# tlmasyg3d

December 21, 2024

1. Write a program to implement the following tasks

a. Check if a given number is prime in O(sqrt(n)) time

[2]:
```
def is_prime(n):
    if n <= 1:
        return False
    elif n == 2:
        return True
    else:
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True

number = int(input("Enter a number: "))
if is_prime(number):
    print("PRIME")
else:
    print("NOT PRIME")
```

```
Enter a number:  3
```

```
PRIME
```

b. Linear search

[2]:
```
def linearsearch(arr, val):
    n = len(arr)
    for i in range(n):
        if arr[i] == val:
            return i
    return -1

arr = [1, 2, 3, 4]
val = 3
i = linearsearch(arr, val)

if i != -1:
    print(f"YES. The number {val} found at index {i}")
```

```python
else:
    print("NO. not found")
```

```
YES. The number 3 found at index 2
```

c. Binary search

[4]:
```python
def binarysearch(x, start, end, key):
    if start > end:
        return -1
    mid = (start + end) // 2
    if x[mid] == key:
        return mid
    elif x[mid] > key:
        return binarysearch(x, start, mid - 1, key)
    else:
        return binarysearch(x, mid + 1, end, key)

x = [1, 2, 3, 4, 8, 9]
key = 4
start = 0
end = len(x) - 1
i = binarysearch(x, start, end, key)

if i != -1:
    print(f"YES. The number {key} found at index {i}")
else:
    print("NO. Not found")
```

```
YES. The number 4 found at index 3
```

d. Bubble sort

[7]:
```python
def bubbleSort(arr, n):
    for i in range(n - 1): #sorting requires at most n-1 passes for n elements)
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]


arr = [64, 34, 25, 12, 22, 11, 90]
n = len(arr)
bubbleSort(arr, n)
print("Sorted array is:", arr)
```

```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

e. Selection sort

[6]:

```
def Selectionsort(arr):
    n=len(arr)
    for num in range(n-1):
        min=num
        for i in range(num+1,n):
            if arr[i]< arr[min]:
                min=i

        arr[min], arr[num] = arr[num], arr[min]

arr=[4,2,7,1,8,3]
Selectionsort(arr)
print(arr)
```

```
[1, 2, 3, 4, 7, 8]
```

2. What is the time complexity of bubble sort? Modify the bubble sort and selection sort algorithm to make worst case and best case complexities different.

[ ]:

```
Time Complexity of Bubble Sort>>
Worst Case:  ( ^2)
Best Case:  ( )

Time Complexity of Selection Sort>>
Worst Case:  ( ^2)
Best Case:  ( ^2)
```

[8]:

```
def bubbleSort(arr, n):
    for i in range(n - 1):
        swapped = False
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps occurred, the array is sorted
        if swapped == False:
            break


# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
n = len(arr)
bubbleSort(arr, n)
print("Optimized Bubble Sort - Sorted array is:", arr)
```

```
Optimized Bubble Sort - Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

[8]:
```python
def Selectionsort(arr):
    n = len(arr)
    for num in range(n - 1):
        min_index = num
        for i in range(num + 1, n):
            if arr[i] < arr[min_index]:
                min_index = i
        # Swap only if the current element isn't already in the correct position
        if min_index != num:
            arr[num], arr[min_index] = arr[min_index], arr[num]


# Example usage
arr = [4, 2, 7, 1, 8]
Selectionsort(arr)
print("Optimized Selection Sort - Sorted array is:", arr)
```

```
Optimized Selection Sort - Sorted array is: [1, 2, 4, 7, 8]
```

3. Write a program to implement recursive algorithms to compute

a. The largest element in an array

[13]:
```python
def Largest(arr, n):
    if n == 1:
        return arr[0]
    return max(arr[n-1], Largest(arr, n-1))

arr=[4,2,7,1,8]
n = len(arr)
print(Largest(arr,n))
```

```
8
```

b. GCD of two numbers in O(log n) time.

[12]:
```python
def GCD(a,b):
    if b==0:
        return a
    else:
        return GCD(b, a % b)

num1 = 48
num2 = 18
print(GCD(num1, num2))
```

```
6
```

c. x power y in O(log n ) time

[14]:
```python
def power(x,y):
    if y==0:
        return 1
    else:
        return x*power(x,y-1)

num1 = 8
num2 = 2
print(power(num1, num2))
```

```
64
```

d. if the given number is a numeric palindrome

[6]:
```python
def is_palindrome(num):
    str_num = str(num)
    return str_num == str_num[::-1]

number = 12321
if is_palindrome(number):
    print(f"{number} is a numeric palindrome.")
else:
    print(f"{number} is not a numeric palindrome.")
```

```
12321 is a numeric palindrome.
```

e. reverse of a given number

[7]:
```python
def reverse_number(num):
    return int(str(num)[::-1])

number = 12345
reversed_number = reverse_number(number)
print(f"The reverse of {number} is: {reversed_number}")
```

```
The reverse of 12345 is: 54321
```

4. Given a sorted array A of integers and a number m, implement a program to find the position p where m can be inserted so that the array remains sorted, in O(log n) time

[1]:
```python
def find_insert_position(arr, m):
    left, right = 0, len(arr)

    while left < right:
        mid = (left + right) // 2
        if arr[mid] < m:
            left = mid + 1
        else:
            right = mid
```

```
    return left

A = [1, 3, 5, 8, 7, 9,10,17]
m = 6
position = find_insert_position(A, m)
print(f"The position to insert {m} is: {position}")
```

```
The position to insert 6 is: 3
```

5. Implement a program to compute n power 1/k for a given n and k in O(log n) time.

[ ]:
```
def kth_root(n, k, precision=1e-6):
    low, high = 0, n
    while high - low > precision:
        mid = (low + high) / 2
        if mid**k < n:
            low = mid
        else:
            high = mid
    return low

n = 27
k = 3
result = kth_root(n, k)
print(f"The {k}-th root of {n} is approximately: {result}")
```

7. Implement BFS using a Queue.

[5]:
```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    result = []

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            for neighbor in graph.get(node, []):
                if neighbor not in visited:
                    queue.append(neighbor)
    return result

if __name__ == "__main__":
    graph = {
        'A': ['B', 'C'],
```

```
        'B': ['D', 'E'],
        'C': ['F'],
        'D': [],
        'E': ['F'],
        'F': []
    }
    start_node = 'A'
    print("BFS Traversal:", bfs(graph, start_node))
```

```
BFS Traversal: ['A', 'B', 'C', 'D', 'E', 'F']
```

8. Implement DFS using a Stack

[1]:
```
def dfs_stack(graph, start_node):
    visited = set()
    stack = [start_node]
    result = []
    while stack:
        current_node = stack.pop()
        if current_node not in visited:
            visited.add(current_node)
            result.append(current_node)

            for neighbor in reversed(graph[current_node]): #neighbors are added␣
 ↪to the stack in reverse order, so they are processed in the original order␣
 ↪during traversal
                if neighbor not in visited:
                    stack.append(neighbor)

    return result
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
start_node = 'A'
print("DFS Order:", dfs_stack(graph, start_node))
```

```
DFS Order: ['A', 'B', 'D', 'E', 'F', 'C']
```

9. The diameter of an unweighted tree is the number of edges between the farthest nodes. Consider a tree with n nodes and m edges. Give a program to find the diameter in O(n) time.

[10]:
```
from collections import deque, defaultdict
```

```python
# BFS function (from the first code snippet)
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    distances = {start: 0}  # Track distances for BFS
    farthest_node = start

    while queue:
        node = queue.popleft()
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                distances[neighbor] = distances[node] + 1 #Update the distance for this neighbor.
                if distances[neighbor] > distances[farthest_node]: #Update the farthest nodE
                    farthest_node = neighbor

    return farthest_node, distances[farthest_node]

def find_tree_diameter(n, edges):
    if n == 1:
        return 0  #single-node tree has a diameter of 0.

    #building the adjacency list for the tree
    tree = defaultdict(list)
    for u, v in edges:
        tree[u].append(v)
        tree[v].append(u)

    # Step 1: Find the farthest node from an arbitrary node (e.g., node 1)
    farthest_node_from_start, _ = bfs(tree, 1)  #gives farthest node(here 4) and distance

    # Step 2: Find the farthest node from the node found in step 1
    a, diameter = bfs(tree, farthest_node_from_start)   #farthest node found (here 3) and tree's diameter

    return diameter

# Example usage:
n = 5
edges = [
    (1, 2),
    (2, 4),
    (2, 5)
```

```
]
print("Diameter of the tree:", find_tree_diameter(n, edges))
```

```
Diameter of the tree: 2
```

10. Write a program to detect cycles in a graph using bfs.

[6]:
```
from collections import defaultdict, deque

def has_cycle_bfs(n, edges):
    # Build the adjacency list
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)  # Since the graph is undirected

    visited = set()

    def bfs(start):
        queue = deque([(start, -1)])  # (current_node, parent_node)
        visited.add(start)

        while queue:
            current, parent = queue.popleft()

            for neighbor in graph[current]:
                if neighbor not in visited:  #If the node hasn't been visited␣
↪yet (it might be in a disconnected component of the graph), we start BFS␣
↪from that node
                    visited.add(neighbor)
                    queue.append((neighbor, current))
                elif neighbor != parent:  # Cycle detected
                    return True

        return False

    # Check for cycles in each component of the graph
    for node in range(1, n + 1):  # Assuming nodes are 1-indexed
        if node not in visited:
            if bfs(node):
                return True

    return False


# Example Usage:
n = 5
edges = [
```

```
    (1, 2),
    (1, 3),
    (2, 4),
    (3, 4),  # This edge introduces a cycle
    (4, 5)
]

if has_cycle_bfs(n, edges):
    print("The graph has a cycle.")
else:
    print("The graph is acyclic.")
```

```
The graph has a cycle.
```

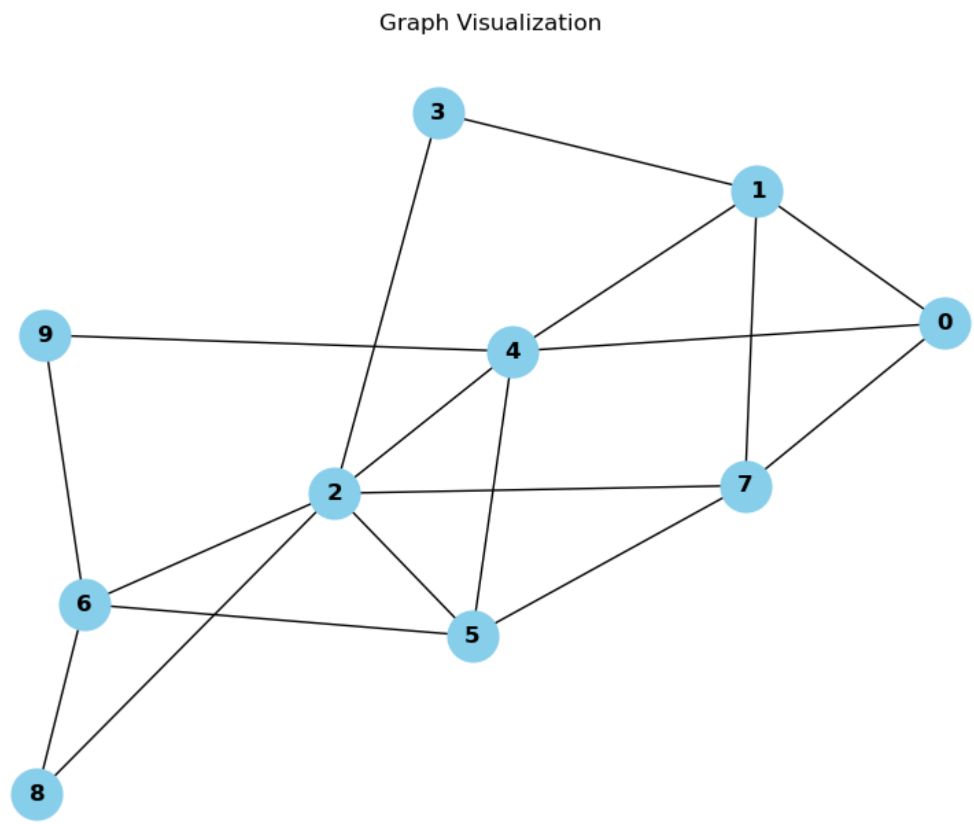
11. The NetworkX package is a comprehensive Python library for the creation, manipulation, and study of complex networks (graphs). It provides tools that are particularly useful for network analysis, visualization, and graph algorithms.

a. Install NetworkX, and matplotli

[4]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

b. Create and draw the following graph.

[11]:
```
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])

edges = [
    (1, 0), (1, 3), (1, 7), (1,4),
    (2, 3), (2, 7), (2, 5), (2, 8), (2,6), (2,4),
    (3, 2),(3,1),
    (4, 0), (4, 5), (4, 9), (4, 2),(4,1),
    (5, 6), (5,2), (5,7),(5,4),
    (6, 8), (6, 9),(6,2),(6,5),
    (7,0)
]
G.add_edges_from(edges)

plt.figure(figsize=(8, 6))
nx.draw(G, with_labels=True, node_color="skyblue", node_size=700,
  ↪font_weight="bold")
plt.title("Graph Visualization")
plt.show()
```




c. Find the following using built-in functions in NetworkX.

d. Degree of each node

[12]:
```
print("Degree of each node:")
for node, degree in G.degree():
    print(f"Node {node}: {degree}")
```

```
Degree of each node:
Node 1: 4
Node 2: 6
Node 3: 2
Node 4: 5
Node 5: 4
Node 6: 4
Node 7: 4
Node 8: 2
Node 9: 2
Node 0: 3
```

ii. BFS

[15]:
```python
bfs_traversal = list(nx.bfs_tree(G, source=1).nodes())
print("BFS Traversal Order:", bfs_traversal)
```

```
BFS Traversal Order: [1, 0, 3, 7, 4, 2, 5, 9, 8, 6]
```

iii. DFS

[19]:
```python
dfs_traversal = list(nx.dfs_preorder_nodes(G, source=1))
print("DFS Traversal Order:", dfs_traversal)
```

```
DFS Traversal Order: [1, 0, 4, 2, 3, 7, 5, 6, 8, 9]
```

iv. Diameter of the graph

[16]:
```python
if nx.is_connected(G):  #checks if a graph G is connected
    diameter = nx.diameter(G)
    print(f"\nDiameter of the graph: {diameter}")
else:
    print("\nGraph is not connected. Diameter cannot be calculated.")
```

```

Diameter of the graph: 3
```

v. Shortest path between two given nodes

[17]:
```python
source, target = 1, 9
if nx.has_path(G, source, target):  #checks if there is any path between source␣
↪and target
    shortest_path = nx.shortest_path(G, source=source, target=target)
    print(f"\nShortest path between node {source} and {target}:␣
↪{shortest_path}")
else:
    print(f"\nNo path exists between node {source} and {target}.")
```

```

Shortest path between node 1 and 9: [1, 4, 9]
```

vi. Shortest path between all pairs of nodes

[18]:
```python
print("\nShortest path between all pairs of nodes:")
shortest_paths = dict(nx.all_pairs_shortest_path(G))
for source, paths in shortest_paths.items():
    print(f"From node {source}:")
    for target, path in paths.items():
        print(f"  To node {target}: {path}")
```

```

Shortest path between all pairs of nodes:
From node 1:
  To node 1: [1]
  To node 0: [1, 0]
```

```
    To node 3: [1, 3]
    To node 7: [1, 7]
    To node 4: [1, 4]
    To node 2: [1, 3, 2]
    To node 5: [1, 7, 5]
    To node 9: [1, 4, 9]
    To node 8: [1, 3, 2, 8]
    To node 6: [1, 3, 2, 6]
  From node 2:
    To node 2: [2]
    To node 3: [2, 3]
    To node 7: [2, 7]
    To node 5: [2, 5]
    To node 8: [2, 8]
    To node 6: [2, 6]
    To node 4: [2, 4]
    To node 1: [2, 3, 1]
    To node 0: [2, 7, 0]
    To node 9: [2, 6, 9]
  From node 3:
    To node 3: [3]
    To node 1: [3, 1]
    To node 2: [3, 2]
    To node 0: [3, 1, 0]
    To node 7: [3, 1, 7]
    To node 4: [3, 1, 4]
    To node 5: [3, 2, 5]
    To node 8: [3, 2, 8]
    To node 6: [3, 2, 6]
    To node 9: [3, 1, 4, 9]
  From node 4:
    To node 4: [4]
    To node 1: [4, 1]
    To node 2: [4, 2]
    To node 0: [4, 0]
    To node 5: [4, 5]
    To node 9: [4, 9]
    To node 3: [4, 1, 3]
    To node 7: [4, 1, 7]
    To node 8: [4, 2, 8]
    To node 6: [4, 2, 6]
  From node 5:
    To node 5: [5]
    To node 2: [5, 2]
    To node 4: [5, 4]
    To node 6: [5, 6]
    To node 7: [5, 7]
    To node 3: [5, 2, 3]
```

```
    To node 8: [5, 2, 8]
    To node 1: [5, 4, 1]
    To node 0: [5, 4, 0]
    To node 9: [5, 4, 9]
  From node 6:
    To node 6: [6]
    To node 2: [6, 2]
    To node 5: [6, 5]
    To node 8: [6, 8]
    To node 9: [6, 9]
    To node 3: [6, 2, 3]
    To node 7: [6, 2, 7]
    To node 4: [6, 2, 4]
    To node 1: [6, 2, 3, 1]
    To node 0: [6, 2, 7, 0]
  From node 7:
    To node 7: [7]
    To node 1: [7, 1]
    To node 2: [7, 2]
    To node 5: [7, 5]
    To node 0: [7, 0]
    To node 3: [7, 1, 3]
    To node 4: [7, 1, 4]
    To node 8: [7, 2, 8]
    To node 6: [7, 2, 6]
    To node 9: [7, 1, 4, 9]
  From node 8:
    To node 8: [8]
    To node 2: [8, 2]
    To node 6: [8, 6]
    To node 3: [8, 2, 3]
    To node 7: [8, 2, 7]
    To node 5: [8, 2, 5]
    To node 4: [8, 2, 4]
    To node 9: [8, 6, 9]
    To node 1: [8, 2, 3, 1]
    To node 0: [8, 2, 7, 0]
  From node 9:
    To node 9: [9]
    To node 4: [9, 4]
    To node 6: [9, 6]
    To node 1: [9, 4, 1]
    To node 2: [9, 4, 2]
    To node 0: [9, 4, 0]
    To node 5: [9, 4, 5]
    To node 8: [9, 6, 8]
    To node 3: [9, 4, 1, 3]
    To node 7: [9, 4, 1, 7]
```

```
From node 0:
  To node 0: [0]
  To node 1: [0, 1]
  To node 4: [0, 4]
  To node 7: [0, 7]
  To node 3: [0, 1, 3]
  To node 2: [0, 4, 2]
  To node 5: [0, 4, 5]
  To node 9: [0, 4, 9]
  To node 8: [0, 4, 2, 8]
  To node 6: [0, 4, 2, 6]
```

12. Consider a chess board in which a knight is currently located at position (i, j) where i and j are row and column indices respectively. Given another position (a, b) in the chess board, write a program to compute minimum number of moves required to move the knight from position (i, j) to (a, b).

[4]:
```python
from collections import deque

def min_knight_moves(n, m, start, end):
    # Possible moves of a knight
    directions = [
        (2, 1), (2, -1), (-2, 1), (-2, -1),
        (1, 2), (1, -2), (-1, 2), (-1, -2)
    ]

    # BFS initialization
    queue = deque([(start[0], start[1], 0)])  # (current_row, current_col,␣
 ↪moves)
    visited = set()
    visited.add((start[0], start[1]))

    while queue:
        x, y, moves = queue.popleft()

        # If we reach the target position
        if (x, y) == (end[0], end[1]):
            return moves

        # Explore all possible knight moves
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 1 <= nx <= n and 1 <= ny <= m and (nx, ny) not in visited:
                visited.add((nx, ny))
                queue.append((nx, ny, moves + 1))

    return -1  # Target is unreachable (shouldn't happen for valid inputs)
```

```python
# Example usage:
n = 8  # Chessboard size (8x8)
m = 8
start = (1, 1)  # Starting position (i, j)
end = (8, 8)    # Target position (a, b)

print("Minimum moves required:", min_knight_moves(n, m, start, end))
```

```
Minimum moves required: 6
```

6. Implement Strassen's algorithm to multiply two nxn matrices.

[ ]:
```python
import numpy as np

def add_matrices(A, B):
    return [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in␣
 ↪range(len(A))]

def subtract_matrices(A, B):
    return [[A[i][j] - B[i][j] for j in range(len(A[0]))] for i in␣
 ↪range(len(A))]

def split(matrix):
    n = len(matrix)
    mid = n // 2
    A11 = [[matrix[i][j] for j in range(mid)] for i in range(mid)]
    A12 = [[matrix[i][j] for j in range(mid, n)] for i in range(mid)]
    A21 = [[matrix[i][j] for j in range(mid)] for i in range(mid, n)]
    A22 = [[matrix[i][j] for j in range(mid, n)] for i in range(mid, n)]
    return A11, A12, A21, A22

def combine(C11, C12, C21, C22):
    n = len(C11)
    new_matrix = [[0] * (2 * n) for _ in range(2 * n)]
    for i in range(n):
        for j in range(n):
            new_matrix[i][j] = C11[i][j]
            new_matrix[i][j + n] = C12[i][j]
            new_matrix[i + n][j] = C21[i][j]
            new_matrix[i + n][j + n] = C22[i][j]
    return new_matrix

def strassen(A, B):
    n = len(A)

    # Base case: 1x1 matrix multiplication
    if n == 1:
        return [[A[0][0] * B[0][0]]]
```

```
    # Splitting
    A11, A12, A21, A22 = split(A)
    B11, B12, B21, B22 = split(B)

    # Calculate the 7 products (M1 to M7) using recursive calls
    M1 = strassen(add_matrices(A11, A22), add_matrices(B11, B22))  # M1 = (A11 + A22)(B11 + B22)
    M2 = strassen(add_matrices(A21, A22), B11)                     # M2 = (A21 + A22)B11
    M3 = strassen(A11, subtract_matrices(B12, B22))                # M3 = A11(B12 - B22)
    M4 = strassen(A22, subtract_matrices(B21, B11))                # M4 = A22(B21 - B11)
    M5 = strassen(add_matrices(A11, A12), B22)                     # M5 = (A11 + A12)B22
    M6 = strassen(subtract_matrices(A21, A11), add_matrices(B11, B12))  # M6 = (A21 - A11)(B11 + B12)
    M7 = strassen(subtract_matrices(A12, A22), add_matrices(B21, B22))  # M7 = (A12 - A22)(B21 + B22)

    # Compute the 4 resulting submatrices
    C11 = add_matrices(subtract_matrices(add_matrices(M1, M4), M5), M7)  # C11 = M1 + M4 - M5 + M7
    C12 = add_matrices(M3, M5)                                            # C12 = M3 + M5
    C21 = add_matrices(M2, M4)                                            # C21 = M2 + M4
    C22 = add_matrices(subtract_matrices(add_matrices(M1, M3), M2), M6)  # C22 = M1 + M3 - M2 + M6

    # Combine the 4 submatrices into the final result
    return combine(C11, C12, C21, C22)


A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

result = strassen(A, B)
print("Resultant Matrix:")
for row in result:
    print(row)
```

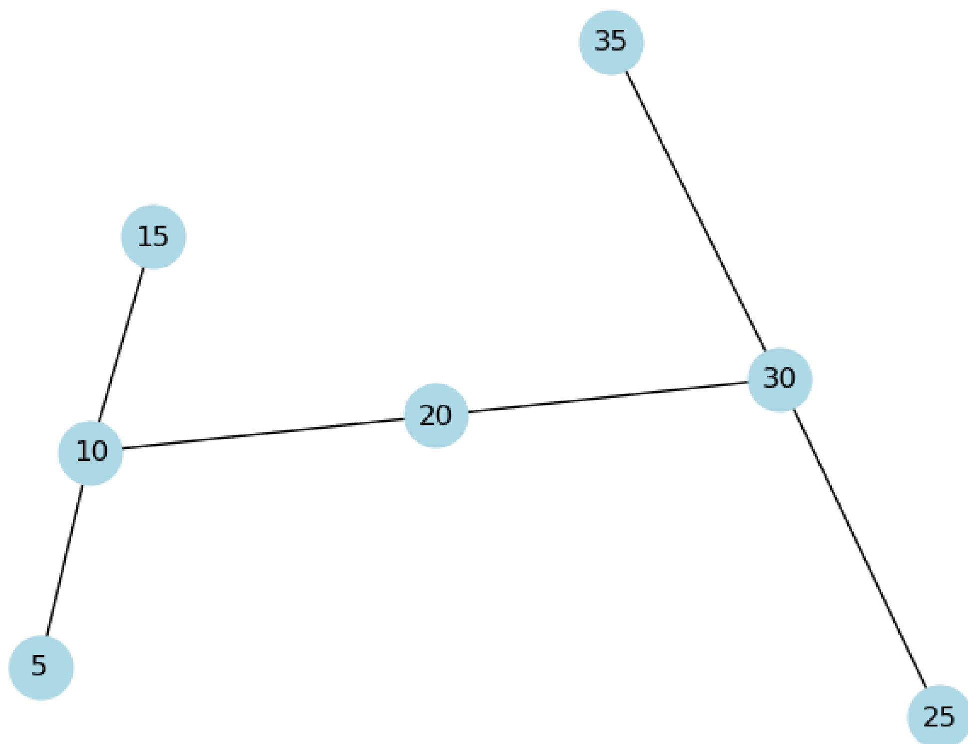
13. Construct a binary search tree using NetworkX, and demonstrate insertion and traversal (in-order, pre-order, post-order).

[7]:
```python
import networkx as nx
import matplotlib.pyplot as plt

class BinarySearchTree:
    def __init__(self):
        self.tree = nx.DiGraph()  # Directed graph to represent BST
        self.root = None

    def insert(self, value):
        """Insert a value into the BST."""
        if self.root is None:
            self.root = value
            self.tree.add_node(value)  # Add root node
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, current, value):
        """Recursive helper function for inserting a value."""
        if value < current:  # Should go to the left subtree
            left_child = next((child for child in self.tree.successors(current)␣
↪if child < current), None)
            if left_child is None:
                self.tree.add_edge(current, value)
            else:
                self._insert_recursive(left_child, value)
        elif value > current:  # Should go to the right subtree
            right_child = next((child for child in self.tree.
↪successors(current) if child > current), None)
            if right_child is None:
                self.tree.add_edge(current, value)
            else:
                self._insert_recursive(right_child, value)

    def in_order(self, current=None):
        """Perform in-order traversal."""
        if current is None:
            current = self.root
        result = []
        left_child = next((child for child in self.tree.successors(current) if␣
↪child < current), None)
        if left_child:
            result.extend(self.in_order(left_child))
        result.append(current)
        right_child = next((child for child in self.tree.successors(current) if␣
↪child > current), None)
        if right_child:
            result.extend(self.in_order(right_child))
```

```
        return result

    def pre_order(self, current=None):
        """Perform pre-order traversal."""
        if current is None:
            current = self.root
        result = [current]
        left_child = next((child for child in self.tree.successors(current) if child < current), None)
        if left_child:
            result.extend(self.pre_order(left_child))
        right_child = next((child for child in self.tree.successors(current) if child > current), None)
        if right_child:
            result.extend(self.pre_order(right_child))
        return result

    def post_order(self, current=None):
        """Perform post-order traversal."""
        if current is None:
            current = self.root
        result = []
        left_child = next((child for child in self.tree.successors(current) if child < current), None)
        if left_child:
            result.extend(self.post_order(left_child))
        right_child = next((child for child in self.tree.successors(current) if child > current), None)
        if right_child:
            result.extend(self.post_order(right_child))
        result.append(current)
        return result

    def visualize(self):
        """Visualize the tree using NetworkX's spring layout."""
        pos = nx.spring_layout(self.tree)  # Spring layout for general visualization
        nx.draw(self.tree, pos, with_labels=True, arrows=False, node_size=700, node_color="lightblue")
        plt.show()

# Example usage:
bst = BinarySearchTree()
values = [20, 10, 30, 5, 15, 25, 35]
for value in values:
    bst.insert(value)
```

```python
print("In-order traversal:", bst.in_order())
print("Pre-order traversal:", bst.pre_order())
print("Post-order traversal:", bst.post_order())

# Visualize the tree
bst.visualize()
```

```
In-order traversal: [5, 10, 15, 20, 25, 30, 35]
Pre-order traversal: [20, 10, 5, 15, 30, 25, 35]
Post-order traversal: [5, 15, 10, 25, 35, 30, 20]
```




14.Implement a partially persistent array

[8]:
```python
class PersistentArray:
    def __init__(self, size):
        """Initialize a partially persistent array of given size."""
        self.size = size
        self.versions = [{}]  # List of dictionaries to store changes per␣
 ↪version
        self.current_array = [None] * size  # Initial array state
```

```
    def update(self, index, value):
        """
        Update the value at the specified index.
        Create a new version with the updated array.
        """
        if index < 0 or index >= self.size:
            raise IndexError("Index out of bounds")

        # Update current array
        self.current_array[index] = value

        # Record the update in the new version
        new_version = self.versions[-1].copy()  # Copy previous version's␣
↪changes
        new_version[index] = value
        self.versions.append(new_version)  # Save the new version

    def get(self, version, index):
        """
        Get the value at a given index for a specific version.
        If the version is not explicitly updated at that index,
        fallback to the nearest previous version.
        """
        if version < 0 or version >= len(self.versions):
            raise IndexError("Version out of bounds")
        if index < 0 or index >= self.size:
            raise IndexError("Index out of bounds")

        # Find the value for the given version
        for v in range(version, -1, -1):  # Traverse back to find the latest␣
↪update
            if index in self.versions[v]:
                return self.versions[v][index]

        return None  # If no update found, return None

    def get_latest_version(self):
        """Return the latest version of the array."""
        return self.current_array.copy()

# Example usage
array = PersistentArray(5)

# Initial updates
array.update(0, 10)  # Update index 0 to 10 (Version 1)
array.update(1, 20)  # Update index 1 to 20 (Version 2)
```

```python
array.update(2, 30)  # Update index 2 to 30 (Version 3)

# Access versions
print("Value at index 1, version 1:", array.get(1, 1))  # Access index 1,␣
↪version 1 → 20
print("Value at index 2, version 2:", array.get(2, 2))  # Access index 2,␣
↪version 2 → 30
print("Value at index 0, version 0:", array.get(0, 0))  # Access index 0,␣
↪version 0 → 10

# Latest array state
print("Latest array state:", array.get_latest_version())
```

```
Value at index 1, version 1: None
Value at index 2, version 2: None
Value at index 0, version 0: None
Latest array state: [10, 20, 30, None, None]
```

15. Implement backtracking algorithm for NQueens problem

[9]:
```python
def print_solution(board):
    """Print the chessboard solution."""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")

def is_safe(board, row, col, n):
    """Check if a queen can be placed at board[row][col]."""
    # Check the column for any queen
    for i in range(row):
        if board[i][col]:
            return False

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

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

    return True

def solve_n_queens(board, row, n):
    """Use backtracking to solve the N-Queens problem."""
    if row == n:
```

```python
        # If all queens are placed, print the solution
        print_solution(board)
        return True

    res = False
    for col in range(n):
        if is_safe(board, row, col, n):
            # Place a queen
            board[row][col] = True
            # Recur to place the next queen
            res = solve_n_queens(board, row + 1, n) or res
            # Backtrack
            board[row][col] = False

    return res

def n_queens(n):
    """Main function to solve N-Queens problem for size n."""
    # Initialize the chessboard with False (no queens placed)

    if not solve_n_queens(board, 0, n):
        print("No solution exists.")
    else:
        print("Solutions found.")

# Example usage:
n = 8  # Change this value for different board sizes
n_queens(n)
```

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .


Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
```

```
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .

Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .

. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .

. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
Q . . . . . . .
. . . . . . . Q
```

```
. . . . . Q . .
. . Q . . . . .
```

```
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . . . Q
. . Q . . . . .
. . . . Q . . .
```

```
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . Q . . .
```

```
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
```

```
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

```
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
```

```
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
```

```
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
```

```
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
```

```
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
```

```
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
```

```
. . Q . . . . .
. . . . Q . . .
```

```
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .


. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .


. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .


. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .


. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
```

```
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
```

```
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
```

```
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

```
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
```

```
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
. . . . . Q . .
```

```
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
Q . . . . . . .
. . . . Q . . .
```

```
. . Q . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
```

```
. . . Q . . . .
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
```

```
. . . Q . . . .
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
```

```
. . Q . . . . .
. . . . . . Q .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
```

```
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
```

```
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
```

```
. . . Q . . . .
. . . . . Q . .
Q . . . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
```

```
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
```

```
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
```

```
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

```
. . . Q . . . .
. . . . . . Q .
```

```
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .
```

```
. . . Q . . . .
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
```

```
. . . Q . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
```

```
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
```

```
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

```
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
```

```
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
```

```
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
```

```
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

```
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
```

```
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .

. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . . . Q
. . Q . . . . .

. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .

. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .

. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . . . Q
```

```
. . . . . Q . .
. . . Q . . . .
```

```
. . . . Q . . .
. . Q . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
```

```
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
```

```
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
```

```
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
```

```
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
```

```
. . Q . . . . .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q

. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .

. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. Q . . . . . .

. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .

. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .

. . . . . Q . .
Q . . . . . . .
```

```
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
```

```
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
```

```
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .
. . Q . . . . .
```

```
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
```

```
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
```

```
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
```

```
. . . . . Q . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
```

```
. . . . . Q . .
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
```

```
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
```

```
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
```

```
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
Q . . . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
```

```
. . . . . Q . .
. . . Q . . . .
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
```

```
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
```

```
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
```

```
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. Q . . . . . .
```

```
. . . . Q . . .
. . Q . . . . .
```

```
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
```

```
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
```

```
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
```

```
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .
```

```
. . . . . . Q .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
```

```
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .


. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .
. . . Q . . . .


. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .


. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .


. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .


. . . . . . . Q
. Q . . . . . .
```

```
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .


. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .


. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .


. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .


Solutions found.
```

16.Implement the solution of NQueens problem using persistent ds. Use bakers method.

[10]:
```python
class PersistentBoard:
    """A class to represent a persistent board state."""
    def __init__(self, size, queens=None):
        self.size = size
        self.queens = queens if queens else []
```

```python
    def add_queen(self, row, col):
        """Return a new board state with a queen added."""
        new_queens = self.queens + [(row, col)]
        return PersistentBoard(self.size, new_queens)

    def is_valid(self, row, col):
        """Check if placing a queen at (row, col) is valid."""
        for r, c in self.queens:
            # Check column and diagonal conflicts
            if c == col or abs(row - r) == abs(col - c):
                return False
        return True

    def __str__(self):
        """String representation of the board for visualization."""
        board = [["." for _ in range(self.size)] for _ in range(self.size)]
        for r, c in self.queens:
            board[r][c] = "Q"
        return "\n".join("".join(row) for row in board)


def solve_nqueens(size):
    """Solve the N-Queens problem using Baker's method with persistent data␣
 ↪structure."""
    def backtrack(row, board, solutions):
        if row == size:
            # Found a valid solution
            solutions.append(board)
            return

        for col in range(size):
            if board.is_valid(row, col):
                # Add a queen and continue
                new_board = board.add_queen(row, col)
                backtrack(row + 1, new_board, solutions)

    solutions = []
    initial_board = PersistentBoard(size)
    backtrack(0, initial_board, solutions)
    return solutions


# Example usage
n = 4
solutions = solve_nqueens(n)
print(f"Number of solutions for {n}-Queens: {len(solutions)}")
for i, solution in enumerate(solutions, start=1):
```

```python
    print(f"\nSolution {i}:\n{solution}")
```

```
Number of solutions for 4-Queens: 2

Solution 1:
.Q..
…Q
Q…
..Q.

Solution 2:
..Q.
Q…
…Q
.Q..
```

17. Implement Kruskal's algorithm

[1]:
```python
class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]  # Each vertex is its own parent initially
        self.rank = [0] * n  # Rank to keep the tree flat

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

    def union(self, u, v):
        # Union by rank
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            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 kruskal_algorithm(edges, n):
    # Sort edges by weight
    edges.sort(key=lambda edge: edge[2])  # Sort by the 3rd element (weight)
```

```
    disjoint_set = DisjointSet(n)  # Create a disjoint-set for the vertices
    mst_edges = []  # To store edges of the MST
    total_cost = 0  # To store the total weight of the MST

    for u, v, weight in edges:
        # Check if adding this edge forms a cycle
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            mst_edges.append((u, v, weight))
            total_cost += weight

    return mst_edges, total_cost


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

mst_edges, total_cost = kruskal_algorithm(edges, n)

print("Edges in MST:", mst_edges)
print("Total cost of MST:", total_cost)
```

```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total cost of MST: 19
```

[1]:
```
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices  # Number of vertices
        self.edges = []           # List to store edges (u, v, weight)

    def add_edge(self, u, v, weight):
        """Add an edge to the graph."""
        self.edges.append((weight, u, v))

    def find(self, parent, i):
        """Find the parent of a node using path compression."""
        if parent[i] == i:
            return i
        parent[i] = self.find(parent, parent[i])  # Path compression
```

```
        return parent[i]

    def union(self, parent, rank, x, y):
        """Perform union of two sets."""
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    def kruskal_mst(self):
        """Find and return the Minimum Spanning Tree using Kruskal's algorithm.
 ↪"""
        # Step 1: Sort edges by weight
        self.edges.sort()

        # Initialize parent and rank arrays
        parent = []
        rank = []
        for node in range(self.vertices):
            parent.append(node)
            rank.append(0)

        mst = []  # List to store the MST
        mst_cost = 0  # Cost of the MST

        # Step 2: Add edges to MST while avoiding cycles
        for weight, u, v in self.edges:
            root_u = self.find(parent, u)
            root_v = self.find(parent, v)

            # Check if including this edge creates a cycle
            if root_u != root_v:
                mst.append((u, v, weight))
                mst_cost += weight
                self.union(parent, rank, root_u, root_v)

        return mst, mst_cost


# Example usage
if __name__ == "__main__":
```

```
    # Create a graph with 4 vertices
    g = Graph(4)

    # Add edges: (u, v, weight)
    g.add_edge(0, 1, 10)
    g.add_edge(0, 2, 6)
    g.add_edge(0, 3, 5)
    g.add_edge(1, 3, 15)
    g.add_edge(2, 3, 4)

    # Find and print the MST
    mst, mst_cost = g.kruskal_mst()
    print("Edges in the MST:")
    for u, v, weight in mst:
        print(f"({u}, {v}) - {weight}")
    print("Cost of the MST:", mst_cost)
```

```
Edges in the MST:
(2, 3) - 4
(0, 3) - 5
(0, 1) - 10
Cost of the MST: 19
```

18. Implement Prims algorithm

[2]:
```
import sys

def prims_algorithm(graph, n):
    # Initialize MST set
    selected = [False] * n
    selected[0] = True  # Starting with the first vertex
    edges_count = 0
    total_cost = 0

    # Store the result
    mst_edges = []

    while edges_count < n - 1:
        min_weight = sys.maxsize
        x, y = 0, 0

        # Find the minimum edge that connects the tree to another vertex
        for u in range(n):
            if selected[u]:
                for v in range(n):
                    if not selected[v] and graph[u][v] and graph[u][v] <␣
↪min_weight:
                        min_weight = graph[u][v]
```

```
                        x, y = u, v

        # Add this edge to the MST
        selected[y] = True
        mst_edges.append((x, y, min_weight))
        total_cost += min_weight
        edges_count += 1

    return mst_edges, total_cost

# Example graph as an adjacency matrix
# graph[i][j] holds the weight of the edge between i and j
graph = [
    [0, 10, 6, 5],
    [10, 0, 15, 0],
    [6, 15, 0, 4],
    [5, 0, 4, 0]
]

n = 4  # Number of vertices

mst_edges, total_cost = prims_algorithm(graph, n)

print("Edges in MST:", mst_edges)
print("Total cost of MST:", total_cost)
```

```
Edges in MST: [(0, 3, 5), (3, 2, 4), (0, 1, 10)]
Total cost of MST: 19
```

20. Implement 1D range query tree.

[16]:
```
class SegmentTree:
    def __init__(self, arr):  # Corrected __init__
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, l, r):
        if l == r:  # Leaf node
            self.tree[node] = arr[l]
        else:
            mid = (l + r) // 2
            self.build(arr, 2 * node + 1, l, mid)
            self.build(arr, 2 * node + 2, mid + 1, r)
            self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def update(self, idx, value, node, l, r):
        if l == r:
```

```python
            self.tree[node] = value
        else:
            mid = (l + r) // 2
            if idx <= mid:
                self.update(idx, value, 2 * node + 1, l, mid)
            else:
                self.update(idx, value, 2 * node + 2, mid + 1, r)
            self.tree[node] = self.tree[2 * node + 1] + self.tree[2 * node + 2]

    def query(self, L, R, node, l, r):
        if R < l or L > r:  # No overlap
            return 0
        if L <= l and r <= R:  # Complete overlap
            return self.tree[node]
        mid = (l + r) // 2
        return self.query(L, R, 2 * node + 1, l, mid) + self.query(L, R, 2 *
 ↪node + 2, mid + 1, r)

    def range_sum(self, L, R):
        return self.query(L, R, 0, 0, self.n - 1)

    def update_value(self, idx, value):
        self.update(idx, value, 0, 0, self.n - 1)


# Example Usage
arr = [1, 3, 5, 7, 9, 11]
st = SegmentTree(arr)

print("Sum of range (1, 3):", st.range_sum(1, 3))  # Output: 15
st.update_value(1, 10)  # Update index 1 to 10
print("After update, sum of range (1, 3):", st.range_sum(1, 3))  # Output: 22
```

```
Sum of range (1, 3): 15
After update, sum of range (1, 3): 22
```