

## Advance Algorithmic & Problem Solving

**WEEK NO. : 1**  
**DATE:** 26-02-2024

**STUDENT NAME :Shristi Yadav**  
**NO. :**

### PROBLEM NO. :1

**AIM :** To find missing number in array, Rotate Bits

### INTRODUCTION:

To find missing number in array

This **PROGRAM** calculates the missing number in an array of  $n$  distinct numbers ranging from  $0$  to  $n$  by comparing the sum of the first  $n$  natural numbers to the sum of the elements in the array.

### SOURCE CODE

```
def missing_number(nums):  
    n = len(nums)  
    expected_sum = n * (n + 1) // 2  
    actual_sum = sum(nums)  
    return expected_sum - actual_sum
```

```
# Example usage:  
nums = [3, 0, 1]  
print(missing_number(nums))
```

### OUTPUT

```
2
```

```
=== Code Execution Successful ===
```

### To rotate bits in array

These functions perform bitwise rotations on an integer. The **left\_rotate** function performs a left circular shift, while the **right\_rotate** function performs a right circular shift, both within a specified bit width (default is 32 bits).

### SOURCE CODE

```
def left_rotate(n, d, bits=32):  
    return ((n << d) | (n >> (bits - d))) & ((1 << bits) - 1)  
  
def right_rotate(n, d, bits=32):  
    return ((n >> d) | (n << (bits - d))) & ((1 << bits) - 1)  
  
# Example usage:  
n = 16  
d = 2  
  
left_rotated = left_rotate(n, d)  
right_rotated = right_rotate(n, d)  
  
print(f'Left rotated: {left_rotated}')  
print(f'Right rotated: {right_rotated}')
```

### OUTPUT

```
Left rotated: 64  
Right rotated: 4
```

```
=== Code Execution Successful ===
```

**WEEK NO. : 2**

**STUDENT NAME :Shristi Yadav**

**DATE:3/03/24**

**NO. :**

### **PROGRAM NO:2**

**AIM:** To find if a number is power of 2 or not,find bit difference

#### **INTRODUCTION:**

##### **Power of 2 Check**

This **PROGRAM** checks if a given number is a power of 2 by using a bitwise operation that confirms if the number has exactly one bit set in its binary representation.

#### **SOURCE CODE**

```
def is_power_of_2(n):  
    return n > 0 and (n & (n - 1)) == 0
```

```
n = 16  
print(is_power_of_2(n))
```

#### **OUTPUT**

```
True
```

```
=== Code Execution Successful ===
```

## Bit Difference

This **PROGRAM** calculates the number of differing bits between two integers by using the XOR operation followed by counting the number of set bits in the result.

## SOURCE CODE

```
def bit_difference(a, b):  
    xor = a ^ b  
    count = 0  
    while xor:  
        count += xor & 1  
        xor >>= 1  
    return count
```

```
a = 29  
b = 15  
print(bit_difference(a, b))
```

## OUTPUT

```
2
```

```
=== Code Execution Successful ===
```

**WEEK NO. : 2**

**STUDENT NAME :Shristi Yadav**

**DATE: 4/03/24**

**NO. :**

### **PROGRAM NO:3**

**AIM:** To detect Duplicates in Array, Longest Consecutive 1's (Hamming Weight)

#### **INTRODUCTION**

##### **Detect Duplicates in Array**

This **PROGRAM** detects if there are any duplicates in a given array by using a set to track elements that have already been seen, returning **True** if a duplicate is found and **False** otherwise.

#### **SOURCE CODE**

```
def contains_duplicates(nums):
```

```
    seen = set()
```

```
    for num in nums:
```

```
        if num in seen:
```

```
            return True
```

```
        seen.add(num)
```

```
    return False
```

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

```
print(contains_duplicates(nums))
```

#### **OUTPUT**

```
True
```

```
=== Code Execution Successful ===
```

### Longest Consecutive 1's (Hamming Weight)

This **PROGRAM** calculates the length of the longest sequence of consecutive 1's in the binary representation of a given number using bitwise operations to check each bit.

### SOURCE CODE

```
def contains_duplicates(nums):
```

```
    seen = set()
```

```
    for num in nums:
```

```
        if num in seen:
```

```
            return True
```

```
        seen.add(num)
```

```
    return False
```

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

```
print(contains_duplicates(nums))
```

### OUTPUT

```
True
```

```
=== Code Execution Successful ===
```

**WEEK NO. : 3**

**STUDENT NAME :Shristi Yadav**

**DATE: 10/03/24**

**NO. :**

### **PROGRAM NO:4**

**AIM:** Finding Unique element in Array, Finding Maximum XOR Subarray

#### **INTRODUCTION:**

##### **Finding Unique Element in Array**

This **PROGRAM** finds the unique element in an array where every other element appears twice, by using the XOR operation which cancels out duplicate numbers, leaving the unique number.

#### **SOURCE CODE**

```
def find_unique_element(nums):
```

```
    unique = 0
```

```
    for num in nums:
```

```
        unique ^= num
```

```
    return unique
```

```
nums = [4, 1, 2, 1, 2]
```

```
print(find_unique_element(nums))
```

#### **OUTPUT**

```
4
```

```
=== Code Execution Successful ===
```

##### **Finding Maximum XOR Subarray**

This **PROGRAM** finds the maximum XOR value of any subarray within a given array by maintaining a prefix XOR and using a trie to efficiently compute the maximum possible XOR for each prefix.

#### **SOURCE CODE**

```
class TrieNode:
```

```
    def __init__(self):
```

```
        self.left = None
```

```
        self.right = None
```

```
class Trie:
```

```

def __init__(self):
    self.root = TrieNode()

def insert(self, num):
    node = self.root
    for i in range(31, -1, -1):
        bit = (num >> i) & 1
        if bit == 0:
            if not node.left:
                node.left = TrieNode()
            node = node.left
        else:
            if not node.right:
                node.right = TrieNode()
            node = node.right

def max_xor(self, num):
    node = self.root
    max_xor = 0
    for i in range(31, -1, -1):
        bit = (num >> i) & 1
        if bit == 0:
            if node.right:
                max_xor |= (1 << i)
                node = node.right
            else:
                node = node.left
        else:
            if node.left:
                max_xor |= (1 << i)
                node = node.left
            else:
                node = node.right
    return max_xor

def find_maximum_xor_subarray(nums):
    trie = Trie()
    max_xor = 0
    prefix_xor = 0
    trie.insert(0)
    for num in nums:
        prefix_xor ^= num
        max_xor = max(max_xor, trie.max_xor(prefix_xor))
        trie.insert(prefix_xor)
    return max_xor

nums = [3, 10, 5, 25, 2, 8]
print(find_maximum_xor_subarray(nums))

```



## OUTPUT

```
31
```

```
=== Code Execution Successful ===
```

**WEEK NO. :** 4

**STUDENT NAME :**Shristi Yadav

**DATE:** 11/03/24

**NO. :**

### **PROGRAM NO:5**

**AIM:** To find the Kth largest elements in an array, Rearrange an array in maximum minimum form using Two Pointer Technique

#### **INTRODUCTION**

#### **Find the Kth Largest Element in an Array**

This **PROGRAM** finds the Kth largest element in an array using the heap data structure to maintain the K largest elements encountered.

#### **SOURCE CODE**

```
import heapq

def find_kth_largest(nums, k):
    return heapq.nlargest(k, nums)[-1]

nums = [3, 2, 1, 5, 6, 4]
k = 2
print(find_kth_largest(nums, k))
```

#### **OUTPUT**

```
5

=== Code Execution Successful ===
```

## Rearrange an Array in Maximum Minimum Form Using Two Pointer Technique

This **PROGRAM** rearranges an array in a specific order where the largest element is followed by the smallest element, then the second largest by the second smallest, and so on, using the two-pointer technique.

### SOURCE CODE

```
def rearrange_max_min(arr):
    n = len(arr)
    result = [0] * n
    left, right = 0, n - 1
    flag = True

    for i in range(n):
        if flag:
            result[i] = arr[right]
            right -= 1
        else:
            result[i] = arr[left]
            left += 1
        flag = not flag

    for i in range(n):
        arr[i] = result[i]

arr = [1, 2, 3, 4, 5, 6, 7]
rearrange_max_min(arr)
print(arr)
```

### OUTPUT

```
[7, 1, 6, 2, 5, 3, 4]
```

```
=== Code Execution Successful ===
```

**WEEK NO. :** 5

**STUDENT NAME :**Shristi Yadav

**DATE:** 17/03/24

**NO. :**

### **PROGRAM NO:6**

**AIM:** Move all zeroes to end of array, Rearrange array such that even positioned are greater than odd

#### **INTRODUCTION**

1. Iterate through the array and move all zeroes to the end.
2. Sort the array such that even positioned elements are greater than odd positioned elements.

#### **SOURCE CODE**

```
def move_zeroes_to_end(arr):
    # Move all zeroes to the end of the array
    non_zero_index = 0
    for i in range(len(arr)):
        if arr[i] != 0:
            arr[non_zero_index], arr[i] = arr[i], arr[non_zero_index]
            non_zero_index += 1

def rearrange_even_odd(arr):
    arr.sort()
    for i in range(1, len(arr), 2):
        if i < len(arr) - 1:
            arr[i], arr[i + 1] = arr[i + 1], arr[i]

# Example usage:
arr = [0, 2, 3, 0, 5, 8, 0, 1]
move_zeroes_to_end(arr)
rearrange_even_odd(arr)
print(arr)
```

#### **OUTPUT**

```
[0, 0, 0, 2, 1, 5, 3, 8]
```

```
=== Code Execution Successful ===
```

**WEEK NO. : 5**

**STUDENT NAME :Shristi Yadav**

**DATE: 18/03/24**

**NO. :**

### **PROGRAM NO:7**

**AIM:** Find sub-array with given sum, Find the smallest missing number

### **INTRODUCTION**

#### **Find Sub-array with Given Sum:**

Using the sliding window technique to find a sub-array with a given sum

### **SOURCE CODE**

```
def find_subarray_with_sum(arr, target_sum):
    current_sum = arr[0]
    start = 0

    for end in range(1, len(arr)):
        while current_sum > target_sum and start < end:
            current_sum -= arr[start]
            start += 1

        if current_sum == target_sum:
            return arr[start:end]

        current_sum += arr[end]

    return []

# Example usage:
arr = [1, 4, 20, 3, 10, 5]
target_sum = 33
result = find_subarray_with_sum(arr, target_sum)
print("Sub-array with sum", target_sum, ":", result)
```

### **OUTPUT**

```
Sub-array with sum 33 : [20, 3, 10]

=== Code Execution Successful ===
```

### Find Smallest Missing Number:

To find the smallest missing number, you can iterate through the array and check for the smallest positive integer that is not present

### SOURCE CODE

```
def smallest_missing_number(arr):  
    arr_set = set(arr)  
    missing_number = 1  
  
    while missing_number in arr_set:  
        missing_number += 1  
  
    return missing_number  
  
arr = [0, -10, 1, 3, -20]  
result = smallest_missing_number(arr)  
print("Smallest missing number:", result)
```

### OUTPUT

```
Smallest missing number: 2
```

```
=== Code Execution Successful ===
```

**WEEK NO. : 6**

**STUDENT NAME :Shristi Yadav**

**DATE: 31/03/24**

**NO. :**

### **PROGRAM NO:8**

**AIM:** Merge two sorted arrays with  $O(1)$  extra space, Search an element in a sorted and rotated array

#### **INTRODUCTION**

##### **Merge two sorted arrays with $o(1)$ extra space**

To merge two sorted arrays with  $O(1)$  extra space, we can use the merge operation similar to the merge step of the merge sort algorithm.

#### **SOURCE CODE**

```
def merge_sorted_arrays(arr1, arr2):
    m, n = len(arr1), len(arr2)
    i, j = m - 1, 0

    while i >= 0 and j < n:
        if arr1[i] > arr2[j]:
            arr1[i], arr2[j] = arr2[j], arr1[i]
            i -= 1
            j += 1
        else:
            break

    arr1.sort()
    arr2.sort()

arr1 = [1, 3, 5, 7]
arr2 = [2, 4, 6, 8]
merge_sorted_arrays(arr1, arr2)
print("Merged array 1:", arr1)
print("Merged array 2:", arr2)
```

#### **OUTPUT**

```
Merged array 1: [1, 2, 3, 4]
Merged array 2: [5, 6, 7, 8]

=== Code Execution Successful ===
```

## Search an element in a sorted and rotated array

We can use binary search to implement the **PROGRAM**

### SOURCE CODE

```
def search_rotated_array(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid

        if arr[low] <= arr[mid]:
            if arr[low] <= target < arr[mid]:
                high = mid - 1
            else:
                low = mid + 1
        else:
            if arr[mid] < target <= arr[high]:
                low = mid + 1
            else:
                high = mid - 1

    return -1

# Example usage:
arr = [4, 5, 6, 7, 8, 9, 1, 2, 3]
target = 6
index = search_rotated_array(arr, target)
if index != -1:
    print("Element", target, "found at index", index)
else:
    print("Element", target, "not found")
```

### OUTPUT

```
Element 6 found at index 2
=== Code Execution Successful ===
```



**WEEK NO. : 6**

**STUDENT NAME :Shristi Yadav**

**DATE: 1/04/24**

**NO. :**

**PROGRAM NO:9**

**AIM:** To sort elements by frequency,to sort an array of dates

**INTRODUCTION**

**To sort elements by frequency**

1. Create a dictionary to count the frequency of each element in the array.
2. Sort the elements based on their frequency.
3. If two elements have the same frequency, sort them based on their original order in the array.

**SOURCE CODE**

```
from collections import Counter

def sort_by_frequency(arr):
    # Count the frequency of each element
    freq_map = Counter(arr)
    # Initialize an empty list to store the sorted array
    sorted_arr = []
    # Iterate through the frequency map
    for num, freq in freq_map.items():
        # Add the number to the sorted array 'freq' times
        sorted_arr.extend([num] * freq)
    return sorted_arr

# Example usage:
arr = [4, 6, 2, 2, 6, 6, 4, 4, 4]
sorted_arr = sort_by_frequency(arr)
print("Sorted array by frequency:", sorted_arr)
```

## OUTPUT

```
Sorted array by frequency: [4, 4, 4, 4, 6, 6, 6, 2, 2]
```

## Sorting an Array of Dates

### SOURCE CODE

```
from datetime import datetime
```

```
def sort_dates(arr):
```

```
    # Create a dictionary to map date objects to their string representations
```

```
    date_map = { }
```

```
    for date_str in arr:
```

```
        date_obj = datetime.strptime(date_str, '%Y-%m-%d')
```

```
        date_map[date_obj] = date_str
```

```
    # Initialize an empty list to store the sorted dates
```

```
    sorted_dates = []
```

```
    # Iterate through sorted date objects
```

```
    for date_obj in sorted(date_map):
```

```
        # Append the string representation of the date to the sorted list
```

```
        sorted_dates.append(date_map[date_obj])
```

```
    return sorted_dates
```

```
# Example usage:
```

```
dates = ['2024-05-30', '2023-12-25', '2024-01-01', '2022-10-15']
```

```
sorted_dates = sort_dates(dates)

print("Sorted dates:", sorted_dates)
```

## OUTPUT

```
Sorted dates: ['2022-10-15', '2023-12-25', '2024-01-01', '2024-05-30']
```

**WEEK NO. :** 7

**STUDENT NAME :** Shristi Yadav

**DATE:** 7/04/24

**NO. :**

### **PROGRAM NO:10**

**AIM:** To use Bucket Sort To Sort an Array with Negative Numbers, K-Way Merge Sort

### **INTRODUCTION**

**Bucket Sort for an array with negative numbers works:**

1. Find the maximum and minimum values in the array.
2. Determine the range of values covered by the buckets.
3. Initialize empty buckets to hold elements within their respective ranges.
4. Distribute elements into buckets based on their values.
5. Sort each bucket, typically using another sorting algorithm like insertion sort.
6. Concatenate the sorted elements from all buckets to produce the final sorted array.

### **SOURCE CODE**

```
def bucket_sort(arr):  
  
    # Find the maximum and minimum values in the array  
  
    max_val = max(arr)  
  
    min_val = min(arr)  
  
  
    # Initialize buckets  
  
    bucket_range = max_val - min_val + 1  
  
    buckets = [[] for _ in range(bucket_range)]  
  
  
    # Place elements in their respective buckets  
  
    for num in arr:
```

```

        buckets[num - min_val].append(num)

# Sort each bucket and concatenate them
sorted_arr = []

for bucket in buckets:

    sorted_arr.extend(sorted(bucket))

return sorted_arr

# Example usage:

arr = [5, -10, 8, -3, 2, -6, 4]

sorted_arr = bucket_sort(arr)

print("Sorted array with negative numbers using Bucket Sort:", sorted_arr)

```

## OUTPUT

```

Sorted array with negative numbers using Bucket Sort: [-10, -6, -3, 2, 4, 5, 8]

=== Code Execution Successful ===

```

## K-Way Merge Sort works:

1. Divide the input array into K sub-arrays of approximately equal size.
2. Recursively apply K-Way Merge Sort to each sub-array.
3. Merge the sorted sub-arrays using a K-way merge operation:
  - Take the first element from each sub-array and compare them.
  - Select the smallest element and append it to the merged array.
  - Repeat this process until all elements from all sub-arrays are merged.

4. Return the merged array as the sorted **OUTPUT**.

## SOURCE CODE

```
def merge(arrays):
    result = []
    # Merge arrays until only one array remains
    while len(arrays) > 1:
        # Take the first two arrays and merge them
        merged_array = []
        i, j = 0, 0
        while i < len(arrays[0]) and j < len(arrays[1]):
            if arrays[0][i] < arrays[1][j]:
                merged_array.append(arrays[0][i])
                i += 1
            else:
                merged_array.append(arrays[1][j])
                j += 1
        # Add remaining elements from the first array
        while i < len(arrays[0]):
            merged_array.append(arrays[0][i])
            i += 1
        # Add remaining elements from the second array
        while j < len(arrays[1]):
            merged_array.append(arrays[1][j])
            j += 1
        # Remove the first two arrays and add the merged array
        arrays = arrays[2:]
        arrays.append(merged_array)
    # Return the final merged array
    return arrays[0]

# Example usage:
arrays = [[5, 10, 15], [2, 7, 12], [1, 6, 11], [3, 8, 13], [4, 9, 14]]
```

```
sorted_array = merge(arrays)
print("Sorted array using K-Way Merge Sort:", sorted_array)
```

## OUTPUT

```
Sorted array using K-Way Merge Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15]

=== Code Execution Successful ===
```

**PROGRAM NO:11**

**AIM:** To find Sum of natural numbers using recursion, Decimal to binary number using recursion

**INTRODUCTION****Sum of natural numbers using recursion**

1. Define a recursive function `sum_of_natural_numbers(n)` that takes an integer `n` as input.
2. If `n` is equal to 1, return 1 (base case).
3. Otherwise, return the sum of `n` and the result of calling the function recursively with `n-1`.
4. Repeat this process until the base case is reached.

**SOURCE CODE**

```
def sum_of_natural_numbers(n):  
    # Base case: If n is 0, return 0  
    if n == 0:  
        return 0  
  
    # Recursive case: Add n to the sum of numbers from 1 to n-1  
    return n + sum_of_natural_numbers(n - 1)  
  
# Example usage:  
n = 5  
print("Sum of natural numbers from 1 to", n, "is:", sum_of_natural_numbers(n))
```

**OUTPUT**

```
Sum of natural numbers from 1 to 5 is: 15  
=== Code Execution Successful ===
```



## Decimal to Binary Number Conversion Using Recursion:

Converting a decimal number to a binary number using recursion involves repeatedly dividing the decimal number by 2 and appending the remainder to the binary representation until the quotient becomes 0. Then, the binary representation is constructed by concatenating the remainders in reverse order.

### SOURCE CODE

```
def decimal_to_binary(n):  
  
    # Base case: If n is 0, return empty string  
  
    if n == 0:  
  
        return "  
  
    # Recursive case: Divide n by 2 and append the remainder to binary representation  
  
    return decimal_to_binary(n // 2) + str(n % 2)  
  
  
# Example usage:  
  
decimal_num = 10  
  
binary_num = decimal_to_binary(decimal_num)  
  
print("Binary representation of", decimal_num, "is:", binary_num)
```

### OUTPUT

```
Binary representation of 10 is: 1010
```

```
=== Code Execution Successful ===
```

**PROGRAM NO:12**

**AIM:** Print reverse of a string using recursion, **PROGRAM** for length of a string using recursion

**INTRODUCTION**

**Print reverse of a string using recursion**

Printing the reverse of a string using recursion involves defining a base case and a recursive case. The base case specifies when the recursion should stop, and the recursive case defines how to break down the problem into smaller sub-problems until reaching the base case. In this case, the base case is when the length of the string is 0, and the recursive case involves printing the last character of the string and then recursively printing the reverse of the substring excluding the last character

**SOURCE CODE**

```
def print_reverse_string(s):
    # Base case: If the length of the string is 0, return
    if len(s) == 0:
        return
    # Recursive case: Print the last character and recursively print the reverse of the substring
    print(s[-1], end="")
    print_reverse_string(s[:-1])

# Example usage:
string = "hello"
print("Reverse of the string", string, "is:", end=' ')
print_reverse_string(string)
```

**OUTPUT**

```
Reverse of the string hello is: olleh
=== Code Execution Successful ===A
```

**PROGRAM for length of a string using recursion**

Finding the length of a string using recursion involves defining a base case and a recursive case. The base case specifies when the recursion should stop, and the recursive case defines how to break down the problem into smaller sub-problems until reaching the base case. In this case, the base case is when the string is empty, and the recursive case involves incrementing the length counter and recursively calculating the length of the substring excluding the first character.

## SOURCE CODE

```
def string_length(s):  
    # Base case: If the string is empty, return 0  
    if s == "":  
        return 0  
    # Recursive case: Increment length counter and recursively calculate length of substring  
    return 1 + string_length(s[1:])  
  
# Example usage:  
string = "hello"  
print("Length of the string", string, "is:", string_length(string))
```

## OUTPUT

```
Length of the string hello is: 5  
  
=== Code Execution Successful ===
```

**PROGRAM NO:13**

**AIM:** Sort the Queue using Recursion, Reversing a queue using recursion

**INTRODUCTION****Sort the Queue using Recursion**

1. Define a function to recursively sort the queue.
2. Define a function to insert an element at its correct position in the queue.
3. In the sorting function, if the queue is empty or contains only one element, return the queue.
4. Otherwise, remove an element from the front of the queue.
5. Recursively sort the remaining elements in the queue.
6. Insert the removed element back into the queue at its correct position.
7. Return the sorted queue.

**SOURCE CODE**

```
def insert_at_correct_position(queue, element):  
    # If queue is empty or element is greater than the last element in queue, append it  
    if len(queue) == 0 or element >= queue[-1]:  
        queue.append(element)  
        return  
    # Remove elements from the rear of queue until we find the correct position for element  
    front = queue.pop()  
    insert_at_correct_position(queue, element)  
    queue.append(front)  
  
def sort_queue(queue):  
    if len(queue) <= 1:  
        return queue  
    # Remove an element from the front  
    front = queue.pop(0)
```

```

# Recursively sort the remaining elements in the queue
sort_queue(queue)
# Insert the removed element back into the queue at its correct position
insert_at_correct_position(queue, front)
return queue

# Example usage:
queue = [4, 2, 1, 3, 5]
print("Original Queue:", queue)
sorted_queue = sort_queue(queue)
print("Sorted Queue:", sorted_queue)

```

## OUTPUT

```

Original Queue: [4, 2, 1, 3, 5]
Sorted Queue: [1, 2, 3, 4, 5]

=== Code Execution Successful ===

```

## Reversing a queue using recursion

1. Define a function to recursively reverse the queue.
2. If the queue is empty, return an empty queue.
3. Otherwise, remove an element from the front of the queue.
4. Recursively reverse the remaining elements in the queue.
5. Insert the removed element back into the rear of the reversed queue.
6. Return the reversed queue.

## SOURCE CODE

```

def reverse_queue(queue):
    if len(queue) == 0:
        return queue
    # Remove an element from the front
    front = queue.pop(0)

```

```
# Recursively reverse the remaining elements in the queue
reverse_queue(queue)
# Insert the removed element back into the rear of the reversed queue
queue.append(front)
return queue
```

# Example usage:

```
queue = [1, 2, 3, 4, 5]
print("Original Queue:", queue)
reversed_queue = reverse_queue(queue)
print("Reversed Queue:", reversed_queue)
```

## OUTPUT

```
Original Queue: [1, 2, 3, 4, 5]
Reversed Queue: [5, 4, 3, 2, 1]
```

```
=== Code Execution Successful ===
```

**WEEK NO. :** 10

**STUDENT NAME :**Shristi Yadav

**DATE:** 21/04/24

**NO. :**

**PROGRAM NO:14**

**AIM:** To find Length of longest palindromic sub-string : Recursion, Convert a String to an Integer using Recursion

**INTRODUCTION**

**Length of longest palindromic sub-string**

1. Define a function to recursively find the length of the longest palindromic substring.
2. Define a base case: If the length of the string is 1 or 0, return 1 or 0 respectively, as a single character or an empty string is always a palindrome.
3. If the first and last characters of the string are equal, recursively find the length of the longest palindromic substring in the substring excluding the first and last characters.
4. If the first and last characters of the string are not equal, recursively find the length of the longest palindromic substring in both substrings excluding either the first or last character, and return the maximum length.
5. Return the length of the longest palindromic substring.

**SOURCE CODE**

```
def longest_palindromic_substring(s):  
    # Base case: If the length of the string is 1 or 0, return 1 or 0 respectively  
    if len(s) <= 1:  
        return len(s)  
  
    # If the first and last characters are equal, check the substring excluding the first and last  
    # characters  
    if s[0] == s[-1]:  
        return 2 + longest_palindromic_substring(s[1:-1])  
  
    # If the first and last characters are not equal, check both substrings excluding either the first  
    # or last character  
    else:  
        return max(longest_palindromic_substring(s[1:]), longest_palindromic_substring(s[:-1]))  
  
# Example usage:  
string = "babad"
```

```
print("Length of longest palindromic substring in", string, "is:",  
      longest_palindromic_substring(string))
```

## OUTPUT

```
Length of longest palindromic substring in babad is: 3  
  
=== Code Execution Successful ===
```

## Convert a String to an Integer using Recursion

1. Define a function to recursively convert a string to an integer.
2. Define a base case: If the string is empty, return 0.
3. Convert the first character of the string to its corresponding integer value using ASCII representation.
4. Recursively convert the remaining substring to an integer.
5. Combine the integer values of the first character and the remaining substring.
6. Return the integer value.

## SOURCE CODE

```
def string_to_integer(s):  
    # Base case: If the string is empty, return 0  
    if not s:  
        return 0  
  
    # Convert the first character to its corresponding integer value using ASCII representation  
    digit = ord(s[0]) - ord('0')  
  
    # Recursively convert the remaining substring to an integer  
    remaining_int = string_to_integer(s[1:])  
  
    # Combine the integer values of the first character and the remaining substring  
    return digit * (10 ** len(s[1:])) + remaining_int  
  
# Example usage:  
string = "12345"
```



```
print("Integer value of", string, "is:", string_to_integer(string))
```

## OUTPUT

```
Integer value of 12345 is: 12345
```

```
=== Code Execution Successful ===|
```

WEEK NO. : 10

STUDENT NAME :Shristi Yadav

DATE: 22/04/24

NO. :

### PROGRAM NO:15

**AIM:** To find DFS traversal of a Tree, How to Sort a Stack using Recursion

#### INTRODUCTION

##### DFS traversal of a Tree

- **Preorder:** In preorder traversal, we visit the current node first, then recursively traverse the left subtree, and finally recursively traverse the right subtree.
- **Inorder:** In inorder traversal, we recursively traverse the left subtree first, then visit the current node, and finally recursively traverse the right subtree. This approach is typically used for binary search trees to get the elements in sorted order.
- **Postorder:** In postorder traversal, we recursively traverse the left subtree first, then recursively traverse the right subtree, and finally visit the current node.

#### SOURCE CODE

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

def dfs_preorder(root):
    if root is None:
        return
    print(root.val, end=" ")
    dfs_preorder(root.left)
    dfs_preorder(root.right)

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

print("DFS Preorder Traversal:")
dfs_preorder(root)
```

## OUTPUT

```
DFS Preorder Traversal:  
1 2 4 5 3  
=== Code Execution Successful ===
```

## Sort a Stack using Recursion

1. Define a function to recursively sort the stack.
2. Define a base case: If the stack is empty or contains only one element, return.
3. Pop an element from the stack.
4. Recursively sort the remaining elements in the stack.
5. Insert the popped element back into the stack at its correct position.
6. Return the sorted stack.

## SOURCE CODE

```
def insert_in_sorted_order(stack, element):
```

```
    # Base case: If the stack is empty or the element is greater than or equal to the top element,  
    push the element
```

```
    if len(stack) == 0 or element >= stack[-1]:
```

```
        stack.append(element)
```

```
        return
```

```
    # If the element is smaller than the top element, pop the top element and recursively insert the  
    element in sorted order
```

```
    top = stack.pop()
```

```
    insert_in_sorted_order(stack, element)
```

```
    stack.append(top)
```

```
def sort_stack(stack):
```

```
    # Base case: If the stack is empty or contains only one element, return
```

```
    if len(stack) <= 1:
```

```
        return
```

```
    # Pop an element from the stack
```

```
    top = stack.pop()
```

```
# Recursively sort the remaining elements in the stack
sort_stack(stack)
# Insert the popped element back into the stack at its correct position
insert_in_sorted_order(stack, top)
```

# Example usage:

```
stack = [5, 2, 7, 3, 1]
print("Original Stack:", stack)
sort_stack(stack)
print("Sorted Stack:", stack)
```

## OUTPUT

```
Original Stack: [5, 2, 7, 3, 1]
Sorted Stack: [1, 2, 3, 5, 7]
```

```
=== Code Execution Successful ===
```

**PROGRAM NO:16****AIM:**Find geometric sum of the series using recursion, Implement Queue using Stacks**INTRODUCTION****geometric sum of the series**

1. Define a function to recursively calculate the geometric sum.
2. Define a base case: If the current term becomes less than a very small value (epsilon), return 0 to avoid infinite recursion.
3. Calculate the current term of the series using the formula  $1/(2^k)$ , where  $k$  is the current index.
4. Recursively calculate the geometric sum for the next index.
5. Add the current term to the geometric sum calculated for the next index.
6. Return the geometric sum.

**SOURCE CODE**

```
def geometric_sum(k):  
    epsilon = 1e-9 # A very small value to avoid infinite recursion  
    # Base case: If the current term becomes less than epsilon, return 0  
    if k < epsilon:  
        return 0  
    # Calculate the current term of the series  
    term = 1 / (2 ** k)  
    # Recursively calculate the geometric sum for the next index  
    next_term = geometric_sum(k - 1)  
    # Add the current term to the geometric sum calculated for the next index  
    return term + next_term  
  
# Example usage:  
k = 5  
print("Geometric sum of the series up to", k, "terms:", geometric_sum(k))
```

## OUTPUT

```
Geometric sum of the series up to 5 terms: 0.96875  
=== Code Execution Successful ===
```

### , Implement Queue using Stacks

1. Define a class **QueueUsingStacks** with two stacks: **enqueue\_stack** for enqueueing elements and **dequeue\_stack** for dequeueing elements.
2. Implement the **enqueue** operation:
  - Push the element onto the **enqueue\_stack**.
3. Implement the **dequeue** operation:
  - If the **dequeue\_stack** is empty, transfer all elements from the **enqueue\_stack** to the **dequeue\_stack**.
  - Pop and return the top element from the **dequeue\_stack**.
4. Implement the **is\_empty** operation to check if the queue is empty.

## SOURCE CODE

```
class QueueUsingStacks:
    def __init__(self):
        self.enqueue_stack = []
        self.dequeue_stack = []

    def enqueue(self, element):
        self.enqueue_stack.append(element)

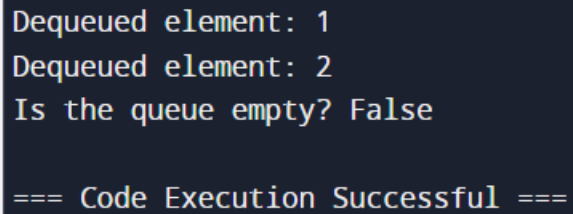
    def dequeue(self):
        if not self.dequeue_stack:
            # Transfer elements from enqueue_stack to dequeue_stack
            while self.enqueue_stack:
                self.dequeue_stack.append(self.enqueue_stack.pop())
            # Pop and return the top element from dequeue_stack
            return self.dequeue_stack.pop() if self.dequeue_stack else None

    def is_empty(self):
        return not (self.enqueue_stack or self.dequeue_stack)

# Example usage:
queue = QueueUsingStacks()
queue.enqueue(1)
queue.enqueue(2)
```

```
queue.enqueue(3)
print("Dequeued element:", queue.dequeue())
queue.enqueue(4)
print("Dequeued element:", queue.dequeue())
print("Is the queue empty?", queue.is_empty())
```

## OUTPUT

A terminal window with a dark background and light-colored text. It displays the output of the code: "Dequeued element: 1", "Dequeued element: 2", and "Is the queue empty? False". At the bottom, it shows "=== Code Execution Successful ===".

```
Dequeued element: 1
Dequeued element: 2
Is the queue empty? False

=== Code Execution Successful ===
```

**PROGRAM NO:17****AIM:** To find Infix to Postfix Conversion using Stack, Height of Binary Tree**INTRODUCTION****find Infix to Postfix Conversion using Stack**

Infix notation is the common arithmetic and logical formula notation, in which operators are written between the operands they act on (e.g.,  $2 + 3 * 4$ ). Postfix notation, also known as Reverse Polish Notation (RPN), is a mathematical notation in which every operator follows all of its operands (e.g.,  $2\ 3\ 4\ * +$ ). Converting infix notation to postfix notation involves rearranging the operands and operators according to the rules of operator precedence.

**SOURCE CODE**

```
def infix_to_postfix(infix_expr):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    postfix_expr = []
    stack = []

    for char in infix_expr:
        if char.isdigit():
            postfix_expr.append(char)
        elif char in precedence:
            while (stack and stack[-1] != '(' and
                   precedence.get(stack[-1], 0) >= precedence[char]):
                postfix_expr.append(stack.pop())
            stack.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack[-1] != '(':
                postfix_expr.append(stack.pop())
            stack.pop()

    while stack:
        postfix_expr.append(stack.pop())

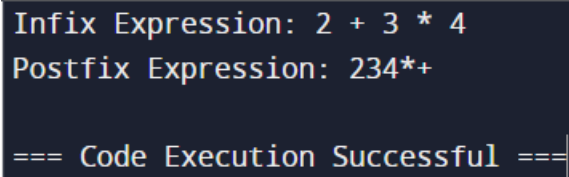
    return "".join(postfix_expr)

# Example usage:
infix_expression = "2 + 3 * 4"
```



```
postfix_expression = infix_to_postfix(infix_expression)
print("Infix Expression:", infix_expression)
print("Postfix Expression:", postfix_expression)
```

## OUTPUT



```
Infix Expression: 2 + 3 * 4
Postfix Expression: 234*+

=== Code Execution Successful ===
```

## Height of Binary Tree

The height of a binary tree is the length of the longest path from the root node to any leaf node in the tree. It represents the number of edges on the longest path from the root node to a leaf node. The height of an empty tree (or a tree with only one node) is 0.

## SOURCE CODE

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

def height_of_binary_tree(root):
    if root is None:
        return -1
    left_height = height_of_binary_tree(root.left)
    right_height = height_of_binary_tree(root.right)
    return max(left_height, right_height) + 1

# Example usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print("Height of Binary Tree:", height_of_binary_tree(root))
```

## OUTPUT

```
Height of Binary Tree: 2
```

```
=== Code Execution Successful ===
```

**PROGRAM NO:18**

**AIM:**To create Mirror tree , Largest value in each level of binary tree

**INTRODUCTION****Mirror tree**

The mirror of a binary tree is obtained by swapping the left and right subtrees of every node in the tree. This effectively reflects the binary tree across its vertical axis.

**SOURCE CODE**

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

def mirror_tree(root):
    if root is None:
        return None
    # Recursively mirror the left subtree and the right subtree
    left_mirror = mirror_tree(root.left)
    right_mirror = mirror_tree(root.right)
    # Swap the left and right subtrees of the current node
    root.left, root.right = right_mirror, left_mirror
    return root

# Example usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

mirrored_root = mirror_tree(root)
print("Mirrored Tree:")
print("    ", mirrored_root.val)
print("  /\\"")
print(" ", mirrored_root.right.val, " ", mirrored_root.left.val)
print("  /\\"")
print("    ", mirrored_root.left.right.val, " ", mirrored_root.left.left.val)
```

## OUTPUT

```
^ Mirrored Tree:
    1
   / \
  2   3
   / \
  /   \

ERROR!
Traceback (most recent call last):
  File "<main.py>", line 30, in <module>
AttributeError: 'NoneType' object has no attribute 'val'

=== Code Exited With Errors ===
```

## Largest value in each level of binary tree

Finding the largest value in each level of a binary tree involves performing a level-order traversal of the tree and keeping track of the maximum value in each level.

## SOURCE CODE

```
from collections import deque

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

def largest_values_in_levels(root):
    if root is None:
        return []

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

    while queue:
        level_max = float('-inf')
        level_size = len(queue)

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

```
        level_max = max(level_max, node.val)
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

    max_values.append(level_max)

    return max_values

# Example usage:
root = TreeNode(1)
root.left = TreeNode(3)
root.right = TreeNode(2)
root.left.left = TreeNode(5)
root.left.right = TreeNode(3)
root.right.right = TreeNode(9)

print("Largest value in each level:", largest_values_in_levels(root))
```

## OUTPUT

```
Largest value in each level: [1, 3, 9]
```

```
=== Code Execution Successful ===
```

**PROGRAM NO:19**

**AIM:** To find Minimum Time to rot all oranges, Huffman Coding

**INTRODUCTION****Minimum Time to rot all oranges**

In the "Minimum Time to Rot All Oranges" problem, you're given a grid where each cell represents an orange, and you need to determine the minimum time required to rot all oranges. Oranges rot over time, and adjacent fresh oranges (horizontally or vertically) rot each other in one unit of time. However, diagonally adjacent oranges do not rot each other.

**SOURCE CODE**

```
from collections import deque

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

    # Enqueue rotten oranges and count fresh oranges
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 2:
                queue.append((i, j))
            elif grid[i][j] == 1:
                fresh_count += 1

    # Define directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    time = 0

    while queue and fresh_count > 0:
        time += 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
```

```

        queue.append((nx, ny))
        fresh_count -= 1

    return time if fresh_count == 0 else -1

# Example usage:
grid = [
    [2, 1, 1],
    [1, 1, 0],
    [0, 1, 1]
]
print("Minimum time to rot all oranges:", min_time_to_rot_oranges(grid))

```

## OUTPUT

```

from collections import deque

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

    # Enqueue rotten oranges and count fresh oranges
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 2:
                queue.append((i, j))
            elif grid[i][j] == 1:
                fresh_count += 1

    # Define directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    time = 0

    while queue and fresh_count > 0:
        time += 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
                    queue.append((nx, ny))
                    fresh_count -= 1

    return time if fresh_count == 0 else -1

# Example usage:
grid = [

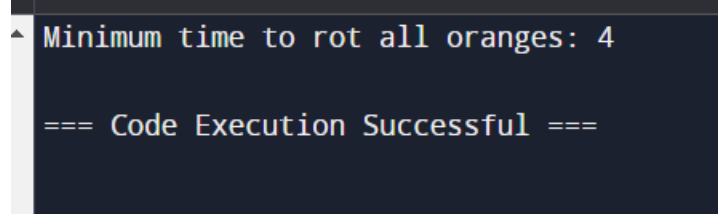
```

```

    [2, 1, 1],
    [1, 1, 0],
    [0, 1, 1]
]
print("Minimum time to rot all oranges:", min_time_to_rot_oranges(grid))

```

## OUTPUT



```

^ Minimum time to rot all oranges: 4

=== Code Execution Successful ===

```

## Huffman Coding

Huffman coding is a lossless data compression algorithm used for encoding data. It assigns variable-length codes to input characters based on their frequencies, with shorter codes assigned to more frequent characters. This results in a prefix-free binary code, where no code is a prefix of any other code.

## SOURCE CODE

```

import heapq
from collections import defaultdict

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

def build_huffman_tree(freq_map):
    heap = []
    for char, freq in freq_map.items():
        heapq.heappush(heap, (freq, Node(char, freq)))

    while len(heap) > 1:
        freq1, node1 = heapq.heappop(heap)
        freq2, node2 = heapq.heappop(heap)
        internal_node = Node(None, freq1 + freq2)
        internal_node.left, internal_node.right = node1, node2
        heapq.heappush(heap, (freq1 + freq2, internal_node))

```



```
return heapq.heappop(heap)[1]

def generate_huffman_codes(root, code, huffman_codes)
    if root:
        if root.char:
            huffman_codes[root.char] = code
            generate_huffman_codes(root.left, code + '0', huffman_codes)
            generate_huffman_codes(root.right, code + '1', huffman_codes)

def huffman_coding(text):
    freq_map
```

## OUTPUT

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 25
    def generate_huffman_codes(root, code, huffman_codes)
                                                                    ^
SyntaxError: expected ':'

=== Code Exited With Errors ===
```

**WEEK NO. :** 13

**STUDENT NAME :**Shristi Yadav

**DATE:** 19/05/24

**NO. :**

### **PROGRAM NO:20**

**AIM:** Coin change problem, Floyd Warshall Algorithm

#### **INRODUCTION:**

#### **COIN CHANGE PROBLEM**

The coin change problem involves finding the number of ways to make change for a given amount using a specific set of coin denominations. It is a classic problem in dynamic **PROGRAM**ming.

#### **SOURCE CODE**

```
def coin_change(amount, coins):

    ways = [0] * (amount + 1)

    ways[0] = 1

    for coin in coins:

        for j in range(coin, amount + 1):

            ways[j] += ways[j - coin]

    return ways[amount]

# Example usage:

amount = 5

coins = [1, 2, 5]

print("Number of ways to make change for amount", amount, ":", coin_change(amount, coins))
```

## OUTPUT

```
Number of ways to make change for amount 5 : 4
```

```
=== Code Execution Successful ===
```

## Floyd Warshall Algorithm

The Floyd-Warshall algorithm is a dynamic **PROGRAM**ming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights (but with no negative cycles). It's particularly useful for finding shortest paths in dense graphs, where other algorithms like Dijkstra's might be less efficient.

## SOURCE CODE

```
INF = float('inf')
```

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

    for i in range(num_vertices):
        dist[i][i] = 0

    for i in range(num_vertices):
        for j in range(num_vertices):
            if graph[i][j] != INF:
                dist[i][j] = graph[i][j]

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

```
# Example usage:
```

```
graph = [
    [0, 5, INF, 10],
    [INF, 0, 3, INF],
    [INF, INF, 0, 1],
    [INF, INF, INF, 0]
```

```
]
```

```
shortest_distances = floyd_warshall(graph)  
for row in shortest_distances:  
    print(row)
```

## OUTPUT

```
[0, 5, 8, 9]  
[inf, 0, 3, 4]  
[inf, inf, 0, 1]  
[inf, inf, inf, 0]  
  
=== Code Execution Successful ===
```

**WEEK NO. :** 13

**STUDENT NAME :**Shristi Yadav

**DATE:** 20/05/24

**NO. :**

### **PROGRAM NO:21**

**AIM:** To implement PRIM'S ALGORITHM

### **INTRODUCTION**

Prim's algorithm is a greedy algorithm used to find a minimum spanning tree (MST) for a weighted undirected graph. The algorithm starts with an arbitrary vertex and grows the MST by iteratively adding the shortest edge that connects a vertex in the MST to a vertex outside the MST until all vertices are included in the MST.

### **SOURCE CODE**

```
import heapq

def prim(graph):
    num_vertices = len(graph)

    mst = set()

    total_cost = 0

    pq = []

    # Start from vertex 0
    heapq.heappush(pq, (0, 0)) # (weight, vertex)

    while len(mst) < num_vertices:
        weight, vertex = heapq.heappop(pq)

        if vertex not in mst:
            mst.add(vertex)
```

```

        total_cost += weight

    for neighbor, edge_weight in graph[vertex]:

        if neighbor not in mst:

            heapq.heappush(pq, (edge_weight, neighbor))

    return total_cost

# Example usage:

graph = {

    0: [(1, 2), (2, 1), (3, 4)],

    1: [(0, 2), (2, 3)],

    2: [(0, 1), (1, 3), (3, 5)],

    3: [(0, 4), (2, 5)]

}

minimum_cost = prim(graph)

print("Minimum cost of MST:", minimum_cost)

```

## OUTPUT

```

Minimum cost of MST: 7

=== Code Execution Successful ===

```

**WEEK NO. :** 14

**STUDENT NAME :**Shristi Yadav

**DATE:** 26/05/24

**NO. :**

### **PROGRAM NO:22**

**AIM:**To implement longest subsequence problem

### **INTRODUCTION**

The Longest Common Subsequence (LCS) problem is a classic problem in computer science and bioinformatics. Given two sequences (usually strings), the goal is to find the longest subsequence present in both of them.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. For example, "ABC", "AC", and "BC" are subsequences of "ABCD".

### **SOURCE CODE**

```
def longest_common_subsequence(s1, s2):  
  
    m, n = len(s1), len(s2)  
  
    dp = [[0] * (n + 1) for _ in range(m + 1)]  
  
    for i in range(1, m + 1):  
        for j in range(1, n + 1):  
            if s1[i - 1] == s2[j - 1]:  
                dp[i][j] = 1 + dp[i - 1][j - 1]  
            else:  
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])  
  
    return dp[m][n]
```

# Example usage:

```
s1 = "AGGTAB"
```

```
s2 = "GXTXAYB"
```

```
print("Length of Longest Common Subsequence:", longest_common_subsequence(s1, s2))
```

## OUTPUT

```
Length of Longest Common Subsequence: 4
```

```
=== Code Execution Successful ===
```