def insertion\_sort(arr):  
 for i in range(1, len(arr)):  
 key = arr[i]  
 j = i - 1  
  
 while j >= 0 and arr[j] > key:  
 arr[j + 1] = arr[j]  
 j -= 1  
  
 arr[j + 1] = key  
  
# Example unsorted list of numbers  
numbers = [5, 2, 9, 1, 5, 6]  
  
# Print the original list  
print("Original list:", numbers)  
  
# Perform Insertion Sort  
insertion\_sort(numbers)  
  
# Print the sorted list  
print("Sorted list:", numbers )

2.

def dfs\_recursive(graph, vertex, visited):  
 # Mark the current vertex as visited  
 visited[vertex] = True  
 print(vertex, end=' ')  
  
 # Recursive DFS on all adjacent vertices  
 for neighbor in graph[vertex]:  
 if not visited[neighbor]:  
 dfs\_recursive(graph, neighbor, visited)  
  
# Example graph represented as an adjacency list  
# The graph has 6 vertices: 0, 1, 2, 3, 4, 5  
graph = {  
 0: [1, 3],  
 1: [0,2,4],  
 2: [0, 1,4],  
 3: [4,0],  
 4: [2,1,3],  
 5: []  
}  
  
# Number of vertices in the graph  
num\_vertices = len(graph)  
  
# Initialize the visited array  
visited = [False] \* num\_vertices  
  
# Perform recursive DFS starting from vertex 0  
print("DFS Traversal:")  
dfs\_recursive(graph, 0, visited)

3.

def count\_inversions(arr):  
 n = len(arr)  
 inversions = 0  
  
 for i in range(n):  
 for j in range(i + 1, n):  
 if arr[i] > arr[j]:  
 inversions += 1  
  
 return inversions  
  
  
# Example usage  
arr = [ [1,2,6,4,5,8,7,3] , [1,2,3,4,5,6,7,8] ,[8,7,4,5,6,3,2,1] ]  
inversions = count\_inversions(arr)  
print("Number of inversions:", inversions)

4.

def quicksort(arr):  
 if len(arr) <= 1:  
 return arr  
  
 pivot = arr[len(arr) // 2]  
 left = [x for x in arr if x < pivot]  
 middle = [x for x in arr if x == pivot]  
 right = [x for x in arr if x > pivot]  
  
 return quicksort(left) + middle + quicksort(right)  
  
# Example unsorted list of numbers  
numbers = [38, 27, 43, 3, 9, 82, 10]  
  
# Perform Quicksort  
sorted\_numbers = quicksort(numbers)  
  
# Print the sorted list  
print("Sorted list:", sorted\_numbers)

o/p

Sorted list: [3, 9, 10, 27, 38, 43, 82]

5.

import heapq  
  
def prim(graph):  
 MST = set()  
 start\_vertex = next(iter(graph))  
 MST.add(start\_vertex)  
 min\_cost = 0  
  
 priority\_queue = [(cost, start\_vertex, vertex) for vertex, cost in graph[start\_vertex].items()]  
 heapq.heapify(priority\_queue)  
  
 while priority\_queue:  
 cost, u, v = heapq.heappop(priority\_queue)  
  
 if v not in MST:  
 MST.add(v)  
 min\_cost += cost  
  
 for vertex, edge\_cost in graph[v].items():  
 if vertex not in MST:  
 heapq.heappush(priority\_queue, (edge\_cost, v, vertex))  
  
 return min\_cost  
  
# Example graph represented as an adjacency list with weights  
graph = {  
 'A': {'B': 2, 'C': 4},  
 'B': {'A': 2, 'C': 1, 'D': 7},  
 'C': {'A': 4, 'B': 1, 'D': 3},  
 'D': {'B': 7, 'C': 3}  
}  
  
# Calculate the minimum cost using Prim's algorithm  
min\_cost = prim(graph)  
  
print("Minimum Cost:", min\_cost)

**Minimum Cost: 6**

6.

def subset\_sum(nums, target\_sum):  
 dp = [False] \* (target\_sum + 1)  
 dp[0] = True  
  
 for num in nums:  
 for j in range(target\_sum, num - 1, -1):  
 dp[j] = dp[j] or dp[j - num]  
  
 return dp[target\_sum]  
  
  
# Example usage  
nums = [3, 34, 4, 12, 5, 2]  
target\_sum = 9  
  
if subset\_sum(nums, target\_sum):  
 print("There is a subset with the given sum.")  
else:  
 print("No subset with the given sum exists.")

o/p

There is a subset with the given sum.