**Assignment of DAA..**

**Yajat Rajput**

**12213133**

Q1. MERGE SORT

#include <iostream>

#include <vector>

void merge(std::vector<int>& arr, int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    std::vector<int> left\_half(n1);

    std::vector<int> right\_half(n2);

    for (int i = 0; i < n1; i++)

        left\_half[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        right\_half[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        if (left\_half[i] <= right\_half[j]) {

            arr[k] = left\_half[i];

            i++;

        } else {

            arr[k] = right\_half[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = left\_half[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = right\_half[j];

        j++;

        k++;

    }

}

void mergeSort(std::vector<int>& arr, int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);

    }

}

int main() {

    std::vector<int> arr = {12, -5, 7, 2, 8, -3, 1, 0, -9, 4};

    mergeSort(arr, 0, arr.size() - 1);

    std::cout << "Merge Sorted Array: ";

    for (int num : arr) {

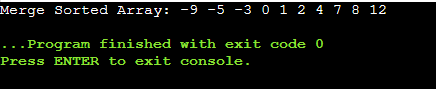
        std::cout << num << " ";

    }

    return 0;

}

**OUTPUT –**



Q2.MAX HEAP

#include <iostream>

#include <vector>

void heapify(std::vector<int>& arr, int n, int i) {

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

    if (left < n && arr[left] > arr[largest])

        largest = left;

    if (right < n && arr[right] > arr[largest])

        largest = right;

    if (largest != i) {

        std::swap(arr[i], arr[largest]);

        heapify(arr, n, largest);

    }

}

void heapSort(std::vector<int>& arr) {

    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {

        std::swap(arr[0], arr[i]);

        heapify(arr, i, 0);

    }

}

int main() {

    std::vector<int> arr = {12, -5, 7, 2, 8, -3, 1, 0, -9, 4};

    heapSort(arr);

    std::cout << "Heap Sorted Array: ";

    for (int num : arr) {

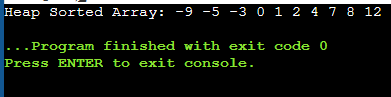
        std::cout << num << " ";

    }

    return 0;

}

**OUTPUT—**



Q3. Krushkal’s Algo.

#include <iostream>

#include <vector>

#include <algorithm>

class Edge {

public:

    int src, dest, weight;

    Edge(int s, int d, int w) : src(s), dest(d), weight(w) {}

};

class Graph {

public:

    int V;

    std::vector<Edge> edges;

    Graph(int vertices) : V(vertices) {}

    void addEdge(int src, int dest, int weight) {

        edges.push\_back(Edge(src, dest, weight));

    }

    void kruskalMST();

};

class DisjointSet {

public:

    int \*parent, \*rank;

    int n;

    DisjointSet(int n) {

        this->n = n;

        parent = new int[n];

        rank = new int[n];

        for (int i = 0; i < n; i++) {

            parent[i] = i;

            rank[i] = 0;

        }

    }

    int find(int u) {

        if (u != parent[u])

            parent[u] = find(parent[u]);

        return parent[u];

    }

    void unionSets(int x, int y) {

        int rootX = find(x);

        int rootY = find(y);

        if (rank[rootX] < rank[rootY])

            parent[rootX] = rootY;

        else if (rank[rootX] > rank[rootY])

            parent[rootY] = rootX;

        else {

            parent[rootY] = rootX;

            rank[rootX]++;

        }

    }

};

bool compareEdges(Edge a, Edge b) {

    return a.weight < b.weight;

}

void Graph::kruskalMST() {

    std::sort(edges.begin(), edges.end(), compareEdges);

    DisjointSet ds(V);

    std::cout << "Edges in the Minimum Spanning Tree using Kruskal's Algorithm:\n";

    for (Edge edge : edges) {

        int rootSrc = ds.find(edge.src);

        int rootDest = ds.find(edge.dest);

        if (rootSrc != rootDest) {

            std::cout << edge.src << " - " << edge.dest << "   Weight: " << edge.weight << "\n";

            ds.unionSets(rootSrc, rootDest);

        }

    }

}

int main() {

    Graph graph(9);

    graph.addEdge(0, 1, 4);

    graph.addEdge(0, 7, 8);

    graph.addEdge(1, 2, 8);

    graph.addEdge(1, 7, 11);

    graph.addEdge(2, 3, 7);

    graph.addEdge(2, 8, 2);

    graph.addEdge(2, 5, 4);

    graph.addEdge(3, 4, 9);

    graph.addEdge(3, 5, 14);

    graph.addEdge(4, 5, 10);

    graph.addEdge(5, 6, 2);

    graph.addEdge(6, 7, 1);

    graph.addEdge(6, 8, 6);

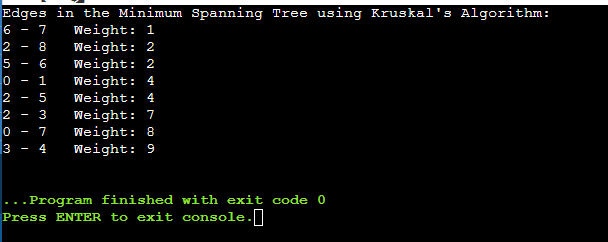
    graph.addEdge(7, 8, 7);

    graph.kruskalMST();

    return 0;

}

**OUTPUT—**

****

PRIM’s ALGO.

#include <iostream>

#include <vector>

#include <climits>

class Graph {

public:

    int V;

    std::vector<std::vector<int>> adjMatrix;

    Graph(int vertices) : V(vertices) {

        adjMatrix.resize(V, std::vector<int>(V, 0));

    }

    void addEdge(int src, int dest, int weight) {

        adjMatrix[src][dest] = weight;

        adjMatrix[dest][src] = weight;

    }

    void primMST();

};

int minKey(std::vector<int>& key, std::vector<bool>& mstSet) {

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < key.size(); v++) {

        if (!mstSet[v] && key[v] < min) {

            min = key[v];

            min\_index = v;

        }

    }

    return min\_index;

}

void Graph::primMST() {

    std::vector<int> parent(V, -1);

    std::vector<int> key(V, INT\_MAX);

    std::vector<bool> mstSet(V, false);

    key[0] = 0;

    for (int count = 0; count < V - 1; count++) {

        int u = minKey(key, mstSet);

        mstSet[u] = true;

        for (int v = 0; v < V; v++) {

            if (adjMatrix[u][v] && !mstSet[v] && adjMatrix[u][v] < key[v]) {

                parent[v] = u;

                key[v] = adjMatrix[u][v];

            }

        }

    }

    std::cout << "Edges in the Minimum Spanning Tree using Prim's Algorithm:\n";

    for (int i = 1; i < V; i++)

        std::cout << parent[i] << " - " << i << "   Weight: " << adjMatrix[i][parent[i]] << "\n";

}

int main() {

    Graph graph(9);

    graph.addEdge(0, 1, 4);

    graph.addEdge(0, 7, 8);

    graph.addEdge(1, 2, 8);

    graph.addEdge(1, 7, 11);

    graph.addEdge(2, 3, 7);

    graph.addEdge(2, 8, 2);

    graph.addEdge(2, 5, 4);

    graph.addEdge(3, 4, 9);

    graph.addEdge(3, 5, 14);

    graph.addEdge(4, 5, 10);

    graph.addEdge(5, 6, 2);

    graph.addEdge(6, 7, 1);

    graph.addEdge(6, 8, 6);

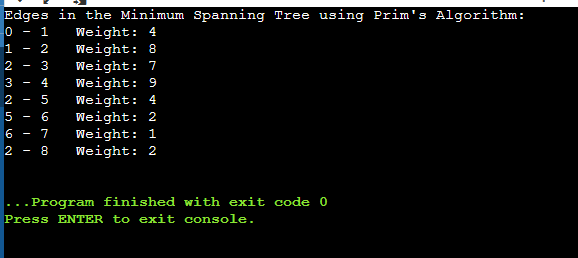
    graph.addEdge(7, 8, 7);

    graph.primMST();

    return 0;

}

**OUTPUT—**

****

**Q4.** Dijkstra Algo.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

class Graph {

public:

    int V;

    std::vector<std::vector<std::pair<int, int>>> adjList;

    Graph(int vertices) : V(vertices) {

        adjList.resize(V);

    }

    void addEdge(int src, int dest, int weight) {

        adjList[src].emplace\_back(dest, weight);

        adjList[dest].emplace\_back(src, weight); // Assuming the graph is undirected

    }

    void dijkstra(int start);

};

void Graph::dijkstra(int start) {

    std::vector<int> dist(V, std::numeric\_limits<int>::max());

    dist[start] = 0;

    std::priority\_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<std::pair<int, int>>> pq;

    pq.push({0, start});

    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();

        for (const auto& neighbor : adjList[u]) {

            int v = neighbor.first;

            int weight = neighbor.second;

            if (dist[v] > dist[u] + weight) {

                dist[v] = dist[u] + weight;

                pq.push({dist[v], v});

            }

        }

    }

    std::cout << "Shortest distances from node " << start << ":\n";

    for (int i = 0; i < V; ++i) {

        std::cout << "Node " << i << ": ";

        if (dist[i] == std::numeric\_limits<int>::max())

            std::cout << "INF\n";

        else

            std::cout << dist[i] << "\n";

    }

}

int main() {

    Graph graph(6);

    graph.addEdge(0, 1, 2);

    graph.addEdge(0, 2, 4);

    graph.addEdge(1, 2, 1);

    graph.addEdge(1, 3, 7);

    graph.addEdge(2, 4, 3);

    graph.addEdge(3, 5, 1);

    graph.addEdge(4, 3, 2);

    graph.addEdge(4, 5, 5);

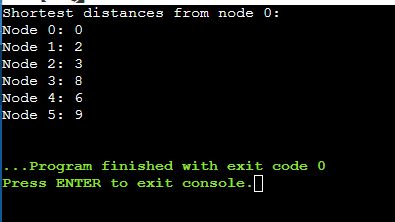
    int startNode = 0;

    graph.dijkstra(startNode);

    return 0;

}

**OUTPUT—**

****

**Q5.** Bellman Ford Algo..

#include <iostream>

#include <vector>

#include <limits>

class Edge {

public:

    int src, dest, weight;

    Edge(int s, int d, int w) : src(s), dest(d), weight(w) {}

};

class Graph {

public:

    int V, E;

    std::vector<Edge> edges;

    Graph(int vertices, int edgesCount) : V(vertices), E(edgesCount) {}

    void addEdge(int src, int dest, int weight) {

        edges.push\_back(Edge(src, dest, weight));

    }

    void bellmanFord(int start);

};

void Graph::bellmanFord(int start) {

    std::vector<int> dist(V, std::numeric\_limits<int>::max());

    dist[start] = 0;

    for (int i = 1; i <= V - 1; ++i) {

        for (const Edge& edge : edges) {

            int u = edge.src;

            int v = edge.dest;

            int weight = edge.weight;

            if (dist[u] != std::numeric\_limits<int>::max() && dist[u] + weight < dist[v]) {

                dist[v] = dist[u] + weight;

            }

        }

    }

    // Check for negative weight cycles

    for (const Edge& edge : edges) {

        int u = edge.src;

        int v = edge.dest;

        int weight = edge.weight;

        if (dist[u] != std::numeric\_limits<int>::max() && dist[u] + weight < dist[v]) {

            std::cerr << "Graph contains negative weight cycle. Bellman-Ford algorithm cannot be applied.\n";

            return;

        }

    }

    std::cout << "Shortest distances from node " << start << ":\n";

    for (int i = 0; i < V; ++i) {

        std::cout << "Node " << i << ": ";

        if (dist[i] == std::numeric\_limits<int>::max())

            std::cout << "INF\n";

        else

            std::cout << dist[i] << "\n";

    }

}

int main() {

    Graph graph(5, 8); // 5 vertices, 8 edges

    graph.addEdge(0, 1, -1);

    graph.addEdge(0, 2, 4);

    graph.addEdge(1, 2, 3);

    graph.addEdge(1, 3, 2);

    graph.addEdge(1, 4, 2);

    graph.addEdge(3, 2, 5);

    graph.addEdge(3, 1, 1);

    graph.addEdge(4, 3, -3);

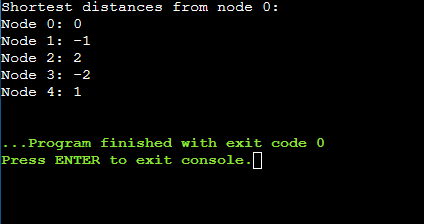
    int startNode = 0;

    graph.bellmanFord(startNode);

    return 0;

}

**OUTPUT—**

****

**Q6.** Floyd-warshall Algo

#include <iostream>

#include <vector>

#include <limits>

class Graph {

public:

    int V;

    std::vector<std::vector<int>> adjMatrix;

    Graph(int vertices) : V(vertices) {

        adjMatrix.resize(V, std::vector<int>(V, std::numeric\_limits<int>::max()));

        // Initializing the diagonal with zeros

        for (int i = 0; i < V; ++i) {

            adjMatrix[i][i] = 0;

        }

    }

    void addEdge(int src, int dest, int weight) {

        adjMatrix[src][dest] = weight;

    }

    void floydWarshall();

};

void Graph::floydWarshall() {

    for (int k = 0; k < V; ++k) {

        for (int i = 0; i < V; ++i) {

            for (int j = 0; j < V; ++j) {

                if (adjMatrix[i][k] != std::numeric\_limits<int>::max() &&

                    adjMatrix[k][j] != std::numeric\_limits<int>::max() &&

                    adjMatrix[i][k] + adjMatrix[k][j] < adjMatrix[i][j]) {

                    adjMatrix[i][j] = adjMatrix[i][k] + adjMatrix[k][j];

                }

            }

        }

    }

    std::cout << "Shortest paths between all pairs of vertices:\n";

    for (int i = 0; i < V; ++i) {

        for (int j = 0; j < V; ++j) {

            if (adjMatrix[i][j] == std::numeric\_limits<int>::max()) {

                std::cout << "INF ";

            } else {

                std::cout << adjMatrix[i][j] << " ";

            }

        }

        std::cout << std::endl;

    }

}

int main() {

    Graph graph(4);

    graph.addEdge(0, 1, 5);

    graph.addEdge(0, 2, 9);

    graph.addEdge(0, 3, 4);

    graph.addEdge(1, 0, 2);

    graph.addEdge(1, 2, 3);

    graph.addEdge(2, 3, 2);

    graph.addEdge(3, 1, 7);

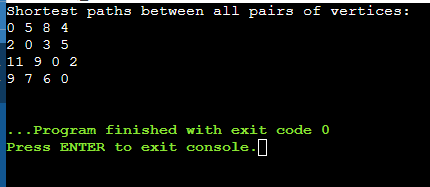
    graph.addEdge(3, 2, 6);

    graph.floydWarshall();

    return 0;

}

**OUTPUT—**

****

**Q7.** Binary Tree

#include <iostream>

class TreeNode {

public:

    int data;

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}

};

class BinarySearchTree {

public:

    TreeNode\* root;

    BinarySearchTree() : root(nullptr) {}

    void insert(int value) {

        root = insertRecursive(root, value);

    }

    void preOrderTraversal() {

        std::cout << "Pre-order Traversal: ";

        preOrderRecursive(root);

        std::cout << std::endl;

    }

    void inOrderTraversal() {

        std::cout << "In-order Traversal: ";

        inOrderRecursive(root);

        std::cout << std::endl;

    }

    void postOrderTraversal() {

        std::cout << "Post-order Traversal: ";

        postOrderRecursive(root);

        std::cout << std::endl;

    }

private:

    TreeNode\* insertRecursive(TreeNode\* node, int value) {

        if (node == nullptr) {

            return new TreeNode(value);

        }

        if (value < node->data) {

            node->left = insertRecursive(node->left, value);

        } else if (value > node->data) {

            node->right = insertRecursive(node->right, value);

        }

        return node;

    }

    void preOrderRecursive(TreeNode\* node) {

        if (node != nullptr) {

            std::cout << node->data << " ";

            preOrderRecursive(node->left);

            preOrderRecursive(node->right);

        }

    }

    void inOrderRecursive(TreeNode\* node) {

        if (node != nullptr) {

            inOrderRecursive(node->left);

            std::cout << node->data << " ";

            inOrderRecursive(node->right);

        }

    }

    void postOrderRecursive(TreeNode\* node) {

        if (node != nullptr) {

            postOrderRecursive(node->left);

            postOrderRecursive(node->right);

            std::cout << node->data << " ";

        }

    }

};

int main() {

    BinarySearchTree bst;

    // Input numbers to build the BST

    int numbers[] = {50, 30, 70, 20, 40, 60, 80};

    // Build the BST

    for (int number : numbers) {

        bst.insert(number);

    }

    // Perform traversals

    bst.preOrderTraversal();

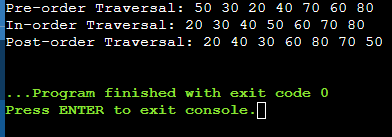
    bst.inOrderTraversal();

    bst.postOrderTraversal();

    return 0;

}

**OUTPUT—**

****

**Q10.** Hamiltonian cycle.

#include <iostream>

#include <vector>

#include <algorithm>

class HamiltonianCycle {

public:

    HamiltonianCycle(int vertices);

    void addEdge(int src, int dest);

    void findHamiltonianCycle();

private:

    int vertices;

    std::vector<std::vector<int>> adjacencyMatrix;

    std::vector<int> path;

    bool isSafe(int v, int pos);

    bool hamiltonianCycleUtil(int pos);

};

HamiltonianCycle::HamiltonianCycle(int vertices) : vertices(vertices) {

    adjacencyMatrix.resize(vertices, std::vector<int>(vertices, 0));

    path.resize(vertices, -1);

}

void HamiltonianCycle::addEdge(int src, int dest) {

    adjacencyMatrix[src][dest] = 1;

    adjacencyMatrix[dest][src] = 1; // Assuming the graph is undirected

}

void HamiltonianCycle::findHamiltonianCycle() {

    path[0] = 0;

    if (hamiltonianCycleUtil(1)) {

        std::cout << "Hamiltonian Cycle found: ";

        for (int vertex : path) {

            std::cout << vertex << " ";

        }

        std::cout << path[0] << std::endl;

    } else {

        std::cout << "No Hamiltonian Cycle exists in the given graph.\n";

    }

}

bool HamiltonianCycle::isSafe(int v, int pos) {

    if (!adjacencyMatrix[path[pos - 1]][v]) {

        return false;

    }

    for (int i = 0; i < pos; ++i) {

        if (path[i] == v) {

            return false;

        }

    }

    return true;

}

bool HamiltonianCycle::hamiltonianCycleUtil(int pos) {

    if (pos == vertices) {

        return adjacencyMatrix[path[pos - 1]][path[0]] == 1;

    }

    for (int v = 1; v < vertices; ++v) {

        if (isSafe(v, pos)) {

            path[pos] = v;

            if (hamiltonianCycleUtil(pos + 1)) {

                return true;

            }

            path[pos] = -1;

        }

    }

    return false;

}

int main() {

    HamiltonianCycle graph(5);

    graph.addEdge(0, 1);

    graph.addEdge(0, 3);

    graph.addEdge(1, 2);

    graph.addEdge(1, 3);

    graph.addEdge(1, 4);

    graph.addEdge(2, 4);

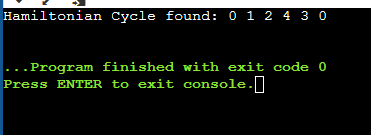
    graph.addEdge(3, 4);

    graph.findHamiltonianCycle();

    return 0;

}

**OUTPUT—**

****

**Q9.** Knapsack Algo.

#include <iostream>

#include <vector>

class Knapsack {

public:

    static int knapsack(int W, const std::vector<int>& weights, const std::vector<int>& values);

};

int Knapsack::knapsack(int W, const std::vector<int>& weights, const std::vector<int>& values) {

    int n = weights.size();

    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {

        for (int w = 1; w <= W; ++w) {

            if (weights[i - 1] <= w) {

                dp[i][w] = std::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][W];

}

int main() {

    std::vector<int> weights = {2, 3, 4, 5};

    std::vector<int> values = {3, 4, 5, 6};

    int maxWeight = 5;

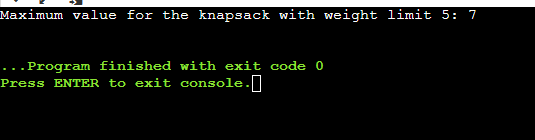
    int maxValue = Knapsack::knapsack(maxWeight, weights, values);

    std::cout << "Maximum value for the knapsack with weight limit " << maxWeight << ": " << maxValue << std::endl;

    return 0;

}

**OUTPUT—**

****