1.Array Implementation of stack.

```
#include <stdio.h>
#include <conio.h> // For getch() function in Turbo C
#include <stdlib.h> // For exit() function

#define MAX 5 // Define maximum size of the stack

int stack[MAX]; // Array to store stack elements
int top = -1;   // Initialize top of stack as -1 (empty stack)

// Function prototypes
void push(int);
void pop();
void display();

int main() { // Corrected from void main to int main
  int choice, value;

  printf("ARRAY IMPLEMENTATION OF STACK\n\n");

  while (1) {
    printf("*************************\n");
    printf("*       MAIN MENU      *\n");
    printf("*************************\n");
    printf("1. Push\n");
    printf("2. Pop\n");
    printf("3. Display\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
```

```
        switch (choice) {
            case 1: // Push operation
                printf("Enter a value to push: ");
                scanf("%d", &value);
                push(value);
                break;

            case 2: // Pop operation
                pop();
                break;

            case 3: // Display stack
                display();
                break;

            case 4: // Exit program
                printf("Thank you for using the program. Exiting...\n");
                getch();  // Used to keep the window open in Turbo C
                return 0;  // Exit the program
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0; // Return from main function
}

// Push function
void push(int value) {
```

```c
    if (top == MAX - 1) {
        printf("\nStack Overflow! Cannot push %d.\n", value);
    } else {
        top++;
        stack[top] = value;
        printf("\n%d pushed onto the stack.\n", value);
    }
}

// Pop function
void pop() {
    if (top == -1) {
        printf("\nStack Underflow! No elements to pop.\n");
    } else {
        printf("\n%d popped from the stack.\n", stack[top]);
        top--;
    }
}

// Display function
void display() {
    int i;  // Declare 'i' here for the loop
    if (top == -1) {
        printf("\nStack is empty.\n");
    } else {
        printf("\nStack contents:\n");
        for (i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}
```

2.Implementation of representation of graph.

```
#include <stdio.h>
#include <conio.h> // Required for Turbo C

#define V 5

// Initialize matrix to 0
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++) {
    for (j = 0; j < V; j++) {
      arr[i][j] = 0; // Explicitly set each element to 0
    }
  }
}

// Add edge: Set arr[src][dest] = 1
void addEdge(int arr[][V], int src, int dest) {
  arr[src][dest] = 1;
}

// Print adjacency matrix
void printAdjMatrix(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++) {
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}
```

```
int main() {
    int adjMatrix[V][V]; // Declare adjacency matrix
    init(adjMatrix);     // Initialize matrix to 0

    // Add edges to adjacency matrix
    addEdge(adjMatrix, 0, 1);
    addEdge(adjMatrix, 0, 2);
    addEdge(adjMatrix, 0, 3);
    addEdge(adjMatrix, 1, 3);
    addEdge(adjMatrix, 1, 4);
    addEdge(adjMatrix, 2, 3);
    addEdge(adjMatrix, 3, 4);

    // Print adjacency matrix
    printf("Adjacency Matrix:\n");
    printAdjMatrix(adjMatrix);

    getch(); // Pause to keep the output visible in Turbo C
    return 0;
}
```

3.Implementation of topological sort.

```
#include<stdio.h>
#include<conio.h>  // Required for Turbo C to use getch()

int main()
{
   int i, j, k, n, a[10][10], indeg[10], flag[10], count = 0;

   printf("Enter the number of vertices:\n");
   scanf("%d", &n);

   printf("Enter the adjacency matrix:\n");
   for (i = 0; i < n; i++) {
      printf("Enter row %d:\n", i + 1);
      for (j = 0; j < n; j++) {
         scanf("%d", &a[i][j]);
      }
   }

   // Initialize indegree and flag arrays
   for (i = 0; i < n; i++) {
      indeg[i] = 0;
      flag[i] = 0;
   }

   // Calculate indegree of each vertex
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         indeg[i] += a[j][i];
      }
   }
```

```
    printf("\nThe topological order is: ");
    while (count < n) {
        for (k = 0; k < n; k++) {
            if (indeg[k] == 0 && flag[k] == 0) {
                printf("%d ", (k + 1));
                flag[k] = 1;

                // Decrease indegree of connected vertices
                for (i = 0; i < n; i++) {
                    if (a[k][i] == 1) {
                        indeg[i]--;
                    }
                }
            }
        }
        count++;
    }

    getch();  // Required for Turbo C to keep the output screen visible
    return 0;
}
```

4.Implementation of Dijkstra's algorithm.

```
#include <stdio.h>
#include <conio.h> // Required for Turbo C
#define MAX 100
#define INF 9999 // Define a large constant for infinity

int n;          // Number of vertices in the graph
int graph[MAX][MAX]; // Adjacency matrix representation of the graph

// Function to find the vertex with the minimum distance
int minDistance(int dist[], int visited[]) {
   int min = INF, min_index = -1;
   int v;
   for (v = 0; v < n; v++) {
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   }
   return min_index;
}

// Function to implement Dijkstra's algorithm
void dijkstra(int src) {
   int dist[MAX];
   int visited[MAX];
   int i, count, u, v;

   // Initialize distances and visited array
   for (i = 0; i < n; i++) {
      dist[i] = INF;
```

```
        visited[i] = 0;
    }
    dist[src] = 0;

    for (count = 0; count < n - 1; count++) {
        u = minDistance(dist, visited);
        visited[u] = 1;

        for (v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print the result
    printf("Vertex \t Distance from Source\n");
    for (i = 0; i < n; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

int main() {
    int edges, u, v, w, src;
    int i, j, m;

    // Input the number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the number of edges: ");
```

```
    scanf("%d", &edges);

    // Initialize the adjacency matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
    }

    // Input edges
    for (m = 0; m < edges; m++) {
        printf("Enter edge (u v w): ");
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
        graph[v][u] = w; // Uncomment this line if the graph is undirected
    }

    // Input the source vertex
    printf("Enter the source vertex: ");
    scanf("%d", &src);

    // Call Dijkstra's algorithm
    dijkstra(src);

    getch(); // Pause to keep the output visible
    return 0;
}
```

5.Implementation of Iterative recursive algorithms

```
#include <stdio.h>
#include <conio.h> // Required for Turbo C

// Method to find factorial using recursion
int factorialUsingRecursion(int n) {
   if (n == 0) {
      return 1;
   }
   return n * factorialUsingRecursion(n - 1); // Recursive call
}

// Method to find factorial using iteration
int factorialUsingIteration(int n) {
   int res = 1;
   int i; // Declare variables at the beginning
   for (i = 2; i <= n; i++) {
      res *= i; // Iterative multiplication
   }
   return res;
}

// Main function
int main() {
   int num = 5; // Input number

   // Output factorial using recursion
   printf("Factorial of %d using Recursion is: %d\n", num, factorialUsingRecursion(num));

   // Output factorial using iteration
   printf("Factorial of %d using Iteration is: %d\n", num, factorialUsingIteration(num));
```

```
    getch(); // Pause to keep the output visible
    return 0;
}
```

6. Implementation of Matrix chain Multiplication.

```
#include <stdio.h>
#include <conio.h> // Required for Turbo C

#define MAX 100
#define INF 9999 // Replace INT_MAX with a large constant

// Function to compute the minimum number of multiplications
int MatrixChainOrder(int p[], int n) {
  int m[MAX][MAX]; // Using static memory for compatibility
  int i, j, k, L, q;

  // Initialize the diagonal of the matrix to zero
  for (i = 1; i < n; i++) {
    m[i][i] = 0;
  }

  // L is the chain length
  for (L = 2; L < n; L++) {
    for (i = 1; i < n - L + 1; i++) {
      j = i + L - 1;
      m[i][j] = INF; // Initialize with a large value
      for (k = i; k <= j - 1; k++) {
        // Calculate the cost
        q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
        if (q < m[i][j]) {
          m[i][j] = q; // Update minimum cost
        }
      }
    }
  }
```

```
    return m[1][n - 1]; // Return the result
}

// Main function
int main() {
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Output the result
    printf("Minimum number of multiplications is %d", MatrixChainOrder(arr, size));

    getch(); // Pause to keep the output visible
    return 0;
}
```