CAP538: ALGORITHM DESIGN AND ANALYSIS

CONTINUOUS ASSESSMENTS (C.A)-3

ST_NAME	**: -**	**EKHLAKH AHMAD**
REG NO.	**: -**	**12209166**
ROLL NO.	**: -**	**03**
SECTION	**: -**	**REC72**
GROUP	**: -**	**01**

Q1. Explain the concept of Dynamic Programming and Greedy Method. Give one Real world use case when Dynamic Programming is more efficient than Greedy method

Dynamic Programming (DP) and the Greedy Method are two fundamental algorithmic strategies used to solve optimization problems.

Dynamic Programming is an optimization technique that solves complex problems by breaking them down into simpler subproblems and solving each of those subproblems just once, storing their solutions.

Characteristics:
Overlapping Subproblems: The problem can be broken down into subproblems which are reused several times.

Optimal Substructure: The optimal solution of the problem can be constructed efficiently from the optimal solutions of its subproblems.
Approach:

Define the Structure: Formulate the problem in terms of subproblems.

Recursive Solution: Write down a recursive solution.

Memoization or Tabulation: Use memoization (top-down approach) or tabulation (bottom-up approach) to solve the problem efficiently.

Example Problems:
Fibonacci Sequence
Knapsack Problem
Longest Common Subsequence
Matrix Chain Multiplication

Greedy Method:
The Greedy Method is an algorithmic strategy that makes a series of choices, each of which looks the best at the moment (locally optimal choice), hoping that these local choices will lead to a globally optimal solution.

Characteristics:
Locally Optimal Choice: Makes the best choice at each step without considering the global situation.

Greedy Choice Property: A global optimum can be arrived at by selecting the local optimums.
Approach:

- **Selection**: At each step, select the best possible option available.
- **Feasibility:** Check if the selected choice leads to a feasible solution.
- **Solution Construction:** Construct the solution incrementally.

Example Problems:

- Coin Change Problem (for specific denominations)
- Fractional Knapsack Problem
- Huffman Coding
- Prim's and Kruskal's algorithms for Minimum Spanning Tree

Real-world Use Case:
Dynamic Programming vs. Greedy Method
The 0/1 Knapsack Problem
Problem Description:
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. You cannot break an item, which means you either take the whole item or don't take it at all (0/1 property).

Greedy Approach:
A typical greedy approach might involve picking items based on the highest value-to-weight ratio. However, this approach does not always yield the optimal solution for the 0/1 Knapsack Problem.
Example: Consider items with weights [10, 20, 30] and values [60, 100, 120], and a knapsack capacity of 50. The greedy approach might pick the item with the highest value-to-weight ratio first, which could lead to suboptimal solutions.
Dynamic Programming Approach:

DP considers all possible combinations of items and builds a solution incrementally. It constructs a table where dp[i][w] represents the maximum value that can be attained with the first i items and a weight limit w.
This ensures that all subproblems are considered and combined to form the optimal solution.

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

int knapsack_01(int weights[], int values[], int n, int
capacity)
{
    int dp[n + 1][capacity + 1];

    for (int i = 0; i <= n; ++i)
    {
        for (int w = 0; w <= capacity; ++w)
        {
            if (i == 0 || w == 0)
            {
                dp[i][w] = 0;
            }
            else if (weights[i - 1] <= w)
            {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w -
weights[i - 1]], dp[i - 1][w]);
            }
            else
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main()
{
    int weights[] = {10, 20, 30};
    int values[] = {60, 100, 120};
```

```
    int capacity = 50;
    int n = sizeof(weights) / sizeof(weights[0]);

    cout << "Maximum value in Knapsack = " <<
knapsack_01(weights, values, n, capacity) << endl;
    return 0;
}
```

.

Q.2. Explain minimum spanning tree Using Kruskal Algorithm

A Minimum Spanning Tree (MST) of a connected, undirected graph is a subset of the edges that connects all the vertices together without any cycles and with the minimum possible total edge weight. Kruskal's Algorithm is a popular method for finding the MST of a graph.

Kruskal's Algorithm
Kruskal's Algorithm is a greedy algorithm that finds the MST by following these steps:

- Sort all the edges in non-decreasing order of their weight.
- Initialize the MST as an empty set.
- Iterate through the sorted edges and add each edge to the MST if it doesn't form a cycle (i.e., if it connects two previously unconnected components).

Steps of Kruskal's Algorithm

- Sort all edges based on their weights.
- Initialize a parent array to keep track of the components of each vertex.
- Process each edge in the sorted order:
- If the edge connects two different components, add it to the MST and merge the components.
- If it connects two vertices in the same component, discard the edge (as it would form a cycle).

Union-Find Data Structure
The Union-Find data structure supports two primary operations efficiently:

- **Find:** Determine which component a particular element is in.
- **Union:** Merge two components into one.

```cpp
#include <iostream>
#include <algorithm>

using namespace std;
class Edge
{
public:
    int src, dest, weight;
};

class Graph
{
public:
    int V, E;
    Edge *edges;

    Graph(int V, int E)
    {
        this->V = V;
        this->E = E;
        edges = new Edge[E];
    }

    ~Graph()
    {
        delete[] edges;
    }
};

class DisjointSets
{
public:
    int *parent;
    int *rank;
    int n;

    DisjointSets(int n)
    {
        this->n = n;
        parent = new int[n + 1];
        rank = new int[n + 1];
```

```
        for (int i = 0; i <= n; ++i)
        {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    ~DisjointSets()
    {
        delete[] parent;
        delete[] rank;
    }

    int find(int u)
    {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    void unionSets(int u, int v)
    {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV)
        {
            if (rank[rootU] < rank[rootV])
            {
                parent[rootU] = rootV;
            }
            else if (rank[rootU] > rank[rootV])
            {
                parent[rootV] = rootU;
            }
            else
            {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
```

```
        }
    }
};

int compareEdges(const void *a, const void *b)
{
    Edge *a1 = (Edge *)a;
    Edge *b1 = (Edge *)b;
    return a1->weight > b1->weight;
}

void kruskalMST(Graph &graph)
{
    int V = graph.V;
    Edge *result = new Edge[V];
    int e = 0;
    int i = 0;

    qsort(graph.edges, graph.E, sizeof(graph.edges[0]),
compareEdges);

    DisjointSets ds(V);

    while (e < V - 1 && i < graph.E)
    {
        Edge nextEdge = graph.edges[i++];

        int x = ds.find(nextEdge.src);
        int y = ds.find(nextEdge.dest);

        if (x != y)
        {
            result[e++] = nextEdge;
            ds.unionSets(x, y);
        }
    }

    cout << "Following are the edges in the constructed
MST\n";
    int minimumCost = 0;
    for (i = 0; i < e; ++i)
```

```
    {
        cout << result[i].src << " -- " << result[i].dest <<
" == " << result[i].weight << endl;
        minimumCost += result[i].weight;
    }
    cout << "Minimum Cost Spanning Tree: " << minimumCost <<
endl;

    delete[] result;
}

int main()
{
    int V = 4;
    int E = 5;
    Graph graph(V, E);

    graph.edges[0].src = 0;
    graph.edges[0].dest = 1;
    graph.edges[0].weight = 10;

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 6;

    graph.edges[2].src = 0;
    graph.edges[2].dest = 3;
    graph.edges[2].weight = 5;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 15;

    graph.edges[4].src = 2;
    graph.edges[4].dest = 3;
    graph.edges[4].weight = 4;

    kruskalMST(graph);

    return 0;
}
```

Q.3. Define recursion with one example of sum of first n natural no and draw state space tree of recursion.

Recursion is a programming technique in which a function calls itself directly or indirectly to solve a problem. It breaks down a problem into smaller, more manageable sub-problems of the same type, and combines their results to solve the original problem. Recursion typically involves a base case that stops the recursion, and a recursive case that continues it.

Example: Sum of First n Natural Numbers

```
#include <iostream>

using namespace std;

int sumOfNaturalNumbers(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else
    {
        return n + sumOfNaturalNumbers(n - 1);
    }
}

int main()
{
    int n = 5;
    cout << "Sum of first " << n << " natural numbers is "
<< sumOfNaturalNumbers(n) << endl;
    return 0;
}
```

State Space Tree of Recursion:
The state space tree for the recursion with n = 5 looks like this:

```
sumOfNaturalNumbers(5) |
    +--sumOfNaturalNumbers(4) |
    +--sumOfNaturalNumbers(3) |
    +--sumOfNaturalNumbers(2) |
    +--sumOfNaturalNumbers(1) |
    +--sumOfNaturalNumbers(0)
```

Explanation of State Space Tree
sumOfNaturalNumbers(5) calls sumOfNaturalNumbers(4).
sumOfNaturalNumbers(4) calls sumOfNaturalNumbers(3).
sumOfNaturalNumbers(3) calls sumOfNaturalNumbers(2).
sumOfNaturalNumbers(2) calls sumOfNaturalNumbers(1).
sumOfNaturalNumbers(1) calls sumOfNaturalNumbers(0).