BCA Semester–V

DisciplineSpecific Course (DSCC-10)

Course Title: Practical inDesign and Analysis of Algorithms
Course Code:055BCA012

Type of Course	Theory /Practical	Credits	Instruction hour per week	Total No.of Lectures/Hours /Semester	Duration of Exam	Formative Assessment Marks	Summative assessment Marks	Total Marks
DSCC-10	**Practical**	**02**	**04**	**56hrs.**	**3hrs.**	**25**	**25**	**50**

Course Outcomes (COs):Attheend ofthecourse,studentswillbeableto:

CO1: Able to calculate complexity of an algorithm.
CO2: Select appropriate design techniques to solve real world problems.
CO3: Apply the dynamic programming technique to solve the problems.

Program Nos	Programs	56.hrs/ sem
1	Write a program to sort a list of N elements using Selection Sort Technique.	
2	Write a program to perform Travelling Sales man Problem	
3	Write program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.	
4	Write program to implement the DFS and BFS algorithm for a graph.	
5	Write a program to find minimum and maximum value in an array using divide and conquer.	
6	Write a test program to implement Divide and Conquer Strategy. Eg: Quick sort algorithm for sorting list of integers in ascending order.	
7	Write a program to implement Merge sort algorithm for sorting a list of integers in ascending order.	
8	Write C program that accepts the vertices and edges for a graph and stores it as an adjacency matrix.	
9	ImplementfunctiontoprintIn-Degree,Out-Degreeandtodisplaythatadjacencymatrix	
10	Write a program to perform Knapsack Problem using GreedySolution	
11	Write program to implement backtracking algorithm for solving problems like Nqueens.	
12	Write a program to implement the backtracking algorithm for the sum of subsets problem	
13	Write program to implement greedy algorithm for job sequencing with deadlines.	
14	WriteprogramtoimplementDynamicProgrammingalgorithmfortheOptimalBinary Search Tree Problem.	
15	WriteaprogramthatimplementsPrim'salgorithmtogenerateminimumcostspanningTree.	
16	Write a program that implements Kruskal's algorithm to generate minimum cost spanning tree.	

Program 1:

Write a program to sort a list of N elements using Selection Sort Technique.

```
def selection_sort(arr):
    n = len(arr)
    # Traverse through all elements
    for i in range(n - 1):
    # Assume the current element is the minimum
        min_index = i
        # Find the minimum element in remaining unsorted array
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Swap the found minimum element with the first element
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
# Driver code
N = int(input("Enter number of elements: "))
elements = []
print("Enter the elements:")
for _ in range(N):
    elements.append(int(input()))
print("Original List:", elements)
sorted_list = selection_sort(elements)
print("Sorted List:", sorted_list)
```

Program 2:

Write a program to perform Travelling Salesman Problem

```
from itertools import permutations

def calculate_distance(route, distances):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distances[route[i]][route[i + 1]]
    # return to the starting city
    total_distance += distances[route[-1]][route[0]]
    return total_distance

def brute_force_tsp(distances, start):
    n = len(distances)
    cities = [i for i in range(n) if i != start]  # exclude the start city
    min_distance = float('inf')
    shortest_route = None
    print("\nAll Possible Routes and Their Distances:\n")
    for perm in permutations(cities):
        current_route = [start] + list(perm) + [start]  # start and end at chosen city
        current_distance = calculate_distance(current_route, distances)
        print(f"Route {current_route} → Distance = {current_distance}")
        if current_distance < min_distance:
            min_distance = current_distance
            shortest_route = current_route
    return shortest_route, min_distance
```

```
# ---------------- MAIN PROGRAM ----------------

# Read number of cities

n = int(input("Enter number of cities: "))

# Read adjacency matrix

print("Enter the distance matrix row by row (use spaces between values):")

distances = []

for i in range(n):

    row = list(map(int, input(f"Row {i+1}: ").split()))

    distances.append(row)

# Read starting node

start = int(input(f"Enter the starting node (0 to {n-1}): "))

# Run TSP brute force

route, total_distance = brute_force_tsp(distances, start)

print("\nShortest Route:", route)

print("Minimum Distance:", total_distance)
```

Program 3:

Write program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.

```
from itertools import combinations
def knapsack_bruteforce_all_subsets(weights, values, capacity):
    n = len(values)
    max_profit = 0
    best_combination = None
    print(f"\nKnapsack Capacity = {capacity}\n")
    print("All Possible Subsets:")
    # Generate all subsets (including empty set)
    for r in range(0, n+1):
        for subset in combinations(range(n), r):
            total_weight = sum(weights[i] for i in subset)
            total_value = sum(values[i] for i in subset)
            items = [i+1 for i in subset]  # 1-based item numbering
            if total_weight <= capacity:
                status = "Considered"
                if total_value > max_profit:
                    max_profit = total_value
                    best_combination = (items, total_weight, total_value)
            else:
                status = "Not Considered (Exceeds Capacity)"
            print(f"Items: {items}, Weight: {total_weight}, Value: {total_value} --> {status}")
    # Display best solution
    print("\nBest Combination Found:")
    print(f"Items: {best_combination[0]}, Weight: {best_combination[1]}, Profit: {best_combination[2]}")
```

```
    print(f"\nMaximum Profit Achievable = {max_profit} (with Capacity = {capacity})")
    return max_profit


# ---------------- MAIN PROGRAM ----------------
if __name__ == "__main__":
    n = int(input("Enter number of items: "))
    weights = []
    values = []
    print("\nEnter weights and values for each item:")
    for i in range(n):
        w = int(input(f"Weight of item {i+1}: "))
        v = int(input(f"Value of item {i+1}: "))
        weights.append(w)
        values.append(v)
    capacity = int(input("\nEnter knapsack capacity: "))
    knapsack_bruteforce_all_subsets(weights, values, capacity)
```

Program 4:

Write program to implement the DFS and BFS algorithm for a graph.

```
# Depth First Search (DFS) and Breadth First Search (BFS)
# Takes graph as input from the user
from collections import deque
def dfs(graph, root):
    visited = []
    stack = [root]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.append(node)
            stack.extend(graph[node])  # Push neighbors
    return visited
def bfs(graph, root):
    visited = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            queue.extend(graph[node])  # Enqueue neighbors
    return visited



if __name__ == "__main__":
```

```
graph = {}
# Take input from user
n = int(input("Enter number of vertices: "))
print("Enter vertex labels (e.g., A, B, C):")
vertices = [input().strip() for _ in range(n)]
# Initialize adjacency list
for v in vertices:
    graph[v] = []
e = int(input("Enter number of edges: "))
print("Enter edges in format: u v (means edge from u -> v)")
for _ in range(e):
    u, v = input().split()
    graph[u].append(v)   # directed edge
    # For undirected graph, also add: graph[v].append(u)
print("\nGraph (Adjacency List):")
for node, neighbors in graph.items():
    print(f"{node} -> {neighbors}")
# Traversals
start_node = input("\nEnter the starting node for traversal: ").strip()
dfs_result = dfs(graph, start_node)
bfs_result = bfs(graph, start_node)
print("\nDFS Traversal starting from node", start_node, ":")
print(" -> ".join(dfs_result))
print("\nBFS Traversal starting from node", start_node, ":")
print(" -> ".join(bfs_result))
```

Program 5:

Write a program to find minimum and maximum value in an array using divide and conquer.

```
# Function to find minimum and maximum using Divide and Conquer
def find_min_max(arr, low, high):
    # Case 1: Only one element
    if low == high:
        return arr[low], arr[low]
    # Case 2: Two elements
    elif high == low + 1:
        if arr[low] < arr[high]:
            return arr[low], arr[high]
        else:
            return arr[high], arr[low]
    # Case 3: More than two elements
    else:
        mid = (low + high) // 2
        # Divide into two halves
        min1, max1 = find_min_max(arr, low, mid)
        min2, max2 = find_min_max(arr, mid + 1, high)
        # Conquer: Combine results
        return min(min1, min2), max(max1, max2)
if __name__ == "__main__":
    # Taking input from user
    arr = list(map(int, input("Enter array elements separated by space: ").split()))
    n = len(arr)
```

```
# Call function

minimum, maximum = find_min_max(arr, 0, n - 1)

# Display results

print("\nArray:", arr)

print("Minimum element:", minimum)

print("Maximum element:", maximum)
```

Program 6:

Write a test program to implement Divide and Conquer Strategy. Eg: Quick sort algorithm for sorting list of integers in ascending order.

```
def quick_sort(arr):
    # Base case: if array has 0 or 1 element, it is already sorted
    if len(arr) <= 1:
        return arr
    # Step 1: Choose a pivot (here we take the last element)
    pivot = arr[-1]
    # Step 2: Partition the array into two halves
    left = [x for x in arr[:-1] if x <= pivot]   # elements <= pivot
    right = [x for x in arr[:-1] if x > pivot]   # elements > pivot
    # Step 3: Recursively sort left and right, then combine
    return quick_sort(left) + [pivot] + quick_sort(right)
if __name__ == "__main__":
    # Take input from user
    arr = list(map(int, input("Enter integers separated by space: ").split()))
    print("\nOriginal Array:", arr)
    sorted_arr = quick_sort(arr)
    print("Sorted Array (Ascending Order):", sorted_arr)
```

Program 7:

Write a program to implement Merge sort algorithm for sorting a list of integers in ascending order.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    # Step 1: Divide - Find the middle index
    mid = len(arr) // 2
    # Step 2: Recursively sort left and right halves
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    # Step 3: Merge sorted halves
    return merge(left_half, right_half)
def merge(left, right):
    merged = []
    i = j = 0
    # Compare elements from left and right, append smaller one
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
if __name__ == "__main__":
```

```
print("Program 7: Merge Sort using Divide and Conquer\n")

# Take input from user

arr = list(map(int, input("Enter integers separated by space: ").split()))

print("\nOriginal Array:", arr)

sorted_arr = merge_sort(arr)

print("Sorted Array (Ascending Order):", sorted_arr)
```

Program 8:

Write C program that accepts the vertices and edges for a graph and stores it as an adjacency matrix.

```
def create_adjacency_matrix(vertices, edges):
    # Step 1: Initialize adjacency matrix with 0s
    adj_matrix = [[0] * vertices for _ in range(vertices)]
    # Step 2: Fill adjacency matrix with given edges
    for (u, v) in edges:
        adj_matrix[u][v] = 1
        adj_matrix[v][u] = 1   # For undirected graph
    return adj_matrix
if __name__ == "__main__":
    print("Program 8: Graph Representation using Adjacency Matrix\n")
    # Input number of vertices and edges
    n = int(input("Enter number of vertices: "))
    e = int(input("Enter number of edges: "))
    # Input edges
    edges = []
    print("\nEnter the edges (format: u v for edge between u and v):")
    for _ in range(e):
        u, v = map(int, input().split())
        edges.append((u, v))
    # Create adjacency matrix
    adj = create_adjacency_matrix(n, edges)
    print("\nAdjacency Matrix:")
    for row in adj:
        print(" ".join(map(str, row)))
```

Program 9:

Implement function to print In-Degree, Out-Degree and to display that adjacency matrix

```
def create_adjacency_matrix(vertices, edges, directed=False):
    # Step 1: Initialize adjacency matrix with 0s
    adj_matrix = [[0] * vertices for _ in range(vertices)]
    # Step 2: Fill adjacency matrix with given edges
    for (u, v) in edges:
        adj_matrix[u][v] = 1
        if not directed:   # For undirected graph, mark both
            adj_matrix[v][u] = 1
        return adj_matrix
def calculate_degrees(adj_matrix, directed=False):
    vertices = len(adj_matrix)
    in_degree = [0] * vertices
    out_degree = [0] * vertices
    for i in range(vertices):
        for j in range(vertices):
            if adj_matrix[i][j] == 1:
                out_degree[i] += 1
                in_degree[j] += 1
    return in_degree, out_degree
if __name__ == "__main__":
    print("Graph Representation with In-Degree and Out-Degree\n")
    # Input graph details
    n = int(input("Enter number of vertices: "))
    e = int(input("Enter number of edges: "))
    directed = input("Is the graph directed? (yes/no): ").lower() == "yes"
```

```
# Input edges
edges = []
print("\nEnter the edges (format: u v for edge u->v):")
for _ in range(e):
    u, v = map(int, input().split())
    edges.append((u, v))
# Create adjacency matrix
adj = create_adjacency_matrix(n, edges, directed)
# Display adjacency matrix
print("\nAdjacency Matrix:")
for row in adj:
    print(" ".join(map(str, row)))
# Calculate and display degrees
in_degree, out_degree = calculate_degrees(adj, directed)
print("\nVertex\tIn-Degree\tOut-Degree")
for i in range(n):
    print(f"{i}\t{in_degree[i]}\t\t{out_degree[i]}")
```

Program 10:

Write a program to perform Knapsack Problem using GreedySolution