Assignment No:01

Problem Statement:

Write a program to implement Fractional knapsack using Greedy algorithm and 0/1 knapsack using Dynamic programming. Show that Greedy strategy does not necessarily yield an optimal solution over a dynamic programming approach.

// Fractional Knapsack using Greedy Algorithm:

```
#include <bits/stdc++.h>
using namespace std;
// Structure for an item
struct Item {
    int profit, weight;
    Item(int profit, int weight) {
        this->profit = profit;
        this->weight = weight;
    }
};
// Comparison function to sort items by profit/weight ratio
static bool cmp(struct Item a, struct Item b) {
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}
// Greedy function to solve fractional knapsack
double fractionalKnapsack(int W, struct Item arr[], int N) {
    // Sort items by ratio
    sort(arr, arr + N, cmp);
    double finalValue = 0.0;
    for (int i = 0; i < N; i++) {
        if (arr[i].weight <= W) {
```

```
            W -= arr[i].weight;
            finalValue += arr[i].profit;
        }
        else {
            finalValue += arr[i].profit * ((double)W / (double)arr[i].weight);
            break;
        }
    }
    return finalValue;
}

int main() {
    int W = 50; // Knapsack capacity
    Item arr[] = { {60, 10}, {100, 20}, {120, 30} };
    int N = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum profit = " << fractionalKnapsack(W, arr, N);
    return 0;
}
```

// 0/1 knapsack using Dynamic programming:

```
#include <bits/stdc++.h>
using namespace std;
int knapsack_dp(int n, int M, int w[], int p[]) {
    int i, j;
    int knapsack[n+1][M+1];
    for (j = 0; j <= M; j++)
```

```
        knapsack[0][j] = 0;
    for (i = 0; i <= n; i++)
        knapsack[i][0] = 0;

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= M; j++) {
            if (w[i-1] <= j) {
                knapsack[i][j] = max(knapsack[i-1][j],
                                     p[i-1] + knapsack[i-1][j - w[i-1]]);
            }
            else {
                knapsack[i][j] = knapsack[i-1][j];
            }
        }
    }
    return knapsack[n][M];
}

int main() {
    int n = 4;
    int M = 5;
    int w[] = {2, 1, 3, 2};
    int p[] = {12, 10, 20, 15};

    cout << "Maximum profit = " << knapsack_dp(n, M, w, p);
    return 0;
}
```

Assignment No:02

Problem Statement:

Write a program to implement Bellman-Ford Algorithm using Dynamic programming and verify the time Complexity.

```
#include <bits/stdc++.h>
using namespace std;
struct Edge {
    int u, v, w;
};
void bellmanFord(int V, int E, vector<Edge>& edges, int src) {
    vector<int> dist(V, INT_MAX), pred(V, -1);
    dist[src] = 0;
// Step 2: Relax edges repeatedly
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].u;
            int v = edges[j].v;
            int w = edges[j].w;
            if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pred[v] = u;
            }
        }
    }

    // Step 3: Check for negative-weight cycles
    for (int j = 0; j < E; j++) {
        int u = edges[j].u;
```

```
        int v = edges[j].v;
        int w = edges[j].w;
        if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
            cout << "Graph contains negative weight cycle\n";
            return;
        }
    }
cout << "Vertex distances from source " << src << ":\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX)
            cout << i << " : INF\n";
        else
            cout << i << " : " << dist[i] << " (Predecessor: " << pred[i] << ")\n";
    }
}

int main() {
    int V = 5, E = 8;
    vector<Edge> edges = {
        {0, 1, -1}, {0, 2, 4}, {1, 2, 3},
        {1, 3, 2},  {1, 4, 2}, {3, 2, 5},
        {3, 1, 1},  {4, 3, -3}
    };

    bellmanFord(V, E, edges, 0);
    return 0;
}
```

Assignment No:03

Problem Statement:

Write a recursive program to find the solution of placing n queens on a chessboard so that no queen takes each other.

```
#include <iostream>

#include <cmath>

using namespace std;


int x[20];   // x[k] = column position of queen in row k

int solutionCount = 0;

// Function to check if a queen can be placed at row k, column i

bool Place(int k, int i) {

   for (int j = 1; j < k; j++) {

      if (x[j] == i || abs(x[j] - i) == abs(j - k)) {

         return false; // same column or diagonal

      }

   }

   return true;

}

// Recursive function to solve N-Queens

void NQueens(int k, int n) {

   for (int i = 1; i <= n; i++) {

      if (Place(k, i)) {

         x[k] = i;

         if (k == n) {

            solutionCount++;

            cout << "Solution " << solutionCount << ": ";

            for (int j = 1; j <= n; j++) {
```

```
                cout << x[j] << " ";
            }
            cout << endl;
        } else {
            NQueens(k + 1, n);
        }
    }
  }
}

int main() {
  int n;
  cout << "Enter number of queens: ";
  cin >> n;

  NQueens(1, n);

  cout << "Total number of solutions = " << solutionCount << endl;
  return 0;
}
```

Assignment No:04

Problem Statement:

Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound.

```
#include <bits/stdc++.h>

using namespace std;

#define INF INT_MAX

int n;                  // Number of cities

int cost[20][20];       // Distance matrix

int finalRes = INF;     // Minimum cost

vector<int> finalPath;  // Stores the final path


// Copy current path to final path

void copyToFinal(vector<int>& currPath) {

    finalPath = currPath;

    finalPath.push_back(currPath[0]);

}

// Row reduction

int rowReduction(int tempCost[20][20]) {

    int rowRed = 0;

    for (int i = 0; i < n; i++) {

        int rmin = INF;

        for (int j = 0; j < n; j++)

            if (tempCost[i][j] < rmin)

                rmin = tempCost[i][j];

        if (rmin != INF && rmin != 0) {

            rowRed += rmin;

            for (int j = 0; j < n; j++)
```

```
            if (tempCost[i][j] != INF)
                tempCost[i][j] -= rmin;
        }
    }
    return rowRed;
}
// Column reduction
int colReduction(int tempCost[20][20]) {
    int colRed = 0;
    for (int j = 0; j < n; j++) {
        int cmin = INF;
        for (int i = 0; i < n; i++)
            if (tempCost[i][j] < cmin)
                cmin = tempCost[i][j];
        if (cmin != INF && cmin != 0) {
            colRed += cmin;
            for (int i = 0; i < n; i++)
                if (tempCost[i][j] != INF)
                    tempCost[i][j] -= cmin;
        }
    }
    return colRed;
}
// Calculate bound for choosing edge src -> dest
int checkBounds(int src, int dest, int tempCost[20][20], int currCost) {
    int reduced[20][20];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            reduced[i][j] = tempCost[i][j];
```

```
    for (int j = 0; j < n; j++)
        reduced[src][j] = INF;
    for (int i = 0; i < n; i++)
        reduced[i][dest] = INF;
    reduced[dest][src] = INF;

    int row = rowReduction(reduced);
    int col = colReduction(reduced);

    return currCost + row + col + tempCost[src][dest];
}
// Recursive Branch and Bound for TSP
void TSP(vector<int>& currPath, int currCost, vector<bool>& visited) {
    if (currPath.size() == n) {
        currCost += cost[currPath.back()][currPath[0]];
        if (currCost < finalRes) {
            finalRes = currCost;
            copyToFinal(currPath);
        }
        return;
    }

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            int tempCost[20][20];
            memcpy(tempCost, cost, sizeof(cost));
            int bound = checkBounds(currPath.back(), i, tempCost, currCost);
            if (bound < finalRes) {
```

```
                visited[i] = true;
                currPath.push_back(i);
                TSP(currPath, bound, visited);
                currPath.pop_back();
                visited[i] = false;
            }
        }
    }
}

int main() {
    cout << "INPUT:\nNumber of cities: ";
    cin >> n;
    cout << "Enter distance matrix (" << n << "x" << n << "):\n";

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            cin >> cost[i][j];
            if (cost[i][j] == 0) cost[i][j] = INF;
        }

    vector<int> currPath;
    vector<bool> visited(n, false);
    currPath.push_back(0);
    visited[0] = true;

    int temp[20][20];
    memcpy(temp, cost, sizeof(cost));
    int currCost = rowReduction(temp) + colReduction(temp);
```

```
    TSP(currPath, currCost, visited);

    cout << "\nOUTPUT:\n";
    cout << "Shortest path for traveling salesman problem covering all cities:\n";
    for (int i = 0; i < finalPath.size(); i++)
        cout << finalPath[i] + 1 << " ";
    cout << "\nMinimum cost: " << finalRes << endl;

    return 0;
}
```