**Assignment 9**

**Name** : Komal Potdar

**Roll No**.: 92

**PRN No**.: 12320165

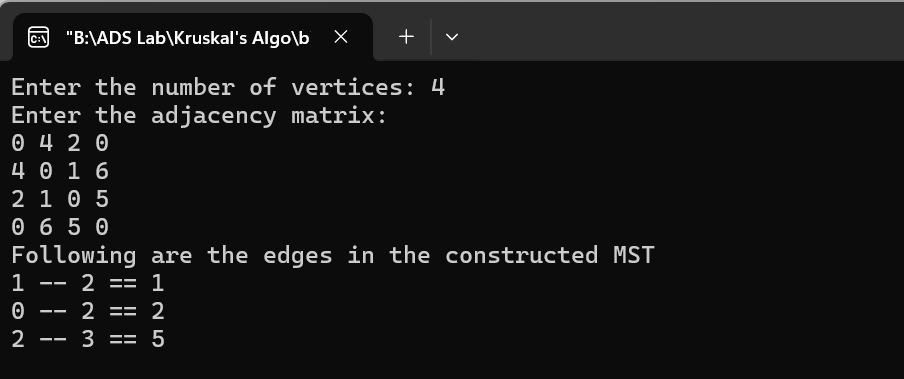
**Div**: CS B SY

**Batch**: 3

1. **WAP to generate Minimum Spanning Tree using Kruskal's Algorithm when graph is represented by  
   A. Adjacency Matrix..**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_VERTICES 100  int comparator(const void\* p1, const void\* p2)  {  const int (\*x)[3] = p1;  const int (\*y)[3] = p2;  return (\*x)[2] - (\*y)[2];  }  void makeSet(int parent[], int rank[], int n)  {  for (int i = 0; i < n; i++)  {  parent[i] = i;  rank[i] = 0;  }  }  int findParent(int parent[], int component)  {  if (parent[component] == component)  return component;  return parent[component] = findParent(parent, parent[component]);  }  void unionSet(int u, int v, int parent[], int rank[], int n)  {  // Finding the parents  u = findParent(parent, u);  v = findParent(parent, v);  if (rank[u] < rank[v])  {  parent[u] = v;  }  else if (rank[u] > rank[v])  {  parent[v] = u;  }  else  {  parent[v] = u;  rank[u]++;  }  }  void kruskalAlgo(int n, int edge[][3])  {    qsort(edge, n, sizeof(edge[0]), comparator);  int parent[n];  int rank[n];  makeSet(parent, rank, n);  int minCost = 0;  printf("Following are the edges in the constructed MST\n");  for (int i = 0; i < n; i++)  {  int v1 = findParent(parent, edge[i][0]);  int v2 = findParent(parent, edge[i][1]);  int wt = edge[i][2];    if (v1 != v2)  {  unionSet(v1, v2, parent, rank, n);  minCost += wt;  printf("%d -- %d == %d\n", edge[i][0],  edge[i][1], wt);  }  }  printf("Minimum Cost Spanning Tree: %d\n", minCost);  }  void createAdjacencyMatrix(int numVertices, int adjacencyMatrix[][MAX\_VERTICES])  {  printf("Enter the adjacency matrix:\n");  for (int i = 0; i < numVertices; i++)  {  for (int j = 0; j < numVertices; j++)  {  scanf("%d", &adjacencyMatrix[i][j]);  }  }  }  int main()  {  int numVertices, numEdges;  printf("Enter the number of vertices: ");  scanf("%d", &numVertices);  numEdges = numVertices \* (numVertices - 1) / 2; // Assuming a complete graph  int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];  createAdjacencyMatrix(numVertices, adjacencyMatrix);  int edge[numEdges][3];  int edgeCount = 0;  // Converting adjacency matrix to edge list  for (int i = 0; i < numVertices; i++)  {  for (int j = i + 1; j < numVertices; j++)  {  if (adjacencyMatrix[i][j] != 0)  {  edge[edgeCount][0] = i;  edge[edgeCount][1] = j;  edge[edgeCount][2] = adjacencyMatrix[i][j];  edgeCount++;  }  }  }  kruskalAlgo(numEdges, edge);  return 0;  } |

Output:



1. **WAP to generate Minimum Spanning Tree using Kruskal's Algorithm when graph is represented by  
   B. Adjacency List**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  typedef struct Node {  int vertex;  int weight;  struct Node\* next;  } Node;  Node\* createNode(int vertex, int weight) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  newNode->vertex = vertex;  newNode->weight = weight;  newNode->next = NULL;  return newNode;  }  void addEdge(Node\*\* head, int vertex, int weight) {  Node\* newNode = createNode(vertex, weight);  newNode->next = \*head;  \*head = newNode;  }  void printAdjacencyList(int numVertices, Node\* adjacencyList[]) {  for (int i = 0; i < numVertices; i++) {  Node\* current = adjacencyList[i];  printf("Adjacency list for vertex %d: ", i);  while (current != NULL) {  printf("(%d, %d) ", current->vertex, current->weight);  current = current->next;  }  printf("\n");  }  }  void freeAdjacencyList(int numVertices, Node\* adjacencyList[]) {  for (int i = 0; i < numVertices; i++) {  Node\* current = adjacencyList[i];  while (current != NULL) {  Node\* temp = current;  current = current->next;  free(temp);  }  }  }  void createEdgeList(int numVertices, Node\* adjacencyList[], int edge[][3], int \*numEdges) {  int edgeCount = 0;  for (int i = 0; i < numVertices; i++) {  Node\* current = adjacencyList[i];  while (current != NULL) {  edge[edgeCount][0] = i;  edge[edgeCount][1] = current->vertex;  edge[edgeCount][2] = current->weight;  edgeCount++;  current = current->next;  }  }  \*numEdges = edgeCount;  }  int comparator(const void\* p1, const void\* p2) {  const int (\*x)[3] = p1;  const int (\*y)[3] = p2;  return (\*x)[2] - (\*y)[2];  }  int findParent(int parent[], int component) {  if (parent[component] == component)  return component;  return parent[component] = findParent(parent, parent[component]);  }  void unionSet(int u, int v, int parent[], int rank[], int n) {  u = findParent(parent, u);  v = findParent(parent, v);  if (rank[u] < rank[v]) {  parent[u] = v;  } else if (rank[u] > rank[v]) {  parent[v] = u;  } else {  parent[v] = u;  rank[u]++;  }  }  void kruskalAlgo(int numEdges, int edge[][3], int numVertices) {  qsort(edge, numEdges, sizeof(edge[0]), comparator);  int parent[numVertices];  int rank[numVertices];  int minCost = 0;  for (int i = 0; i < numVertices; i++) {  parent[i] = i;  rank[i] = 0;  }  printf("Following are the edges in the constructed MST\n");  for (int i = 0; i < numEdges; i++) {  int u = findParent(parent, edge[i][0]);  int v = findParent(parent, edge[i][1]);  int weight = edge[i][2];  if (u != v) {  printf("%d -- %d == %d\n", edge[i][0], edge[i][1], weight);  minCost += weight;  unionSet(u, v, parent, rank, numVertices);  }  }  printf("Minimum Cost Spanning Tree: %d\n", minCost);  }  int main() {  int numVertices;  printf("Enter the number of vertices: ");  scanf("%d", &numVertices);  Node\* adjacencyList[numVertices];  for (int i = 0; i < numVertices; i++) {  adjacencyList[i] = NULL;  printf("Enter adjacency list for vertex %d (-1 to terminate): ", i);  int vertex, weight;  while (true) {  scanf("%d", &vertex);  if (vertex == -1)  break;  scanf("%d", &weight);  addEdge(&adjacencyList[i], vertex, weight);  }  }  printAdjacencyList(numVertices, adjacencyList);  int numEdges = numVertices \* numVertices; // Max possible edges for a complete graph  int edge[numEdges][3];  int actualNumEdges;  createEdgeList(numVertices, adjacencyList, edge, &actualNumEdges);  kruskalAlgo(actualNumEdges, edge, numVertices);  freeAdjacencyList(numVertices, adjacencyList);  return 0;  } |

Output:

