    def connected(self, u, v):
        return self.find(u) == self.find(v)
```

```
# Example usage:
if __name__ == "__main__":
    n = 5
    dsu = DisjointSetUnion(n)

    dsu.union(0, 1)
    dsu.union(2, 3)
    dsu.union(0, 4)

    print(dsu.connected(1, 4))  # Should print True
    print(dsu.connected(2, 4))  # Should print False

    dsu.union(1, 3)

    print(dsu.connected(2, 4))  # Should print True now after merging paths
```

Disjoint Set [Union by Size]

```
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [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 size
            if self.size[root_u] > self.size[root_v]:
                self.parent[root_v] = root_u
                self.size[root_u] += self.size[root_v]
            else:
                self.parent[root_u] = root_v
                self.size[root_v] += self.size[root_u]

    def connected(self, u, v):
        return self.find(u) == self.find(v)

# Example usage:
if __name__ == "__main__":
    n = 5
```

```python
    dsu = DisjointSetUnion(n)

    dsu.union(0, 1)
    dsu.union(2, 3)
    dsu.union(0, 4)

    print(dsu.connected(1, 4))  # Should print True
    print(dsu.connected(2, 4))  # Should print False

    dsu.union(1, 3)

    print(dsu.connected(2, 4))  # Should print True now after merging paths
```

Kruskal's Algorithm

```python
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * 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_mst(graph):
    # Number of vertices in the graph
    V = len(graph)
    edges = []
```

```python
    # Step 1: Create a list of all edges in the graph, sorted by weight
    for u in range(V):
        for v, weight in graph[u]:
            edges.append((weight, u, v))

    edges.sort()  # Sort edges by weight

    # Step 2: Initialize Disjoint Set Union
    dsu = DisjointSetUnion(V)

    # Step 3: Initialize an empty list to store MST edges
    mst = []
    mst_weight = 0

    # Step 4: Process each edge in sorted order
    for weight, u, v in edges:
        if dsu.union(u, v):
            mst.append((u, v, weight))
            mst_weight += weight
            # Early termination if MST is complete
            if len(mst) == V - 1:
                break

    return mst, mst_weight

# Example usage:
if __name__ == '__main__':
    # Example graph as an adjacency list
    graph = {
        0: [(1, 2), (2, 3)],      # (neighbor, weight)
        1: [(0, 2), (2, 1), (3, 1)],
        2: [(0, 3), (1, 1), (3, 4), (4, 5)],
        3: [(1, 1), (2, 4), (4, 3)],
        4: [(2, 5), (3, 3)]
    }

    # Function call to find MST using Kruskal's algorithm
    mst, mst_weight = kruskal_mst(graph)

    # Print the MST edges and their total weight
    print("Edges in the Minimum Spanning Tree:")
    for u, v, weight in mst:
        print(f"{u} - {v} : {weight}")
    print(f"Total weight of MST: {mst_weight}")
```

Number of operations to make network connected

```python
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n  # Number of components initially

    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
            self.count -= 1  # Decrease the count of components

    def connected_components(self):
        return self.count

def number_of_operations_to_connect(n, edges):
    dsu = DisjointSetUnion(n)

    for u, v in edges:
        dsu.union(u, v)

    return dsu.connected_components() - 1

# Example usage:
if __name__ == '__main__':
    n = 6  # Number of nodes
    edges = [(0, 1), (1, 2), (3, 4)]  # Example edges

    operations_needed = number_of_operations_to_connect(n, edges)
    print(f"Number of operations needed to connect the network: 
{operations_needed}")
```

Most stones removed with same rows or columns

```python
import networkx as nx

def max_stones_removed(grid):
    rows, cols = len(grid), len(grid[0])

    # Create a bipartite graph
    G = nx.Graph()

    # Add nodes for rows and columns
    row_nodes = [f'r{i}' for i in range(rows)]
    col_nodes = [f'c{j}' for j in range(cols)]
    G.add_nodes_from(row_nodes, bipartite=0)
    G.add_nodes_from(col_nodes, bipartite=1)

    # Add edges based on grid
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                G.add_edge(f'r{r}', f'c{c}')

    # Find maximum bipartite matching
    matching = nx.bipartite.maximum_matching(G, top_nodes=row_nodes)

    # Size of matching is the maximum number of stones that can be removed
    max_removed = len(matching)

    return max_removed

# Example usage:
if __name__ == '__main__':
    grid = [
        [1, 0, 1],
        [0, 1, 0],
        [1, 0, 1]
    ]

    max_stones = max_stones_removed(grid)
    print(f"Maximum number of stones that can be removed: {max_stones}")
```

Accounts merge

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

    def find(self, u):
```

```python
        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

    def make_set(self, u):
        if u not in self.parent:
            self.parent[u] = u
            self.rank[u] = 0

def accounts_merge(accounts):
    uf = UnionFind()
    email_to_name = {}
    email_to_id = {}
    id_count = 0

    # First pass: Build Union-Find structure
    for account in accounts:
        name = account[0]
        for email in account[1:]:
            email_to_name[email] = name
            if email not in email_to_id:
                email_to_id[email] = id_count
                uf.make_set(id_count)
                id_count += 1
            uf.union(email_to_id[account[1]], email_to_id[email])

    # Second pass: Group emails by their root email
    root_to_emails = {}
    for email in email_to_name:
        root = uf.find(email_to_id[email])
        if root not in root_to_emails:
            root_to_emails[root] = []
        root_to_emails[root].append(email)
```

```python
        # Format result in required format
        merged_accounts = []
        for root, emails in root_to_emails.items():
            merged_accounts.append([email_to_name[emails[0]]] + sorted(emails))

        return merged_accounts

# Example usage:
if __name__ == "__main__":
    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_merge(accounts)
    for account in merged:
        print(account)
```

Number of island II

```python
class UnionFind:
    def __init__(self, n):
        self.parent = [-1] * n
        self.rank = [0] * n
        self.count = 0

    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
            self.count -= 1  # Decrease the count of islands
```

```python
    def add(self, u):
        if self.parent[u] == -1:
            self.parent[u] = u
            self.rank[u] = 1
            self.count += 1

    def get_count(self):
        return self.count

def num_islands2(m, n, positions):
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    grid = [[0] * n for _ in range(m)]
    uf = UnionFind(m * n)
    result = []

    for idx, (x, y) in enumerate(positions):
        if grid[x][y] == 0:
            grid[x][y] = 1
            uf.add(x * n + y)
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] == 1:
                    uf.union(x * n + y, nx * n + ny)
        result.append(uf.get_count())

    return result

# Example usage:
if __name__ == "__main__":
    m = 3
    n = 3
    positions = [[0, 0], [0, 1], [1, 2], [2, 1]]
    print(num_islands2(m, n, positions))  # Output: [1, 1, 2, 3]
```

Swim in rising water

```python
from collections import deque

def swimInWater(grid):
    n = len(grid)
    left, right = grid[0][0], n * n - 1

    def can_reach_target(T):
        if grid[0][0] > T or grid[n-1][n-1] > T:
            return False
```

```
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        queue = deque([(0, 0)])
        visited = set((0, 0))

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

            if x == n-1 and y == n-1:
                return True

            for dx, dy in directions:
                nx, ny = x + dx, y + dy
              if 0 <= nx < n and 0 <= ny < n and (nx, ny) not in visited and
grid[nx]  [ny] <= T:
                    visited.add((nx, ny))
                    queue.append((nx, ny))

        return False

    while left < right:
        mid = (left + right) // 2
        if can_reach_target(mid):
            right = mid
        else:
            left = mid + 1

    return left

if __name__ == "__main__":
    grid = [
        [0, 2],
        [1, 3]
    ]
    print(swimInWater(grid))  # Output: 3
```

Bridges in Graph

```
def find_bridges(graph):
    n = len(graph)
    if n == 0:
        return []

    # Initialization
    disc = [-1] * n  # discovery time of each vertex
    low = [-1] * n   # earliest visited vertex reachable from subtree rooted
with i
    parent = [-1] * n
```

```python
    bridges = []
    time = [0]  # used to track discovery time

    # DFS function to find bridges
    def dfs(u):
        nonlocal time
        disc[u] = low[u] = time[0]
        time[0] += 1

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

                # Check if the subtree rooted at v has a connection back to
one of the   ancestors of 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] > disc[u]:
                    bridges.append((u, v))

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

    # Call DFS for all vertices in case the graph is disconnected
    for i in range(n):
        if disc[i] == -1:
            dfs(i)

    return bridges

if __name__ == "__main__":
    graph = {
        0: [1, 2],
        1: [0, 2],
        2: [0, 1, 3, 4],
        3: [2, 4],
        4: [2, 3, 5],
        5: [4]
    }
    bridges = find_bridges(graph)
    print("Bridges in the graph:")
    for bridge in bridges:
        print(bridge)
```

Articulation Point

```
def find_articulation_points(graph):
    n = len(graph)
    if n == 0:
        return []

    # Initialization
    disc = [-1] * n  # discovery time of each vertex
    low = [-1] * n   # earliest visited vertex reachable from subtree rooted
with i
    parent = [-1] * n
    articulation_points = [False] * n
    time = [0]  # used to track discovery time

    # DFS function to find articulation points
    def dfs(u):
        nonlocal time
        children = 0
        disc[u] = low[u] = time[0]
        time[0] += 1

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

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

                # u is an articulation point in following cases

                # (1) u is root of DFS tree and has two or more children
                if parent[u] == -1 and children > 1:
                    articulation_points[u] = True

               # (2) u is not root and low value of one of its child is more
than   discovery value of u
                if parent[u] != -1 and low[v] >= disc[u]:
                    articulation_points[u] = True

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

    # Call DFS for all vertices in case the graph is disconnected
    for i in range(n):
```

```python
        if disc[i] == -1:
            dfs(i)

    # Collect all articulation points
    result = []
    for i in range(n):
        if articulation_points[i]:
            result.append(i)

    return result

# Example usage:
if __name__ == "__main__":
    graph = {
        0: [1, 2],
        1: [0, 2],
        2: [0, 1, 3, 4],
        3: [2, 4],
        4: [2, 3, 5],
        5: [4]
    }
    articulation_points = find_articulation_points(graph)
    print("Articulation points in the graph:")
    print(articulation_points)
```

Kosaraju's Algorithm

```python
from collections import defaultdict, deque

def kosaraju(graph, n):
    # Step 1: Perform DFS on original graph and store finish times
    def dfs1(v):
        visited[v] = True
        for neighbor in graph[v]:
            if not visited[neighbor]:
                dfs1(neighbor)
        finish_stack.append(v)

    # Step 2: Perform DFS on reversed graph using finish times
    def dfs2(v, component):
        visited[v] = True
        component.append(v)
        for neighbor in reversed_graph[v]:
            if not visited[neighbor]:
                dfs2(neighbor, component)

    # Initialize variables
```

```
    visited = [False] * n
    finish_stack = []

    # Step 1: Perform DFS on original graph to fill finish_stack
    for i in range(n):
        if not visited[i]:
            dfs1(i)

    # Step 2: Transpose the graph
    reversed_graph = defaultdict(list)
    for u in range(n):
        for v in graph[u]:
            reversed_graph[v].append(u)

    # Reset visited array for second DFS
    visited = [False] * n
    sccs = []

    # Step 3: Perform DFS on reversed graph in order of decreasing finish
time
    while finish_stack:
        v = finish_stack.pop()
        if not visited[v]:
            component = []
            dfs2(v, component)
            sccs.append(component)

    return sccs

# Example usage:
if __name__ == "__main__":
    graph = {
        0: [1],
        1: [2, 3],
        2: [0],
        3: [4],
        4: []
    }
    n = 5  # Number of vertices

    sccs = kosaraju(graph, n)
    print("Strongly Connected Components:")
    for scc in sccs:
        print(scc)
```

Dynamic Programming

Climbing Stairs

```python
def climb_stairs(n):
    if n == 0:
        return 1  # There's one way to stay at the ground (doing nothing)
    if n == 1:
        return 1  # Only one way to reach the first step (take one step)

    # Create a list to store the number of ways to reach each step
    dp = [0] * (n + 1)
    dp[0] = 1  # There's one way to stay at the ground (doing nothing)
    dp[1] = 1  # Only one way to reach the first step (take one step)

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

    return dp[n]

# Example usage:
n = 4
print(f"Number of distinct ways to climb {n} steps: {climb_stairs(n)}")
```

Frog Jump(DP-3)

```python
def canCross(stones):
    if len(stones) < 2:
        return True

    # Dictionary to store the reachability status of each stone
    dp = {}

    # Initialize the dictionary with sets for each stone's positions
    for stone in stones:
        dp[stone] = set()

    dp[0].add(0)  # Start at the first stone with 0 jump length

    for stone in stones:
        for jump in dp[stone]:
            for jump_length in [jump - 1, jump, jump + 1]:
                if jump_length > 0 and stone + jump_length in dp:
                    dp[stone + jump_length].add(jump_length)

    # Check if the last stone is reachable
    return len(dp[stones[-1]]) > 0
```

```python
# Example usage:
stones = [0,1,3,5,6,8,12,17]
print(canCross(stones))  # Output: True

stones = [0,1,2,3,4,8,9,11]
print(canCross(stones))  # Output: False
```

Frog Jump with k distances(DP-4)

```python
def canCross(stones, k):
    if len(stones) < 2:
        return True

    # Create a dictionary to store reachability of each stone
    dp = {}

    # Initialize the dictionary with sets for each stone's positions
    for stone in stones:
        dp[stone] = set()

    dp[0].add(0)  # Start at the first stone with 0 jump length

    for stone in stones:
        for jump in dp[stone]:
            for jump_length in [jump - 1, jump, jump + 1]:
                if jump_length > 0 and (stone + jump_length) in dp:
                    dp[stone + jump_length].add(jump_length)

        # Also try jumps from the current stone with k distances
        for jump_length in k:
            if jump_length > 0 and (stone + jump_length) in dp:
                dp[stone + jump_length].add(jump_length)

    # Check if the last stone is reachable
    return len(dp[stones[-1]]) > 0

# Example usage:
stones = [0,1,3,5,6,8,12,17]
k = {1, 2, 3}
print(canCross(stones, k))  # Output: True

stones = [0,1,2,3,4,8,9,11]
k = {1, 2, 3}
print(canCross(stones, k))  # Output: False
```

Maximum sum of non-adjacent elements (DP 5)

```python
def max_sum_non_adjacent(nums):
    n = len(nums)
    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

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

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

    return dp[-1]

# Example usage:
nums = [2, 4, 6, 2, 5]
print(max_sum_non_adjacent(nums))  # Output: 13 (because 2 + 6 + 5 = 13)

nums = [5, 1, 1, 5]
print(max_sum_non_adjacent(nums))  # Output: 10 (because 5 + 5 = 10)

nums = [1, 2, 3, 4, 5, 6]
print(max_sum_non_adjacent(nums))  # Output: 12 (because 2 + 4 + 6 = 12)

nums = [5, -1, -1, 5]
print(max_sum_non_adjacent(nums))  # Output: 10 (because 5 + 5 = 10)

nums = []
print(max_sum_non_adjacent(nums))  # Output: 0 (empty array case)
```

House Robber (DP 6)

```python
def rob(nums):
    n = len(nums)
    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

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

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

    return dp[-1]

# Example usage:
nums = [1, 2, 3, 1]
print(rob(nums))  # Output: 4 (Rob house 1 and 3, total = 1 + 3 = 4)

nums = [2, 7, 9, 3, 1]
print(rob(nums))  # Output: 12 (Rob house 1, 3, and 5, total = 2 + 9 + 1 = 
12)
```

Ninja's Training (DP 7)

```python
def max_skill_points(training_sessions):
    n = len(training_sessions)
    if n == 0:
        return 0
    elif n == 1:
        return training_sessions[0]

    dp = [0] * n
    dp[0] = training_sessions[0]
    if n > 1:
        dp[1] = max(training_sessions[0], training_sessions[1])

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

    return dp[-1]

# Example usage:
training_sessions = [3, 2, 7, 10, 5]
print(max_skill_points(training_sessions))  # Output: 15 (attend sessions 1, 
3, and 5)

training_sessions = [5, 1, 1, 5]
print(max_skill_points(training_sessions))  # Output: 10 (attend sessions 1 
and 4)

training_sessions = [1, 2, 3, 4, 5, 6]
print(max_skill_points(training_sessions))  # Output: 12 (attend sessions 2, 
4, and 6)
```

Grid Unique Paths : DP on Grids (DP8)

```
def unique_paths(m, n):
    # Initialize dp table
    dp = [[0] * n for _ in range(m)]

    # Base case: There is only 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]  # Move from above
            if j > 0:
                dp[i][j] += dp[i][j-1]  # Move from left

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

# Example usage:
m = 3
n = 7
print(unique_paths(m, n))  # Output: 28 (number of unique paths from (0,0)
to (2,6) in a 3x7 grid)
```

Grid Unique Paths 2 (DP 9)

```
def unique_paths_with_obstacles(grid):
    m = len(grid)
    n = len(grid[0])

    # Edge case: Starting position is blocked
    if grid[0][0] == 1:
        return 0

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

    # Base case
    dp[0][0] = 1

    # Fill the dp table
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 0:  # Only process clear cells
```

```
                if i > 0:
                    dp[i][j] += dp[i-1][j]  # Move from above
                if j > 0:
                    dp[i][j] += dp[i][j-1]  # Move from left
    # The result is the number of unique paths to reach the bottom-right
corner
    return dp[m-1][n-1]

# Example usage:
grid1 = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0]
]
print(unique_paths_with_obstacles(grid1))  # Output: 2 (Two unique paths)

grid2 = [
    [0, 1],
    [0, 0]
]
print(unique_paths_with_obstacles(grid2))  # Output: 1 (One unique path)
```

Minimum path sum in Grid (DP 10)

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

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

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

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

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

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

    # Fill the rest of 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])

    # The result is the bottom-right corner of the dp table
    return dp[m-1][n-1]
```

Minimum path sum in Triangular Grid (DP 11)

```
def minPathSum(triangle):
    if not triangle:
        return 0

    n = len(triangle)

    # Initialize the dp table
    dp = [[0] * n for _ in range(n)]

    # Initialize the top of the triangle
    dp[0][0] = triangle[0][0]

    # Fill the dp table
    for i in range(1, n):
        dp[i][0] = dp[i-1][0] + triangle[i][0]
        for j in range(1, i):
            dp[i][j] = triangle[i][j] + min(dp[i-1][j-1], dp[i-1][j])
        dp[i][i] = dp[i-1][i-1] + triangle[i][i]

    # Find the minimum path sum in the last row
    min_path_sum = dp[n-1][0]
    for j in range(1, n):
        min_path_sum = min(min_path_sum, dp[n-1][j])

    return min_path_sum

# Example usage:
triangle = [
    [2],
    [3, 4],
    [6, 5, 7],
    [4, 1, 8, 3]
]
print(minPathSum(triangle))  # Output: 11
```

Minimum/Maximum Falling Path Sum (DP-12)

```python
def minFallingPathSum(matrix):
    if not matrix:
        return 0

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

    # Initialize the dp table for minimum falling path sum
    min_dp = [[0] * cols for _ in range(rows)]

    # Initialize the first row of min_dp
    for j in range(cols):
        min_dp[0][j] = matrix[0][j]

    # Fill the min_dp table
    for i in range(1, rows):
        for j in range(cols):
            if j == 0:
                min_dp[i][j] = matrix[i][j] + min(min_dp[i-1][j], min_dp[i-1][j+1])
            elif j == cols - 1:
                min_dp[i][j] = matrix[i][j] + min(min_dp[i-1][j-1], min_dp[i-1][j])
            else:
                min_dp[i][j] = matrix[i][j] + min(min_dp[i-1][j-1], min_dp[i-1][j], min_dp[i-1][j+1])

    # Find the minimum value in the last row of min_dp
    min_path_sum = min(min_dp[rows-1])

    return min_path_sum

def maxFallingPathSum(matrix):
    if not matrix:
        return 0

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

    # Initialize the dp table for maximum falling path sum
    max_dp = [[0] * cols for _ in range(rows)]

    # Initialize the first row of max_dp
    for j in range(cols):
        max_dp[0][j] = matrix[0][j]

    # Fill the max_dp table
    for i in range(1, rows):
```

```
        for j in range(cols):
            if j == 0:
                max_dp[i][j] = matrix[i][j] + max(max_dp[i-1][j], max_dp[i-1][j+1])
            elif j == cols - 1:
                max_dp[i][j] = matrix[i][j] + max(max_dp[i-1][j-1],
max_dp[i-1][j])
            else:
                max_dp[i][j] = matrix[i][j] + max(max_dp[i-1][j-1],
max_dp[i-1][j], max_dp[i-1][j+1])

    # Find the maximum value in the last row of max_dp
    max_path_sum = max(max_dp[rows-1])

    return max_path_sum

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

print("Minimum Falling Path Sum:", minFallingPathSum(matrix))  # Output: 7
print("Maximum Falling Path Sum:", maxFallingPathSum(matrix))  # Output: 18
```

3-d DP : Ninja and his friends (DP-13)

```
def ninja_and_friends_3d_dp(grid):
    if not grid:
        return 0

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

    # Initialize a 3D dp table
    dp = [[[0] * p for _ in range(n)] for _ in range(m)]

    # Base case initialization
    dp[0][0][0] = grid[0][0][0]

    # Fill the dp table
    for i in range(m):
        for j in range(n):
            for k in range(p):
                if i == 0 and j == 0 and k == 0:
```

```python
                    continue
                min_prev = float('inf')
                if i > 0:
                    min_prev = min(min_prev, dp[i-1][j][k])
                if j > 0:
                    min_prev = min(min_prev, dp[i][j-1][k])
                if k > 0:
                    min_prev = min(min_prev, dp[i][j][k-1])

                dp[i][j][k] = grid[i][j][k] + min_prev

    # The result will be the minimum value at the endpoint
    return dp[m-1][n-1][p-1]

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

print(ninja_and_friends_3d_dp(grid))  # Output: 16 (Example of finding
minimum path sum)
```

Subset sum equal to target (DP- 14)

```python
def subsetSum(nums, target):
    n = len(nums)

    # Initialize a 2D dp array
    dp = [[False] * (target + 1) for _ in range(n + 1)]

    # Base case initialization
    dp[0][0] = True  # sum of 0 is always possible with an empty subset

    # Fill the first column (sum = 0) with True
    for i in range(1, n + 1):
        dp[i][0] = True

    # Fill the dp table
    for i in range(1, n + 1):
        for j in range(1, target + 1):
```

```python
            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(subsetSum(nums, target))  # Output: True (subset [3, 8] sums up to 11)
```

Partition Equal Subset Sum (DP- 15)

```python
def canPartition(nums):
    total_sum = sum(nums)

    # If total_sum is odd, cannot partition into equal sum subsets
    if total_sum % 2 != 0:
        return False

    target = total_sum // 2

    # Initialize a dp array
    dp = [False] * (target + 1)
    dp[0] = True

    # Fill the dp array
    for num in nums:
        for j in range(target, num-1, -1):
            dp[j] = dp[j] or dp[j - num]

    return dp[target]

# Example usage:
nums = [1, 5, 11, 5]
print(canPartition(nums))  # Output: True (Partition into subsets [1, 5, 5]
and [11] with equal sum)
```

Partition Set Into 2 Subsets With Min Absolute Sum Diff (DP- 16)

```python
def minSubsetSumDiff(nums):
    total_sum = sum(nums)
    target = total_sum // 2

    # Initialize dp array for subset sum problem
```

```
    dp = [False] * (target + 1)
    dp[0] = True

    # Fill dp array
    for num in nums:
        for j in range(target, num-1, -1):
            dp[j] = dp[j] or dp[j - num]

    # Find the minimum absolute difference
    min_diff = float('inf')
    for j in range(target + 1):
        if dp[j]:
            min_diff = min(min_diff, abs(total_sum - 2 * j))

    return min_diff

# Example usage:
nums = [1, 2, 3, 9]
print(minSubsetSumDiff(nums))  # Output: 3 (Partition into subsets [1, 2]
and [3, 9], with sums 3 and 12, respectively)
```

Count Subsets with Sum K (DP - 17)

```
def countSubsetsWithSumK(nums, K):
    n = len(nums)

    # Initialize a 2D dp array
    dp = [[0] * (K + 1) for _ in range(n + 1)]

    # Base case initialization
    dp[0][0] = 1  # There's one way to make sum 0 with an empty subset

    # Fill the dp table
    for i in range(1, n + 1):
        for j in range(K + 1):
            dp[i][j] = dp[i-1][j]  # Exclude the current element nums[i-1]
            if j >= nums[i-1]:
                dp[i][j] += dp[i-1][j - nums[i-1]]  # Include the current
element nums[i-1]

    return dp[n][K]

nums = [1, 2, 3, 4, 5]
K = 7
print(countSubsetsWithSumK(nums, K))  # Output: 2 (Subsets with sums 2+5 and
3+4)
```

Count Partitions with Given Difference (DP - 18)

```
def countPartitionsWithGivenDifference(nums, D):
    total_sum = sum(nums)

    # Check if the required sums are integers and within bounds
    if (total_sum - D) % 2 != 0 or (total_sum + D) % 2 != 0:
        return 0

    S2 = (total_sum - D) // 2
    S1 = (total_sum + D) // 2

    # Initialize a dp array for subset sum problem
    dp = [0] * (total_sum + 1)
    dp[0] = 1

    # Fill dp array to count subsets summing up to each possible sum
    for num in nums:
        for j in range(total_sum, num - 1, -1):
            dp[j] += dp[j - num]

    # Check if S1 and S2 are valid indices
    if S2 >= 0 and S2 <= total_sum and S1 >= 0 and S1 <= total_sum:
        return dp[S1] * dp[S2]
    else:
        return 0

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

0/1 Knapsack (DP - 19)

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 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][W]
```

```python
# Example usage:
weights = [2, 2, 3]
values = [6, 10, 12]
W = 5
print(knapsack(weights, values, W))  # Output: 22
```

Minimum Coins (DP - 20)

```python
def minimum_coins(coins, V):
    dp = [float('inf')] * (V + 1)
    dp[0] = 0

    for coin in coins:
        for v in range(coin, V + 1):
            dp[v] = min(dp[v], dp[v - coin] + 1)

    return dp[V]

# Example usage:
coins = [1, 5, 10, 25]
V = 30
print(minimum_coins(coins, V))  # Output: 4
```

Target Sum (DP - 21)

```python
def find_target_sum_ways(nums, S):
    # Calculate the total sum of the nums array
    total_sum = sum(nums)

    # If S is not achievable given the total sum of nums, return 0
    if S > total_sum or (total_sum + S) % 2 != 0:
        return 0

    target = (total_sum + S) // 2

    # Initialize dp array with zeros
    dp = [0] * (target + 1)
    dp[0] = 1  # There's one way to make sum 0: by choosing no elements

    # Iterate through each number in nums
    for num in nums:
        # Update dp array backwards to avoid overwriting previous values
        for j in range(target, num - 1, -1):
            dp[j] += dp[j - num]
```

```python
    return dp[target]

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

Coin Change 2 (DP - 22)

```python
def change(amount, coins):
    dp = [0] * (amount + 1)
    dp[0] = 1

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]

    return dp[amount]

# Example usage:
coins = [1, 2, 5]
amount = 5
print(change(amount, coins))  # Output: 4
```

Unbounded Knapsack (DP - 23)

```python
def unbounded_knapsack(W, weights, values):
    n = len(weights)
    dp = [0] * (W + 1)

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

    return dp[W]

# Example usage:
weights = [2, 3, 4]
values = [6, 8, 12]
W = 8
print(unbounded_knapsack(W, weights, values))  # Output: 24
```

Rod Cutting Problem | (DP - 24)

```python
def rod_cutting(lengths, prices, n):
    dp = [0] * (n + 1)

    for i in range(1, n + 1):
        max_revenue = 0
        for j in range(1, i + 1):
            max_revenue = max(max_revenue, prices[j - 1] + dp[i - j])
        dp[i] = max_revenue

    return dp[n]

# Example usage:
lengths = [1, 2, 3, 4, 5, 6, 7, 8]
prices = [1, 5, 8, 9, 10, 17, 17, 20]
n = 8
print(rod_cutting(lengths, prices, n))  # Output: 22
```

Longest Common Subsequence | (DP - 25)

```python
def longest_common_subsequence(text1, text2):
    m, n = len(text1), 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
```

Print Longest Common Subsequence | (DP - 26)

```python
def print_lcs(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    direction = [[''] * (n + 1) for _ in range(m + 1)]

    # Build dp and direction tables
    for i in range(1, m + 1):
```

```python
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                direction[i][j] = 'diag'
            else:
                if dp[i - 1][j] >= dp[i][j - 1]:
                    dp[i][j] = dp[i - 1][j]
                    direction[i][j] = 'up'
                else:
                    dp[i][j] = dp[i][j - 1]
                    direction[i][j] = 'left'

    # Reconstruct the LCS
    lcs = []
    i, j = m, n
    while i > 0 and j > 0:
        if direction[i][j] == 'diag':
            lcs.append(text1[i - 1])
            i -= 1
            j -= 1
        elif direction[i][j] == 'up':
            i -= 1
        else:  # direction[i][j] == 'left'
            j -= 1

    return ''.join(reversed(lcs))

# Example usage:
text1 = "abcde"
text2 = "ace"
print(print_lcs(text1, text2))  # Output: "ace"
```

Longest Common Substring | (DP - 27)

```python
def longest_common_substring(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    max_length = 0

    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
                max_length = max(max_length, dp[i][j])
            else:
                dp[i][j] = 0
```

```python
    return max_length

# Example usage:
text1 = "abcde"
text2 = "bcdf"
print(longest_common_substring(text1, text2))  # Output: 2
```

Longest Palindromic Subsequence | (DP-28)

```python
def longest_palindromic_subsequence(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

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

    # Build 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: 5
```

Minimum insertions to make string palindrome | DP-29

```python
def min_insertions_to_make_palindrome(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

    for length in range(2, n+1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                dp[i][j] = dp[i+1][j-1]
            else:
                dp[i][j] = min(dp[i+1][j], dp[i][j-1]) + 1
```

```
    return dp[0][n-1]

# Example usage:
s = "abcde"
print(min_insertions_to_make_palindrome(s))  # Output: 4 (abcde --> abcdcba)
```

Minimum Insertions/Deletions to Convert String | (DP- 30)

```
def min_operations_to_convert(s1, s2):
    m = len(s1)
    n = len(s2)

    # Create a (m+1) x (n+1) dp table initialized with zeros
    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 s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1])

    # Minimum operations to convert s1 to s2
    min_operations = dp[m][n]

    return min_operations

# Example usage:
s1 = "heap"
s2 = "pea"
print("Minimum operations to convert '{}' to '{}':".format(s1, s2),
min_operations_to_convert(s1, s2))  # Output: 3 (Insert 'p', Delete 'h' and
'p')
```

Shortest Common Supersequence | (DP - 31)

```
def shortest_common_supersequence(s1, s2):
    m = len(s1)
    n = len(s2)

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

    # Step 2: Fill the DP table
    for i in range(1, m + 1):
```

```python
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1)

    # Step 3: Build the shortest common supersequence
    i, j = m, n
    scs = []

    while i > 0 and j > 0:
        if s1[i - 1] == s2[j - 1]:
            scs.append(s1[i - 1])
            i -= 1
            j -= 1
        elif dp[i - 1][j] < dp[i][j - 1]:
            scs.append(s1[i - 1])
            i -= 1
        else:
            scs.append(s2[j - 1])
            j -= 1

    while i > 0:
        scs.append(s1[i - 1])
        i -= 1

    while j > 0:
        scs.append(s2[j - 1])
        j -= 1

    # Reverse the collected characters to get the correct order
    scs.reverse()

    # Convert list of characters to string
    return ''.join(scs)

# Example usage:
s1 = "abac"
s2 = "cab"
print("Shortest Common Supersequence of '{}' and '{}':".format(s1, s2),
shortest_common_supersequence(s1, s2))  # Output: "cabac"
```

Distinct Subsequences| (DP-32)

```python
def distinct_subsequences(s):
    n = len(s)
    dp = [0] * (n + 1)
```

```
    dp[0] = 1  # Base case: There is one way to form an empty subsequence

    last_occurrence = {}

    for i in range(1, n + 1):
        dp[i] = dp[i - 1]  # exclude s[i - 1] case

        if s[i - 1] in last_occurrence:
            dp[i] += dp[last_occurrence[s[i - 1]]]

        last_occurrence[s[i - 1]] = i - 1

    return dp[n]

# Example usage:
s = "abc"
print("Number of distinct subsequences in '{}':".format(s),
distinct_subsequences(s))  # Output: 7
```

Edit Distance | (DP-33)

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

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

    # Step 2: Fill the DP table
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
              dp[i][j] = j  # If word1 is empty, only option is to insert
all characters   of word2
            elif j == 0:
              dp[i][j] = i  # If word2 is empty, only option is to delete
all characters   of word1
            elif word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # No operation needed
            else:
                dp[i][j] = 1 + min(dp[i][j - 1],  # Insertion
                                   dp[i - 1][j],  # Deletion
                                   dp[i - 1][j - 1])  # Substitution

    # Step 3: Return the minimum edit distance
    return dp[m][n]
```

```
  # Example usage:
  word1 = "horse"
  word2 = "ros"
  print("Minimum edit distance between '{}' and '{}':".format(word1, word2),
min_distance  (word1, word2))  # Output: 3
```

Wildcard Matching | (DP-34)

```
  def isMatch(s, p):
      m = len(s)
      n = len(p)

      # Step 1: Create a DP table and initialize with False
      dp = [[False] * (n + 1) for _ in range(m + 1)]

      # Step 2: Initialize base cases
      dp[0][0] = True  # Empty string and empty pattern match

      for j in range(1, n + 1):
          if p[j - 1] == '*':
              dp[0][j] = dp[0][j - 1]  # Empty string matches if pattern is
'*'

      # Step 3: Fill the DP table
      for i in range(1, m + 1):
          for j in range(1, n + 1):
              if s[i - 1] == p[j - 1] or p[j - 1] == '?':
                  dp[i][j] = dp[i - 1][j - 1]
              elif p[j - 1] == '*':
                  dp[i][j] = dp[i - 1][j] or dp[i][j - 1]

      # Step 4: Return the result
      return dp[m][n]

  # Example usage:
  s = "adceb"
  p = "*a*b"
  print("Does '{}' match '{}'?: {}".format(s, p, isMatch(s, p)))  # Output:
True
```

Best Time to Buy and Sell Stock |(DP-35)

```
  def maxProfit(prices):
      if not prices:
          return 0
```

```python
    min_price = float('inf')
    max_profit = 0

    for price in prices:
        min_price = min(min_price, price)
        max_profit = max(max_profit, price - min_price)

    return max_profit

# Example usage:
prices = [7, 1, 5, 3, 6, 4]
print("Maximum profit:", maxProfit(prices))  # Output: 5 (Buy at 1 and sell at 6)
```

Buy and Sell Stock - II|(DP-36)

```python
def maxProfit(prices):
    total_profit = 0
    n = len(prices)

    for i in range(1, n):
        if prices[i] > prices[i - 1]:
            total_profit += prices[i] - prices[i - 1]

    return total_profit

# Example usage:
prices = [7, 1, 5, 3, 6, 4]
print("Maximum profit:", maxProfit(prices))  # Output: 7 (Buy at 1, sell at 5, buy at 3, sell at 6)
```

Buy and Sell Stocks III|(DP-37)

```python
def maxProfit(prices):
    if not prices:
        return 0

    n = len(prices)

    # Initialize DP arrays
    buy1 = [-float('inf')] * n
    sell1 = [0] * n
    buy2 = [-float('inf')] * n
    sell2 = [0] * n

    # Base case initialization
```

```
    buy1[0] = -prices[0]
    buy2[0] = -prices[0]

    # Fill DP arrays
    for i in range(1, n):
        buy1[i] = max(buy1[i-1], -prices[i])
        sell1[i] = max(sell1[i-1], buy1[i-1] + prices[i])
        buy2[i] = max(buy2[i-1], sell1[i-1] - prices[i])
        sell2[i] = max(sell2[i-1], buy2[i-1] + prices[i])

    # Return the maximum profit after the second sell
    return sell2[-1]

# Example usage:
prices = [3, 3, 5, 0, 0, 3, 1, 4]
print("Maximum profit with at most two transactions:", maxProfit(prices))  #
Output: 6 (Buy on day 4 and sell on day 6, buy on day 7 and sell on day 8)
```

Buy and Stock Sell IV |(DP-38)

```
def maxProfit(k, prices):
    if not prices or k == 0:
        return 0

    n = len(prices)

    # If k >= n/2, then we can perform as many transactions as we want
    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

    # Initialize DP arrays
    buy = [-float('inf')] * (k + 1)
    sell = [0] * (k + 1)

    # Fill DP arrays
    for price in prices:
        for j in range(1, k + 1):
            buy[j] = max(buy[j], sell[j - 1] - price)
            sell[j] = max(sell[j], buy[j] + price)

    # Return the maximum profit after k transactions
    return sell[k]
```

```
  # Example usage:
  k = 2
  prices = [3, 2, 6, 5, 0, 3]
  print("Maximum profit with at most", k, "transactions:", maxProfit(k,
prices))  # Output: 7 (Buy on day 2 and sell on day 3, buy on day 5 and sell
on day 6)
```

Buy and Sell Stocks With Cooldown | (DP-39)

```
   def maxProfit(prices):
       if not prices:
           return 0

       n = len(prices)

       # Initialize DP arrays
       buy = [-float('inf')] * n
       sell = [0] * n
       cooldown = [0] * n

       # Base case initialization
       buy[0] = -prices[0]

       # Fill DP arrays
       for i in range(1, n):
           buy[i] = max(cooldown[i - 1] - prices[i], buy[i - 1])
           sell[i] = max(buy[i - 1] + prices[i], sell[i - 1])
           cooldown[i] = max(sell[i - 1], cooldown[i - 1])

       # Return the maximum profit at the end of the last day
       return max(sell[-1], cooldown[-1])

   # Example usage:
   prices = [1, 2, 3, 0, 2]
   print("Maximum profit with cooldown:", maxProfit(prices))  # Output: 3 (Buy
on day 1, sell on day 2, buy on day 4, sell on day 5)
```

Buy and Sell Stocks With Transaction Fee | (DP-40)

```
  def maxProfit(prices, fee):
      if not prices:
          return 0

      n = len(prices)

      # Initialize DP arrays
```

```
    buy = [-float('inf')] * n
    sell = [0] * n

    # Base case initialization
    buy[0] = -prices[0]

    # Fill DP arrays
    for i in range(1, n):
        buy[i] = max(buy[i - 1], sell[i - 1] - prices[i])
        sell[i] = max(sell[i - 1], buy[i - 1] + prices[i] - fee)

    # Return the maximum profit after the last day
    return sell[-1]

# Example usage:
prices = [1, 3, 2, 8, 4, 9]
fee = 2
print("Maximum profit with transaction fee:", maxProfit(prices, fee))  #
Output: 8 (Buy on day 1, sell on day 4, buy on day 5, sell on day 6)
```

Longest Increasing Subsequence |(DP-41)

```
def lengthOfLIS(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 longest increasing subsequence:", lengthOfLIS(nums))  #
Output: 4 (The LIS is [2, 3, 7, 101])
```

Printing Longest Increasing Subsequence|(DP-42)

```
def printLIS(nums):
    if not nums:
        return
```

```python
    n = len(nums)
    dp = [1] * n
    prev_index = [-1] * n

    # Find LIS length and populate prev_index
    max_length = 1
    max_length_index = 0

    for i in range(1, n):
        for j in range(i):
            if nums[j] < nums[i] and dp[i] < dp[j] + 1:
                dp[i] = dp[j] + 1
                prev_index[i] = j
                if dp[i] > max_length:
                    max_length = dp[i]
                    max_length_index = i

    # Construct LIS from prev_index
    lis_sequence = []
    idx = max_length_index
    while idx != -1:
        lis_sequence.append(nums[idx])
        idx = prev_index[idx]

    # Reverse the lis_sequence to get the correct order
    lis_sequence.reverse()

    # Print the LIS
    print("Longest Increasing Subsequence:", lis_sequence)

# Example usage:
nums = [10, 9, 2, 5, 3, 7, 101, 18]
printLIS(nums)  # Output: [2, 3, 7, 101]
```

Longest Increasing Subsequence |(DP-43)

```python
def lengthOfLIS(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 longest increasing subsequence:", lengthOfLIS(nums))  # Output: 4 (The LIS is [2, 3, 7, 101])
```

Largest Divisible Subset|(DP-44)

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

    nums.sort()
    n = len(nums)
    dp = [1] * n
    prev_index = [-1] * n

    max_len = 1
    max_idx = 0

    # Fill dp array
    for i in range(1, n):
        for j in range(i):
            if nums[i] % nums[j] == 0 and dp[i] < dp[j] + 1:
                dp[i] = dp[j] + 1
                prev_index[i] = j
                if dp[i] > max_len:
                    max_len = dp[i]
                    max_idx = i

    # Construct largest divisible subset
    result = []
    idx = max_idx
    while idx != -1:
        result.append(nums[idx])
        idx = prev_index[idx]

    return result

# Example usage:
nums = [1, 2, 3, 4, 6, 12]
print("Largest divisible subset:", largestDivisibleSubset(nums))  # Output: [1, 2, 4, 12]
```

Longest String Chain|(DP-45)

```python
def longestStrChain(words):
    if not words:
        return 0

    # Sort words by length
    words.sort(key=len)

    # Dictionary to store longest chain ending with each word
    dp = {}
    prev_word_map = {}
    max_chain_length = 1

    # Iterate over each word
    for word in words:
        dp[word] = 1  # Minimum chain length is 1

        # Generate all possible predecessors by removing one character
        for i in range(len(word)):
            predecessor = word[:i] + word[i+1:]
            if predecessor in dp:
                if dp[word] < dp[predecessor] + 1:
                    dp[word] = dp[predecessor] + 1
                    prev_word_map[word] = predecessor

        # Update maximum chain length found so far
        max_chain_length = max(max_chain_length, dp[word])

    # Backtrack to find the longest chain
    longest_chain = []
    current_word = max(dp, key=dp.get)
    while current_word in prev_word_map:
        longest_chain.append(current_word)
        current_word = prev_word_map[current_word]
    longest_chain.append(current_word)

    # Reverse the longest_chain to get the correct order
    longest_chain.reverse()

    return longest_chain

# Example usage:
words = ["a", "b", "ba", "bca", "bda", "bdca"]
print("Longest string chain:", longestStrChain(words))  # Output: ['a',
'ba', 'bca', 'bdca']
```

Longest Bitonic Subsequence |(DP-46)

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

    n = len(nums)
    lis = [1] * n
    lds = [1] * n

    # Compute LIS from left to right
    for i in range(1, n):
        for j in range(i):
            if nums[j] < nums[i]:
                lis[i] = max(lis[i], lis[j] + 1)

    # Compute LDS from right to left
    for i in range(n - 2, -1, -1):
        for j in range(i + 1, n):
            if nums[j] < nums[i]:
                lds[i] = max(lds[i], lds[j] + 1)

    # Compute LBS for each index
    lbs = [0] * n
    for i in range(n):
        lbs[i] = lis[i] + lds[i] - 1

    # Return the maximum length of LBS
    return max(lbs)

# Example usage:
nums = [4, 2, 3, 6, 10, 1, 12]
print("Length of longest bitonic subsequence:",
longestBitonicSubsequence(nums))  # Output: 5 (One possible LBS is [2, 3, 6,
10, 1])
```

Number of Longest Increasing Subsequences|(DP-47)

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

    n = len(nums)
    if n == 1:
        return 1

    # Arrays to store length of LIS and count of LIS ending at each index
```

```python
    length = [1] * n
    count = [1] * n

    # Compute LIS length and count
    for i in range(1, n):
        for j in range(i):
            if nums[j] < nums[i]:
                if length[i] < length[j] + 1:
                    length[i] = length[j] + 1
                    count[i] = count[j]
                elif length[i] == length[j] + 1:
                    count[i] += count[j]

    # Find the maximum length of LIS
    max_length = max(length)
    result = 0

    # Count the number of LIS with maximum length
    for i in range(n):
        if length[i] == max_length:
            result += count[i]

    return result

# Example usage:
nums = [1, 3, 5, 4, 7]
print("Number of longest increasing subsequences:", findNumberOfLIS(nums))
# Output: 2
```

Matrix Chain Multiplication|(DP-48)

```python
def matrixChainOrder(p):
    n = len(p) - 1  # number of matrices in the chain
    dp = [[0] * (n + 1) for _ in range(n + 1)]

    # Initialize dp table
    for length in range(2, n + 1):  # Length of the chain
        for i in range(1, n - length + 2):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k+1][j] + p[i-1] * p[k] * p[j]
                if cost < dp[i][j]:
                    dp[i][j] = cost

    return dp[1][n]
```

```
  # Example usage:
  matrix_dimensions = [30, 35, 15, 5, 10, 20, 25]
  min_scalar_multiplications = matrixChainOrder(matrix_dimensions)
  print("Minimum number of scalar multiplications:",
min_scalar_multiplications)
```

Matrix Chain Multiplication | Bottom-Up|(DP-49)

```
  def matrixChainOrder(p):
      n = len(p) - 1  # number of matrices in the chain
      dp = [[0] * (n + 1) for _ in range(n + 1)]

      # Initialize dp table
      for length in range(2, n + 1):  # Length of the chain
          for i in range(1, n - length + 2):
              j = i + length - 1
              dp[i][j] = float('inf')
              for k in range(i, j):
                  cost = dp[i][k] + dp[k+1][j] + p[i-1] * p[k] * p[j]
                  if cost < dp[i][j]:
                      dp[i][j] = cost

      return dp[1][n]

  # Example usage:
  matrix_dimensions = [30, 35, 15, 5, 10, 20, 25]
  min_scalar_multiplications = matrixChainOrder(matrix_dimensions)
  print("Minimum number of scalar multiplications:",
min_scalar_multiplications)
```

Minimum Cost to Cut the Stick|(DP-50)

```
  def minCost(n, cuts):
      cuts = [0] + sorted(cuts) + [n]
      m = len(cuts)
      dp = [[0] * m for _ in range(m)]

      for length in range(2, m):
          for l in range(m - length):
              r = l + length
              dp[l][r] = float('inf')
              for k in range(l + 1, r):
                  dp[l][r] = min(dp[l][r], dp[l][k] + dp[k][r] + cuts[r] -
cuts[l])
```

```python
    return dp[0][m-1]

# Example usage:
n = 7
cuts = [1, 3, 4, 5]
print("Minimum cost to cut the stick:", minCost(n, cuts))  # Output: 16
```

Burst Balloons|(DP-51)

```python
def maxCoins(nums):
    nums = [1] + nums + [1]  # Pad nums with 1s on both ends
    n = len(nums)
    dp = [[0] * n for _ in range(n)]

    for length in range(2, n):
        for i in range(n - length):
            j = i + length
            for k in range(i + 1, j):
                dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[i] *
nums[k] * nums[j])

    return dp[0][n-1]

# Example usage:
nums = [3, 1, 5, 8]
print("Maximum coins obtained:", maxCoins(nums))  # Output: 167
```

Evaluate Boolean Expression to True|(DP-52)

```python
def countEval(expression):
    n = len(expression)

    # Step 1: Initialize dp tables
    dpT = [[0] * n for _ in range(n)]
    dpF = [[0] * n for _ in range(n)]

    # Step 2: Initialize base cases
    for i in range(0, n, 2):
        if expression[i] == 'T':
            dpT[i][i] = 1
        else:
            dpF[i][i] = 1

    # Step 3: Fill the dp tables
    for length in range(2, n + 1, 2):  # length of subexpression
```

```
        for i in range(0, n - length + 1, 2):  # starting index of
subexpression
            j = i + length - 1  # ending index of subexpression
            for k in range(i + 1, j, 2):  # operator index
                op = expression[k]

                if op == '&':
                    dpT[i][j] += dpT[i][k - 1] * dpT[k + 1][j]
                    dpF[i][j] += dpF[i][k - 1] * dpF[k + 1][j]
                    dpF[i][j] += dpT[i][k - 1] * dpF[k + 1][j]
                    dpF[i][j] += dpF[i][k - 1] * dpT[k + 1][j]
                elif op == '|':
                    dpT[i][j] += dpT[i][k - 1] * dpT[k + 1][j]
                    dpT[i][j] += dpT[i][k - 1] * dpF[k + 1][j]
                    dpT[i][j] += dpF[i][k - 1] * dpT[k + 1][j]
                    dpF[i][j] += dpF[i][k - 1] * dpF[k + 1][j]
                elif op == '^':
                    dpT[i][j] += dpT[i][k - 1] * dpF[k + 1][j]
                    dpT[i][j] += dpF[i][k - 1] * dpT[k + 1][j]
                    dpF[i][j] += dpT[i][k - 1] * dpT[k + 1][j]
                    dpF[i][j] += dpF[i][k - 1] * dpF[k + 1][j]

    # Step 4: Return the result
    return dpT[0][n - 1]

# Example usage:
expression = "T^F|F"
print("Number of ways to evaluate to true:", countEval(expression))  #
Output: 2
```

Palindrome Partitioning - II|(DP-53)

```
def minCut(s):
    n = len(s)
    if n == 0:
        return 0

    # Step 1: Initialize dp and isPalindrome arrays
    dp = [0] * (n + 1)
    isPalindrome = [[False] * n for _ in range(n)]

    # Step 2: Calculate isPalindrome array
    for i in range(n):
        isPalindrome[i][i] = True

    for length in range(2, n + 1):
        for i in range(n - length + 1):
```

```
            j = i + length - 1
            if length == 2:
                isPalindrome[i][j] = (s[i] == s[j])
            else:
                isPalindrome[i][j] = (s[i] == s[j]) and isPalindrome[i +
1][j - 1]

    # Step 3: Calculate dp array
    for j in range(1, n + 1):
        dp[j] = j - 1  # Maximum cuts for worst case scenario
        for i in range(j):
            if isPalindrome[i][j - 1]:
                dp[j] = min(dp[j], dp[i] + 1)

    return dp[n]

# Example usage:
s = "aab"
print("Minimum cuts needed:", minCut(s))  # Output: 1
```

Partition Array for Maximum Sum|(DP-54)

```
def maxSumAfterPartitioning(arr, k):
    n = len(arr)
    dp = [0] * (n + 1)

    for i in range(1, n + 1):
        max_element = arr[i - 1]
        for j in range(1, min(i, k) + 1):
            max_element = max(max_element, arr[i - j])
            dp[i] = max(dp[i], dp[i - j] + max_element * j)

    return dp[n]

# Example usage:
arr = [1, 15, 7, 9, 2, 5, 10]
k = 3
print("Maximum sum after partitioning:", maxSumAfterPartitioning(arr, k))  #
Output: 84
```

Maximum Rectangle Area with all 1's|(DP-55)

```
def maximalRectangle(matrix):
    if not matrix:
        return 0
```

```python
    m, n = len(matrix), len(matrix[0])
    max_area = 0
    height = [0] * n

    for i in range(m):
        # Update height array based on current row of the matrix
        for j in range(n):
            if matrix[i][j] == '1':
                height[j] += 1
            else:
                height[j] = 0

        # Calculate maximum rectangle area using histogram approach
        stack = []
        for j in range(n + 1):
            while stack and (j == n or height[stack[-1]] > height[j]):
                h = height[stack.pop()]
                width = j if not stack else j - stack[-1] - 1
                max_area = max(max_area, h * width)
            stack.append(j)

    return max_area

# Example usage:
matrix = [
    ["1","0","1","0","0"],
    ["1","0","1","1","1"],
    ["1","1","1","1","1"],
    ["1","0","0","1","0"]
]
print("Maximum rectangle area with all 1's:", maximalRectangle(matrix))  #
Output: 6
```

Count Square Submatrices with All Ones|(DP-56)

```python
def countSquares(matrix):
    m = len(matrix)
    n = len(matrix[0])
    dp = [[0] * n for _ in range(m)]
    count = 0

    # Initialize dp array and count for single cell squares
    for i in range(m):
        for j in range(n):
            dp[i][j] = matrix[i][j]
            count += dp[i][j]
```

```python
    # Fill dp array
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 1:
                dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
                count += dp[i][j]

    return count

# Example usage:
matrix = [
    [0, 1, 1, 1],
    [1, 1, 1, 1],
    [0, 1, 1, 1]
]
print("Number of square submatrices with all ones:", countSquares(matrix))
# Output: 15
```

Tries

Implement Trie - 2 (Prefix Tree)

```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):
        current = self.root
        for char in word:
            if char not in current.children:
                current.children[char] = TrieNode()
            current = current.children[char]
        current.is_end_of_word = True

    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
```

```python
    def startsWith(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 object
trie = Trie()

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

# Search for words in the Trie
print(trie.search("apple"))  # Output: True
print(trie.search("app"))    # Output: True
print(trie.search("banana")) # Output: True
print(trie.search("orange")) # Output: False

# Check if prefix exists in the Trie
print(trie.startsWith("app"))  # Output: True
print(trie.startsWith("ban"))  # Output: True
print(trie.startsWith("or"))   # Output: False
```

Implement Trie - 2 (PrefiLongest String with All Prefixesx Tree)

```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):
        current = self.root
        for char in word:
            if char not in current.children:
                current.children[char] = TrieNode()
            current = current.children[char]
            if current.is_end_of_word:
                return False  # Indicates there was a repeated prefix
        current.is_end_of_word = True
```

```
        return True

def longestStringWithAllPrefixes(strings):
    trie = Trie()
    longest = ""

    for s in strings:
        if not trie.insert(s):
            break
        longest = s

    return longest

# Example usage:
strings = ["abc", "ab", "abcd", "abcde", "abcdef"]
print("Longest string with all unique prefixes:",
longestStringWithAllPrefixes(strings))  # Output: "abcd"
```

Number of Distinct Substrings in a String

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

    # Function to compute LCP array
    def computeLCP(s, suffix_array):
        rank = [0] * n
        lcp = [0] * n
        for i, suffix in enumerate(suffix_array):
            rank[suffix] = 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

    lcp = computeLCP(s, suffix_array)
    distinct_substrings = 0

    # Calculate number of distinct substrings
    for i in range(1, n):
```

```
        distinct_substrings += (n - suffix_array[i] - lcp[i])

    distinct_substrings += n  # Adding the single character substrings

    return distinct_substrings

# Example usage:
s = "banana"
print("Number of distinct substrings:", countDistinctSubstrings(s))  # Output: 15
```

Bit PreRequisites for TRIE Problems

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.bitmask = 0  # Example of using bitmask for Trie node

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        current = self.root
        for char in word:
            idx = ord(char) - ord('a')
            if idx not in current.children:
                current.children[idx] = TrieNode()
            current = current.children[idx]
            current.bitmask |= (1 << idx)  # Setting bitmask for current node

    def search(self, word):
        current = self.root
        for char in word:
            idx = ord(char) - ord('a')
            if idx not in current.children:
                return False
            current = current.children[idx]
        return current.is_end_of_word
```

Maximum XOR of two numbers in an array

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

```python
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, num):
        current = self.root
        for i in range(31, -1, -1):  # considering 32-bit integers
            bit = (num >> i) & 1
            if bit not in current.children:
                current.children[bit] = TrieNode()
            current = current.children[bit]

    def findMaxXOR(self, num):
        current = self.root
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            opposite_bit = 1 - bit

            if opposite_bit in current.children:
                max_xor |= (1 << i)
                current = current.children[opposite_bit]
            else:
                current = current.children[bit]

        return max_xor

def maximumXOR(nums):
    trie = Trie()
    max_xor = 0

    for num in nums:
        trie.insert(num)
        max_xor = max(max_xor, trie.findMaxXOR(num))

    return max_xor

# Example usage:
nums = [3, 10, 5, 25, 2, 8]
print("Maximum XOR of two numbers in the array:", maximumXOR(nums))  #
Output: 28
```

Maximum XOR With an Element From Array

```python
class TrieNode:
    def __init__(self):
```

```python
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, num):
        current = self.root
        for i in range(31, -1, -1):  # considering 32-bit integers
            bit = (num >> i) & 1
            if bit not in current.children:
                current.children[bit] = TrieNode()
            current = current.children[bit]

    def findMaxXOR(self, num):
        current = self.root
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            opposite_bit = 1 - bit

            if opposite_bit in current.children:
                max_xor |= (1 << i)
                current = current.children[opposite_bit]
            else:
                current = current.children[bit]

        return max_xor

def maximizeXor(nums, queries):
    trie = Trie()
    result = [-1] * len(queries)
    nums.sort()
    queries = sorted((x, m, idx) for idx, (x, m) in enumerate(queries))

    j = 0
    for x, m, idx in queries:
        while j < len(nums) and nums[j] <= m:
            trie.insert(nums[j])
            j += 1
        if j > 0:
            result[idx] = trie.findMaxXOR(x)

    return result

# Example usage:
nums = [3, 10, 5, 25, 2, 8]
queries = [[0, 1], [1, 2], [4, 1], [5, 2], [7, 3]]
```

```python
  print("Results of maximum XOR with an element in nums:", maximizeXor(nums, queries))
```

Minimum number of bracket reversals needed to make an expression balanced

```python
def min_reversals_to_balance(expression):
    n = len(expression)
    if n % 2 != 0:
        return -1  # If length of expression is odd, cannot 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)

    # Now stack contains the unmatched brackets
    m = stack.count('(')
    p = stack.count(')')

    # Calculate minimum reversals needed
    return (m // 2) + (p // 2)

# Example usage:
expression = "}}}}}}{{{{"
print(min_reversals_to_balance(expression))  # Output: 4
```

Count and say

```python
def count_and_say(n):
    if n <= 0:
        return ""
    result = "1"  # Base case
    for _ in range(1, n):
        count = 1
        current_char = result[0]
        next_term = []
        for i in range(1, len(result)):
            if result[i] == current_char:
                count += 1
            else:
```

```python
                next_term.append(str(count))
                next_term.append(current_char)
                current_char = result[i]
                count = 1
        next_term.append(str(count))
        next_term.append(current_char)
        result = ''.join(next_term)
    return result

# Example usage:
n = 5
print(count_and_say(n))  # Output: "111221"
```

Rabin Karp

```python
class RabinKarp:
    def search(self, text, pattern):
        n = len(text)
        m = len(pattern)
        if n < m:
            return -1

        # Constants for hashing
        base = 256  # Number of possible characters (ASCII)
        modulus = 10**9 + 7  # A large prime number for modulus operation

        # Compute hash of pattern and first window of text
        pattern_hash = 0
        text_hash = 0
        base_power = 1

        for i in range(m):
            pattern_hash = (pattern_hash * base + ord(pattern[i])) % modulus
            text_hash = (text_hash * base + ord(text[i])) % modulus
            if i < m - 1:
                base_power = (base_power * base) % modulus

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

            # Update the rolling hash for the next window
            if i < n - m:
                text_hash = (text_hash - ord(text[i]) * base_power) %
modulus
```

```
                    text_hash = (text_hash * base + ord(text[i + m])) % modulus
                    text_hash = (text_hash + modulus) % modulus  # Ensure non-
negative

        return -1  # Pattern not found

# Example usage:
rk = RabinKarp()
text = "AABAACAADAABAABA"
pattern = "AABA"
print(rk.search(text, pattern))  # Output: 0 (pattern found starting at
index 0)
```

Z-Function

```
def compute_z_function(s):
    n = len(s)
    Z = [0] * n
    l, r, k = 0, 0, 0

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

    return Z

# Example usage:
s = "aabcaabxaaaz"
Z = compute_z_function(s)
print(Z)  # Output: [0, 1, 0, 0, 3, 1, 0, 0, 2, 2, 1, 0]
```

KMP algo / LPS(pi) array

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

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
```

```python
                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, j = 0, 0
    indices = []

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

        if j == m:
            indices.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 indices

# Example usage:
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
indices = kmp_search(text, pattern)
print("Pattern found at indices:", indices)  # Output: [10]
```

Shortest Palindrome

```python
def shortest_palindrome(s):
    if not s:
        return ""

    # Function to compute LPS array using KMP algorithm
```

```python
    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

    # Reverse the string and find the LPP
    rev_s = s[::-1]
    combined = s + "#" + rev_s
    lps = compute_lps(combined)
    lpp_length = lps[-1]  # LPP length from the combined string

    # Shortest palindrome construction
    prefix_to_add = rev_s[:len(s) - lpp_length]
    shortest_palindrome = prefix_to_add + s

    return shortest_palindrome

# Example usage:
s = "aacecaaa"
print(shortest_palindrome(s))  # Output: "aaacecaaa"
```

Longest happy prefix

```python
def longest_happy_prefix(s):
    n = len(s)
    if n == 0:
        return ""

    # Compute LPS array using KMP algorithm
    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

    # Compute LPS array for the string itself
    lps = compute_lps(s)

    # Length of the longest prefix which is also suffix
    length_of_prefix = lps[-1]

    # Extract the longest happy prefix
    longest_prefix = s[:length_of_prefix]

    return longest_prefix

# Example usage:
s = "ababab"
print(longest_happy_prefix(s))  # Output: "abab"
```

Count palindromic subsequence in given string

```
def count_palindromic_subsequences(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

    for i in range(n):
        dp[i][i] = 1  # Single character is a palindrome

    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] + dp[i][j - 1] + 1
            else:
```

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

    return dp[0][n - 1]

# Example usage:
s = "abcb"
print(count_palindromic_subsequences(s))  # Output: 6
```