In [1]:

```
#Implement depth first search algorithm Use an undirected graph and develop a recursive
#algorithm for searching all the vertices of a graph or tree data structure.


# Graph class representing the undirected graph
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.adj_list = [[] for _ in range(vertices)]  # Adjacency list

    def add_edge(self, u, v):
        # Add an edge between vertices u and v
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def dfs(self, v, visited):
        # Mark the current vertex as visited
        visited[v] = True
        print(v, end=" ")

        # Recur for all the adjacent vertices
        for neighbor in self.adj_list[v]:
            if not visited[neighbor]:
                self.dfs(neighbor, visited)

    def dfs_traversal(self):
        # Initialize visited array
        visited = [False] * self.V

        # Call the recursive helper function for all vertices
        for i in range(self.V):
            if not visited[i]:
                self.dfs(i, visited)


# Example usage
if __name__ == "__main__":
    # Create a graph
    g = Graph(6)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)

    print("Depth-First Traversal (starting from vertex 0):")
    g.dfs_traversal()
```

```
Depth-First Traversal (starting from vertex 0):
0 1 3 2 4 5
```

In [3]:

```
#Breadth First Search algorithm, Use an undirected graph and develop a recursive algorit
#for searching all the vertices of a graph or tree data structure.



from collections import deque

class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.adj_list = [[] for _ in range(vertices)]  # Adjacency list

    def add_edge(self, u, v):
        # Add an edge between vertices u and v
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def bfs_traversal(self):
        # Initialize visited array
        visited = [False] * self.V

        # Create a queue for BFS
        queue = deque()

        # Start from all unvisited vertices
        for i in range(self.V):
            if not visited[i]:
                visited[i] = True
                queue.append(i)

                while queue:
                    vertex = queue.popleft()
                    print(vertex, end=" ")

                    # Visit all adjacent vertices of the dequeued vertex
                    for neighbor in self.adj_list[vertex]:
                        if not visited[neighbor]:
                            visited[neighbor] = True
                            queue.append(neighbor)

# Example usage
if __name__ == "__main__":
    # Create a graph
    g = Graph(6)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)

    print("Breadth-First Traversal:")
    g.bfs_traversal()
```

```
Breadth-First Traversal:
0 1 2 3 4 5
```

In [8]:

```
#Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and
#Backtracking for n-queens problem or a graph coloring problem




class ConstraintSatisfactionProblem:
    def __init__(self, problem_type):
        self.problem_type = problem_type

    def solve_graph_coloring(self, graph, colors, solution):
        n = len(graph)
        if self.backtrack_coloring(graph, colors, solution, 0):
            return solution
        else:
            return None

    def is_valid_coloring(self, graph, solution, vertex, color):
        for neighbor in graph[vertex]:
            if solution[neighbor] == color:
                return False
        return True

    def backtrack_coloring(self, graph, colors, solution, vertex):
        n = len(graph)
        if vertex == n:
            return True
        for color in colors:
            if self.is_valid_coloring(graph, solution, vertex, color):
                solution[vertex] = color
                if self.backtrack_coloring(graph, colors, solution, vertex + 1):
                    return True
                solution[vertex] = -1
        return False

# Example usage
if __name__ == "__main__":
    csp = ConstraintSatisfactionProblem(problem_type="graph_coloring")
    graph = [[1, 2], [0, 2, 3], [0, 1, 3], [1, 2]]
    colors = [0, 1, 2]
    solution = [-1] * len(graph)  # Initialize solution list with the size of the graph
    coloring_solution = csp.solve_graph_coloring(graph, colors, solution)
    print("Graph Coloring Solution:")
    print(coloring_solution)
```

```
Graph Coloring Solution:
[0, 1, 2, 0]
```

In [*]:

```
#Develop an elementary chatbot for any suitable customer interaction application
import random

# Define a list of greetings
greetings = ["hello", "hi", "hey", "greetings"]

# Define a list of responses
responses = ["Hello!", "Hi there!", "Hey!", "Nice to meet you!"]

# Function to generate a random response
def get_random_response():
    return random.choice(responses)

# Main chatbot loop
while True:
    # Get user input
    user_input = input("User: ")

    # Convert user input to lowercase
    user_input = user_input.lower()

    # Check if user input matches any greeting
    if user_input in greetings:
        # Get a random response
        bot_response = get_random_response()
    else:
        # Default response
        bot_response = "I'm sorry, I didn't understand that."

    # Print bot's response
    print("ChatBot:", bot_response)
```

```
User: hello
ChatBot: Hey!
User: hi
ChatBot: Hi there!
User: hey
ChatBot: Hey!
User: greetings
ChatBot: I'm sorry, I didn't understand that.
User: greetings
ChatBot: Hey!
```

User:

In [*]:

```
#Implement Greedy search algorithm for any of the following application:
#Prims Minimal Spanning Tree Algorithm



import heapq

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[] for _ in range(vertices)]

    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))

    def prim_mst(self):
        # Initialize a list to store the MST
        mst = []
        start_vertex = 0  # Choose the first vertex as the starting point

        # Create a list to keep track of visited vertices
        visited = [False] * self.V

        # Create a priority queue to store the edges with their weights
        pq = [(0, start_vertex)]

        while pq:
            weight, vertex = heapq.heappop(pq)

            if visited[vertex]:
                continue

            # Mark the current vertex as visited
            visited[vertex] = True

            # Add the selected edge to the MST
            if vertex != start_vertex:
                mst.append((parent, vertex, weight))

            # Explore the adjacent vertices
            for neighbor, edge_weight in self.graph[vertex]:
                if not visited[neighbor]:
                    heapq.heappush(pq, (edge_weight, neighbor))
                    parent = vertex

        return mst

# Example usage
if __name__ == "__main__":
    g = Graph(6)
    g.add_edge(0, 1, 5)
    g.add_edge(0, 2, 3)
    g.add_edge(1, 2, 2)
    g.add_edge(1, 3, 1)
    g.add_edge(2, 3, 4)
    g.add_edge(2, 4, 6)
    g.add_edge(3, 4, 2)
    g.add_edge(3, 5, 3)
```

```
    g.add_edge(4, 5, 6)

    mst = g.prim_mst()

    # Print the Minimum Spanning Tree edges
    print("Minimum Spanning Tree Edges:")
    for u, v, weight in mst:
        print(u, "--", v, ":", weight)
```

In [*]:

```
#Write a Java/C/C++/Python program to perform encryption using the method
#of Transposition technique.

def encrypt_transposition(plaintext, key):
    # Remove any spaces from the plaintext
    plaintext = plaintext.replace(" ", "")

    # Calculate the number of rows required for the transposition grid
    rows = len(plaintext) // key
    if len(plaintext) % key != 0:
        rows += 1

    # Create the transposition grid
    grid = [[''] * key for _ in range(rows)]

    # Fill the grid with the plaintext characters
    index = 0
    for row in range(rows):
        for col in range(key):
            if index < len(plaintext):
                grid[row][col] = plaintext[index]
                index += 1

    # Read the encrypted message column-wise
    ciphertext = ""
    for col in range(key):
        for row in range(rows):
            ciphertext += grid[row][col]

    return ciphertext

# Example usage
if __name__ == "__main__":
    plaintext = "HELLO WORLD"
    key = 4

    ciphertext = encrypt_transposition(plaintext, key)

    print("Plaintext:", plaintext)
    print("Ciphertext:", ciphertext)


#
```

In [*]:

```
#Write a Java/C/C++/Python program to perform decryption using the method of
#Transposition technique


def decrypt_transposition(ciphertext, key):
    # Calculate the number of rows required for the transposition grid
    rows = len(ciphertext) // key
    if len(ciphertext) % key != 0:
        rows += 1

    # Calculate the number of empty cells in the last row
    empty_cells = (rows * key) - len(ciphertext)

    # Create the transposition grid
    grid = [[''] * key for _ in range(rows)]

    # Calculate the number of columns in the last row
    cols = key - empty_cells

    # Calculate the number of filled cells in the last column
    filled_cells = rows - 1

    # Fill the grid with the ciphertext characters
    index = 0
    for col in range(cols):
        for row in range(rows):
            grid[row][col] = ciphertext[index]
            index += 1

    # Read the decrypted message row-wise
    plaintext = ""
    for row in range(rows):
        if row == rows - 1:
            for col in range(cols):
                plaintext += grid[row][col]
        else:
            for col in range(key):
                plaintext += grid[row][col]

    return plaintext

# Example usage
if __name__ == "__main__":
    ciphertext = "HOLEDLRLWO"
    key = 4

    plaintext = decrypt_transposition(ciphertext, key)

    print("Ciphertext:", ciphertext)
    print("Plaintext:", plaintext)
```

In [*]:

```
#Write a Java/C/C++/Python program to implement AES Algorithm.



from cryptography.fernet import Fernet

# Generate a random encryption key
key = Fernet.generate_key()

# Create a Fernet cipher object with the key
cipher = Fernet(key)

# Encryption
plaintext = b'This is the plaintext message'
ciphertext = cipher.encrypt(plaintext)

# Decryption
decrypted_text = cipher.decrypt(ciphertext)

print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
print("Decrypted text:", decrypted_text
```

In [*]:

```
##Write a Java/C/C++/Python program to implement RSA algorithm



import random


def generate_keypair(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1
    e = find_coprime(phi)

    # Compute modular inverse of e
    d = mod_inverse(e, phi)

    return (n, e), (n, d)


def find_coprime(phi):
    # Find a random number that is coprime with phi
    while True:
        e = random.randint(2, phi - 1)
        if gcd(e, phi) == 1:
            return e


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


def mod_inverse(a, m):
    # Extended Euclidean Algorithm to compute modular inverse
    # Returns None if modular inverse does not exist
    r1, r2 = m, a
    s1, s2 = 0, 1

    while r2 != 0:
        q = r1 // r2
        r1, r2 = r2, r1 - q * r2
        s1, s2 = s2, s1 - q * s2

    if r1 == 1:
        return s1 % m
    else:
        return None


def encrypt(message, public_key):
    n, e = public_key
    encrypted_message = [pow(ord(char), e, n) for char in message]
    return encrypted_message


def decrypt(encrypted_message, private_key):
    n, d = private_key
```

```
    decrypted_message = [chr(pow(char, d, n)) for char in encrypted_message]
    return "".join(decrypted_message)


# Example usage
p = 61
q = 53
public_key, private_key = generate_keypair(p, q)

message = "Hello, World!"
encrypted_message = encrypt(message, public_key)
decrypted_message = decrypt(encrypted_message, private_key)

print("Original message:", message)
print("Encrypted message:", encrypted_message)
print("Decrypted message:", decrypted_message)
```

In [*]:

```
##Implement the Diffie-Hellman Key Exchange algorithm.

def mod_exp(base, exponent, modulus):
    # Modular exponentiation function
    result = 1
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exponent //= 2
    return result


def generate_key(p, g):
    # Generate a private key 'a' as a random integer
    a = random.randint(2, p - 2)

    # Calculate public key 'A'
    A = mod_exp(g, a, p)

    return a, A


def compute_secret_key(public_key, private_key, p):
    # Calculate the shared secret key
    return mod_exp(public_key, private_key, p)


# Example usage
p = 23  # Prime modulus
g = 5   # Generator

# Generate keys for Alice
a_private, A_public = generate_key(p, g)

# Generate keys for Bob
b_private, B_public = generate_key(p, g)

# Exchange public keys

# Compute shared secret key for Alice
alice_secret_key = compute_secret_key(B_public, a_private, p)

# Compute shared secret key for Bob
bob_secret_key = compute_secret_key(A_public, b_private, p)

# Check if the shared secret keys match
if alice_secret_key == bob_secret_key:
    print("Shared secret key:", alice_secret_key)
    print("Key exchange successful!")
else:
    print("Key exchange failed!")
```

In [ ]:

```
####Write a Java/C/C++/Python program that contains a string (char pointer) with a value
#World'. The program should AND or and XOR each character in this string with 127 and
#display the result.



def perform_operations(s):
    result_and = ""
    result_xor = ""
    for char in s:
        # Perform AND operation with 127
        char_and = chr(ord(char) & 127)
        result_and += char_and

        # Perform XOR operation with 127
        char_xor = chr(ord(char) ^ 127)
        result_xor += char_xor

    return result_and, result_xor


# Main program
s = "Hello World"

result_and, result_xor = perform_operations(s)

print("Original String:", s)
print("AND Result:", result_and)
print("XOR Result:", result_xor)
```