1. **BFS & DFS**

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
  int items[SIZE];
  int front;
  int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int);

struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};

void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue();

  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);

  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];
    while (temp) {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}

struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
struct queue* createQueue() {
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
  q->rear = -1;
  return q;
}
int isEmpty(struct queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}
void enqueue(struct queue* q, int value) {
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}
int dequeue(struct queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
  return item;
}
void printQueue(struct queue* q) {
  int i = q->front;

  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
  }
}

int main() {
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;
}
```

OUTPUT

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100
struct Node {
    int vertex;
    struct Node* next;
};
struct Graph {
    int numVertices;
    struct Node* adjList[MAX_VERTICES];
};
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    for (int i = 0; i < numVertices; ++i) {
        graph->adjList[i] = NULL;
    }

    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = dest;
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;
}
void DFS(struct Graph* graph, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    struct Node* temp = graph->adjList[vertex];
    while (temp != NULL) {
        int adjVertex = temp->vertex;
        if (!visited[adjVertex]) {
            DFS(graph, adjVertex, visited);
        }
        temp = temp->next;
    }
}

int main() {
    int numVertices = 4;
    struct Graph* graph = createGraph(numVertices);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 0);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 3);

    int visited[MAX_VERTICES] = {0};
    printf("Depth First Traversal (starting from vertex 2):\n");
    DFS(graph, 2, visited);

    return 0;
}
```

OUTPUT

```
Depth First Traversal (starting from vertex 2):
2 3 0 1
```

2. **Topological Sort**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct AdjListNode {
    int dest;
    struct AdjListNode* next;
} AdjListNode;

typedef struct AdjList {
    AdjListNode* head;
} AdjList;

typedef struct Graph {
    int V;
    AdjList* array;
} Graph;

AdjListNode* newAdjListNode(int dest) {
    AdjListNode* newNode = (AdjListNode*)malloc(sizeof(AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

Graph* createGraph(int V) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->V = V;
    graph->array = (AdjList*)malloc(V * sizeof(AdjList));
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
}

void addEdge(Graph* graph, int src, int dest) {
    AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

void printGraph(Graph* graph) {
    for (int v = 0; v < graph->V; ++v) {
        AdjListNode* pCrawl = graph->array[v].head;
        printf("\nAdjacency list of vertex %d\nhead", v);
        while (pCrawl) {
            printf(" -> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

void DFSUtil(Graph* graph, int v, int visited[]) {
    visited[v] = 1;
    printf("%d ", v);
    AdjListNode* adjList = graph->array[v].head;
    while (adjList) {
        int connectedVertex = adjList->dest;
        if (!visited[connectedVertex])
            DFSUtil(graph, connectedVertex, visited);
        adjList = adjList->next;
    }
}

void DFS(Graph* graph, int startVertex) {
    int* visited = (int*)malloc(graph->V * sizeof(int));
    for (int i = 0; i < graph->V; i++)
        visited[i] = 0;
    DFSUtil(graph, startVertex, visited);
    free(visited);
}

int main() {
    int V = 5;
    Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printf("Graph adjacency list representation:\n");
    printGraph(graph);
    printf("\nDFS starting from vertex 0:\n");
    DFS(graph, 0);

    return 0;
}
```

OUTPUT

```
Graph adjacency list representation:

Adjacency list of vertex 0
head -> 4 -> 1

Adjacency list of vertex 1
head -> 4 -> 3 -> 2 -> 0

Adjacency list of vertex 2
head -> 3 -> 1

Adjacency list of vertex 3
head -> 4 -> 2 -> 1

Adjacency list of vertex 4
head -> 3 -> 1 -> 0

DFS starting from vertex 0:
0 4 3 2 1
```

3. **Prims**

```
#include <stdio.h>
#include <limits.h>

#define MAX_VERTICES 100
int minKey(int key[], int mstSet[], int vertices) {
    int min = INT_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}
void printMST(int parent[], int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}
void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int parent[MAX_VERTICES];
    int key[MAX_VERTICES];
    int mstSet[MAX_VERTICES];
    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < vertices - 1; count++) {

        int u = minKey(key, mstSet, vertices);
        mstSet[u] = 1;
        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, graph, vertices);
}

int main() {
    int vertices;
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];
    printf("Input the adjacency matrix for the graph:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    primMST(graph, vertices);

    return 0;
}
```

OUTPUT

```
Input the number of vertices: 5
Input the adjacency matrix for the graph:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Edge   Weight
0 - 1    10
3 - 2    20
0 - 3    30
2 - 4    10
```