TABLE OF CONTENT

6. WRITE A PROGRAM TO DISPLAY THE USE OF DIJKSTRA ALGORITHM.

```
#include <limits.h>
#include <stdio.h>
#define V 9
int minDistance(int dist[], bool sptSet[])
{
        // Initialize min value
        int min = INT_MAX, min_index;

        for (int v = 0; v < V; v++)
                if (sptSet[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;

        return min_index;
}
void printSolution(int dist[], int n)
{
        printf("Vertex Distance from Source\n");
        for (int i = 0; i < V; i++)
                printf("\t%d \t\t\t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src)
{
        int dist[V]; // The output array. dist[i] will hold the
                                        // shortest
        // distance from src to i

        bool sptSet[V]; // sptSet[i] will be true if vertex i is
                                                // included in shortest
        // Initialize all distances as INFINITE and stpSet[] as
```

```
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet,
            // there is an edge from u to v, and total
            // weight of path from src to v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist, V);
}
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
```

```
                    { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                    { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                    { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                    { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                    { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                    { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                    { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                    { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
    dijkstra(graph, 0);
    return 0;
}
```

OUTPUT:

```
Vertex Distance from Source
    0               0
    1               4
    2               12
    3               19
    4               21
    5               11
    6               9
    7               8
    8               14
```

7. OBTAIN THE TOPOLOGICAL ORDERING OF VERTICES IN A GIVEN DIGRAPH.

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph {
	int V; // No. of vertices'
	list<int>* adj;
	void topologicalSortUtil(int v, bool visited[], stack<int>& Stack);

public:
	Graph(int V); // Constructor
	void addEdge(int v, int w);
	void topologicalSort();
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v’s list.
}
void Graph::topologicalSortUtil(int v, bool visited[],
						stack<int>& Stack)
{
```

```
        visited[v] = true;
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        topologicalSortUtil(*i, visited, Stack);
Stack.push(v);
}
void Graph::topologicalSort()
{
        stack<int> Stack;
        bool* visited = new bool[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;

        for (int i = 0; i < V; i++)
                if (visited[i] == false)
                        topologicalSortUtil(i, visited, Stack);

        while (Stack.empty() == false) {
                cout << Stack.top() << " ";
                Stack.pop();
        }
}
int main()
{
        Graph g(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
```

```
        g.addEdge(2, 3);
        g.addEdge(3, 1);
        cout << "Topological Sort of the given graph: ";
        g.topologicalSort
        return 0;
}
```

OUTPUT:

```
Topological Sort of the given graph: 5 4 2 3 1 0
```

8. PRINT ALL THE NODES REACHABLE FROM A GIVEN STARTING NODE IN A DIGRAPH USING BFS METHOD.

```
#include <bits/stdc++.h>
using namespace std;
class Graph
{
public:
	int V;
	list<int> *adj;

	Graph(int ); // Constructor
	void addEdge(int, int);
	vector<int> BFS(int, int, int []);
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V+1];
}
void Graph::addEdge(int u, int v)
{
	adj[u].push_back(v); // Add w to v's list.
	adj[v].push_back(u); // Add v to w's list.
}
vector<int> Graph::BFS(int componentNum, int src,
									int visited[])
{
	queue<int> queue;
	queue.push(src);
```

```
        visited[src] = componentNum;
        // Vector to store all the reachable nodes from 'src'
        vector<int> reachableNodes;

        while(!queue.empty())
        {
                int u = queue.front();
                queue.pop();
                reachableNodes.push_back(u);

                for (auto itr = adj[u].begin();
                                itr != adj[u].end(); itr++)
                {
                        if (!visited[*itr])
                        {
                                visited[*itr] = componentNum;
                                queue.push(*itr);
                        }
                }
        }
        return reachableNodes;
}
void displayReachableNodes(int n,
                        unordered_map <int, vector<int> > m)
{
        vector<int> temp = m[n];
        for (int i=0; i<temp.size(); i++)
                cout << temp[i] << " ";

        cout << endl;
```

```
}
void findReachableNodes(Graph g, int arr[], int n)
{
        int V = g.V;
        int visited[V+1];
        memset(visited, 0, sizeof(visited));
        unordered_map <int, vector<int> > m;
        int componentNum = 0;
        for (int i = 0 ; i < n ; i++)
        {
                int u = arr[i];
                if (!visited[u])
                {
                        componentNum++;
                        m[visited[u]] = g.BFS(componentNum, u, visited);
                }
                cout << "Reachable Nodes from " << u <<" are\n";
                displayReachableNodes(visited[u], m);
        }
}
int main()
{
        int V = 7;
        Graph g(V);
        g.addEdge(1, 2);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(3, 1);
        g.addEdge(5, 6);
        g.addEdge(5, 7);
```

```
    int arr[] = {2, 4, 5};
    int n = sizeof(arr)/sizeof(int);
    findReachableNodes(g, arr, n);

    return 0;
}
```

OUTPUT:

```
Reachable Nodes from 2 are
2 1 3 4
Reachable Nodes from 4 are
2 1 3 4
Reachable Nodes from 5 are
5 6 7
```

9.CHECK WHETHER A GIVEN GRAPH IS CONNECTED OR NOT USING DFS METHOD.

```
#include<iostream>
#define NODE 5
using namespace std;
int graph[NODE][NODE] = {{0, 1, 0, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 0, 1, 1},
  {1, 0, 0, 0, 0},
  {0, 1, 0, 0, 0}};
void traverse(int u, bool visited[]) {
  visited[u] = true;      //mark v as visited
  for(int v = 0; v<NODE; v++) {
    if(graph[u][v]) {
      if(!visited[v])
        traverse(v, visited);
    }
  }
}
bool isConnected() {
  bool *vis = new bool[NODE];
  //for all vertex u as start point, check whether all nodes are visible or not
  for(int u; u < NODE; u++) {
    for(int i = 0; i<NODE; i++)
      vis[i] = false;    //initialize as no node is visited
      traverse(u, vis);
      for(int i = 0; i<NODE; i++) {
        if(!vis[i])        //if there is a node, not visited by traversal, graph is not connected
          return false;
      }
  }
```

```
  return true;
}
int main() {
  if(isConnected())
    cout << "The Graph is connected.";
  else
    cout << "The Graph is not connected.";
}
```

OUTPUT:

```
The Graph is connected.
```

10. IMPLEMENT 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING.

```
#include <iostream>
using namespace std;
int max(int x, int y) {
  return (x > y) ? x : y;
}
int knapSack(int W, int w[], int v[], int n) {
  int i, wt;
  int K[n + 1][W + 1];
  for (i = 0; i <= n; i++) {
    for (wt = 0; wt <= W; wt++) {
      if (i == 0 || wt == 0)
      K[i][wt] = 0;
      else if (w[i - 1] <= wt)
        K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]);
      else
     K[i][wt] = K[i - 1][wt];
    }
  }
  return K[n][W];
}
int main() {
  cout << "Enter the number of items in a Knapsack:";
  int n, W;
  cin >> n;
  int v[n], w[n];
  for (int i = 0; i < n; i++) {
    cout << "Enter value and weight for item " << i << ":";
    cin >> v[i];
    cin >> w[i];
```

```
  }
  cout << "Enter the capacity of knapsack";
  cin >> W;
  cout << knapSack(W, w, v, n);
  return 0;
}
```

OUTPUT:

```
Enter the number of items in a Knapsack:4
Enter value and weight for item 0:10
50
Enter value and weight for item 1:20
60
Enter value and weight for item 2:30
70
Enter value and weight for item 3:40
90
Enter the capacity of knapsack100
40
```

11. FIND A SUBSET OF A GIVEN SET S = {S1, S2, SN} OF N POSITIVE INTEGERS WHOSE SUM IS EQUAL TO A GIVEN POSITIVE INTEGER D.

```
#include<stdio.h>
int d;
void sum(int,int,int[]);
int main()
{
	int n,w[100],i;
	printf("Enter the no of objects\n");
	scanf("%d",&n);

	printf("Enter the elements in increasing order\n");
	for(i=1;i<=n;i++)
		scanf("%d",&w[i]);
	printf("Enter the maximum capacity\n");
	scanf("%d",&d);
	sum(n,d,w);
}
void sum(int n,int d,int w[])
{
	int x[100],s,k,i,found=0;
	for(i=1;i<=n;i++)
		x[i]=0;
	s=0;
	k=1;
	x[k]=1;
	while(1)
	{
		if(k <= n && x[k]==1)
		{
```

```
        if(s+w[k] == d)
        {
        found=1;
            printf("The solution is\n");
            for(i=1;i<=n;i++)
            {
                if(x[i]==1)
                printf("%d\t",w[i]);
            }
            printf("\n");
            x[k]=0;
        }

        else if(s+w[k] < d)
                s+=w[k];
        else
        {
            x[k]=0;
        }
    }
    else
    {
        k--;
        while(k>0 && x[k]==0)
             k--;

        if(k<=0)
        {

            break;
```

```
                }
                x[k]=0;
                s=s-w[k];
            }
            k=k+1;
            x[k]=1;
 }
 if(!found)
 printf("no solution\n");
}
```

OUTPUT:

```
Enter the no of objects
5
Enter the elements in increasing order
1 2 3 4 5
Enter the maximum capacity
5
The solution is
1    4
The solution is
2    3
The solution is
5
```

12. COMPUTE THE TRANSITIVE CLOSURE OF A GIVEN DIRECTED GRAPH USING WARSHALL'S ALGORITHM.

```
#include<stdio.h>
const int MAX = 100;
void WarshallTransitiveClosure(int graph[MAX][MAX], int numVert);
int main(void)
{
    int i, j, numVert;
    int graph[MAX][MAX];

    printf("Warshall's Transitive Closure\n");
    printf("Enter the number of vertices : ");
    scanf("%d",&numVert);

    printf("Enter the adjacency matrix :-\n");
    for (i=0; i<numVert; i++)
        for (j=0; j<numVert; j++)
            scanf("%d",&graph[i][j]);

    WarshallTransitiveClosure(graph, numVert);

    printf("\nThe transitive closure for the given graph is :-\n");
    for (i=0; i<numVert; i++)
    {
        for (j=0; j<numVert; j++)
        {
            printf("%d\t",graph[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
void WarshallTransitiveClosure(int graph[MAX][MAX], int numVert)
{
   int i,j,k;

   for (k=0; k<numVert; k++)
   {
      for (i=0; i<numVert; i++)
      {
         for (j=0; j<numVert; j++)
         {
            if (graph[i][j] || (graph[i][k] && graph[k][j]))
               graph[i][j] = 1;
         }
      }
   }
}
```

OUTPUT:

```
Warshall's Transitive Closure
Enter the number of vertices : 4
Enter the adjacency matrix :-
0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0

The transitive closure for the given graph is :-
1    0    1    0
0    1    0    1
1    0    1    0
0    1    0    1
```

13. IMPLEMENT ALL-PAIRS SHORTEST PATH PROBLEMS USING FLOYD'S ALGORITHM.

```
#include<iostream>
#include<iomanip>
#define NODE 7
#define INF 999
using namespace std;
int costMat[NODE][NODE] = {
  {0, 3, 6, INF, INF, INF, INF},
  {3, 0, 2, 1, INF, INF, INF},
  {6, 2, 0, 1, 4, 2, INF},
  {INF, 1, 1, 0, 2, INF, 4},
  {INF, INF, 4, 2, 0, 2, 1},
  {INF, INF, 2, INF, 2, 0, 1},
  {INF, INF, INF, 4, 1, 1, 0}
};
void floydWarshal(){
  int cost[NODE][NODE]; //defind to store shortest distance from any node to any node
  for(int i = 0; i<NODE; i++)
    for(int j = 0; j<NODE; j++)
      cost[i][j] = costMat[i][j]; //copy costMatrix to new matrix
      for(int k = 0; k<NODE; k++){
        for(int i = 0; i<NODE; i++)
          for(int j = 0; j<NODE; j++)
            if(cost[i][k]+cost[k][j] < cost[i][j])
              cost[i][j] = cost[i][k]+cost[k][j];
  }
  cout << "The matrix:" << endl;
  for(int i = 0; i<NODE; i++){
    for(int j = 0; j<NODE; j++)
      cout << setw(3) << cost[i][j];
    cout << endl;
```

```
  }
}
int main(){
  floydWarshal();
}
```

OUTPUT:

```
The matrix:
  0   3   5   4   6   7   7
  3   0   2   1   3   4   4
  5   2   0   1   3   2   3
  4   1   1   0   2   3   3
  6   3   3   2   0   2   1
  7   4   2   3   2   0   1
  7   4   3   3   1   1   0
```

14. IMPLEMENT N QUEEN'S PROBLEM USING BACK TRACKING.

```
#include<iostream>
using namespace std;
int grid[10][10];
void print(int n) {
    for (int i = 0;i <= n-1; i++) {
        for (int j = 0;j <= n-1; j++) {
                cout <<grid[i][j]<< " ";
        }
        cout<<endl;
    }
    cout<<endl;
    cout<<endl;
}
bool isSafe(int col, int row, int n) {
    for (int i = 0; i < row; i++) {
        if (grid[i][col]) {
            return false;
        }
    }
    for (int i = row,j = col;i >= 0 && j >= 0; i--,j--) {
        if (grid[i][j]) {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j < n; j++, i--) {
        if (grid[i][j]) {
            return false;
        }
    }
```

```
    return true;
}
bool solve (int n, int row) {
    if (n == row) {
        print(n);
        return true;
    }
    bool res = false;
    for (int i = 0;i <=n-1;i++) {
        if (isSafe(i, row, n)) {
            grid[row][i] = 1;
            res = solve(n, row+1) || res;//if res ==false then backtracking will occur
            grid[row][i] = 0;
        }
    }
    return res;
}
int main()
{
  ios_base::sync_with_stdio(false);
    cin.tie(NULL);
        int n;
        cout<<"Enter the number of queen"<<endl;
        cin >> n;
        for (int i = 0;i < n;i++) {
            for (int j = 0;j < n;j++) {
                grid[i][j] = 0;
            }
        }
        bool res = solve(n, 0);
```

```
    if(res == false) {
      cout << -1 << endl; //if there is no possible solution
    } else {
      cout << endl;
    }
 return 0;
}
```

OUTPUT:

```
Enter the number of queen
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0


0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```