Table of Contents

Exp.No : 01 **Date : 14.03.2024**

Quick Short Algorithm

AIM :

To write a program for The Quick Sort algorithm is a sorting algorithm using C++ programming Language

ALGORITHM :

Step 1 :swap function: Swaps two integers.
Step 2: partition function: Places the pivot at its correct position and partitions the array around it.
Step 3 :quickSort function: Recursively sorts the array by partitioning it.
Step 4: printArray function: Prints the array elements.
Step 5: main function: Initializes an array, prints it, sorts it using Quick Sort, and prints the sorted array.

PROGRAM :

```
#include <iostream.h>

#include <conio.h>


// Function to swap two elements

void swap(int* a, int* b) {

int t = *a;

   *a = *b;

   *b = t;

}


// Partition function to place the pivot element at the correct position

int partition(intarr[], int low, int high) {

int pivot = arr[high]; // Pivot element

int i = (low - 1); // Index of the smaller element


for (int j = low; j <= high - 1; j++) {

      // If the current element is smaller than or equal to the pivot

if (arr[j] <= pivot) {

i++; // Increment the index of the smaller element

swap(&arr[i], &arr[j]);

      }

   }

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}
```

```
// QuickSort function
voidquickSort(intarr[], int low, int high) {
if (low < high) {
        // Partition the array around the pivot element
int pi = partition(arr, low, high);

        // Recursively sort the sub-arrays
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
voidprintArray(intarr[], int size) {
for (int i = 0; i < size; i++) {
cout<<arr[i] << " ";
    }
cout<<endl;
}

void main() {
clrscr(); // Clear the screen
intarr[] = {10, 7, 8, 9, 1, 5};
int n = sizeof(arr) / sizeof(arr[0]);
cout<< "Unsorted array: \n";
printArray(arr, n);
quickSort(arr, 0, n - 1);
```

```
cout<< "Sorted array: \n";
printArray(arr, n);
getch(); // Wait for a key press
}
```

OUTPUT :

RESULT:

Thus the Program has been successfully completed.

Exp.No : 02 **Date : 21.03.2024**

DIJKSTRA’S ALGORITHM

AIM:

To implement dijkstra’s algorithm using C++ programming language

ALGORITHM :

STEP 1: minDistance function: Finds the vertex with the minimum distance value that hasn't been processed yet.

STEP 2: PrintSolution function: Prints the shortest distances from the source vertex to all other vertices.
STEP 3 :dijkstra function: Implements Dijkstra's algorithm to find the shortest path from the source vertex to all other vertices in the graph.
STEP 4 :main function: Defines the graph as an adjacency matrix, calls the **dijkstra** function, and waits for a key press before terminating.

PROGRAM :

```
#include <iostream.h>

#include <conio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 9

// Function to find the vertex with the minimum distance value

intminDistance(intdist[], bool sptSet[]) {

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;

returnmin_index;

}

// Function to print the constructed distance array

voidprintSolution(intdist[], int n) {

cout<< "Vertex \t Distance from Source\n";

for (int i = 0; i < n; i++)

cout<< i << " \t\t " <<dist[i] << "\n";

}

// Function that implements Dijkstra's single source shortest path algorithm
```

```
voiddijkstra(int graph[V][V], intsrc) {
intdist[V]; // Output array. dist[i] will hold the shortest distance from src to i
boolsptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree

   // Initialize all distances as INFINITE and stpSet[] as false
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 the shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
      // Pick the minimum distance vertex from the set of vertices not yet processed
int u = minDistance(dist, sptSet);

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

      // Update dist value of the adjacent vertices of the picked vertex
for (int v = 0; v < V; v++)
         // Update dist[v] only if it 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];
   }
```

```
    // Print the constructed distance array
printSolution(dist, V);
}

void main() {
clrscr(); // Clear the screen

    // Example graph represented as an adjacency matrix
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); // Call Dijkstra's algorithm with source vertex 0

getch(); // Wait for a key press
}1
```

Output:

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

Result :

Thus the given program is executed and output Is verified successfully.

Exp.No :03 **Date : 28.03.2024**

Strassen Matrix Algorithm

AIM :

To implement Strassen Matrix Algorithm using C++ language

ALGORITHM:

Step 1: Add function: Adds two 2x2 matrices.
Step 2 :subtract function: Subtracts one 2x2 matrix from another.
Step 3 :strassen function: Implements Strassen's matrix multiplication algorithm for 2x2 matrices.
Step 4 :printMatrix function: Prints a 2x2 matrix.
Step 5 :main function: Initializes two 2x2 matrices, calls the **strassen** function to multiply them, and prints the result.

PROGRAM :

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
void add(int A[2][2], int B[2][2], int C[2][2]) {
for (int i = 0; i < 2; i++)
for (int j = 0; j < 2; j++)
        C[i][j] = A[i][j] + B[i][j];
}
void subtract(int A[2][2], int B[2][2], int C[2][2]) {
for (int i = 0; i < 2; i++)
for (int j = 0; j < 2; j++)
        C[i][j] = A[i][j] - B[i][j];
}
voidstrassen(int A[2][2], int B[2][2], int C[2][2]) {
int M1[2][2], M2[2][2], M3[2][2], M4[2][2], M5[2][2], M6[2][2], M7[2][2];
int temp1[2][2], temp2[2][2];
 // M1 = (A11 + A22) * (B11 + B22)
add(A, A, temp1);
add(B, B, temp2);
multiply(temp1, temp2, M1);
 // M2 = (A21 + A22) * B11
add(A + 1, A + 3, temp1);
multiply(temp1, B, M2);
// M3 = A11 * (B12 - B22)
subtract(B + 1, B + 3, temp2);
multiply(A, temp2, M3);
```

```
    // M4 = A22 * (B21 - B11)
subtract(B + 2, B, temp2);
multiply(A + 3, temp2, M4);

    // M5 = (A11 + A12) * B22
add(A, A + 1, temp1);
multiply(temp1, B + 3, M5);

    // M6 = (A21 - A11) * (B11 + B12)
subtract(A + 2, A, temp1);
add(B, B + 1, temp2);
multiply(temp1, temp2, M6);

    // M7 = (A12 - A22) * (B21 + B22)
subtract(A + 1, A + 3, temp1);
add(B + 2, B + 3, temp2);
multiply(temp1, temp2, M7);

    // C11 = M1 + M4 - M5 + M7
add(M1, M4, temp1);
subtract(temp1, M5, temp2);
add(temp2, M7, C);

    // C12 = M3 + M5
add(M3, M5, C + 1);
```

```
    // C21 = M2 + M4
add(M2, M4, C + 2);

    // C22 = M1 - M2 + M3 + M6
subtract(M1, M2, temp1);
add(temp1, M3, temp2);
add(temp2, M6, C + 3);
}
voidprintMatrix(int matrix[2][2]) {
for (int i = 0; i < 2; i++) {
for (int j = 0; j < 2; j++)
cout<< matrix[i][j] << " ";
cout<<endl;
    }
}
void main() {
clrscr(); // Clear the screen

int A[2][2] = { {1, 2}, {3, 4} };
int B[2][2] = { {5, 6}, {7, 8} };
int C[2][2]; // Result matrix
strassen(A, B, C);
cout<< "Resultant matrix: \n";
printMatrix(C);
getch(); // Wait for a key press
}
```

Output :

```
Resultant matrix:
19 22
43 50
```

Result :

Thus the given program is executed and output Is verified successfully.

Exp.No :04 **Date : 04.04.2024**

SHORTST PATHS USING FLOYD'S ALGOITHM

AIM:

To write a c++ program to find shortest path using floyd's algorithm.

ALGORITHM:

1.intillalize the solution matrix same as the input graph matrix as a first step.

2.Then update the solution matrix by considering all vertices as an intermediate vertex.

3.The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

4.When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.

5.For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.

6.K is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.

7.k is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j],** if **dist[i][j] >dist[i][k] + dist[k][j].**

Program :

```
#include <iostream.h>

#include <conio.h>

voidfloyds(int b[][7])

{

int i, j, k;

   for (k = 0; k < 7; k++)

   {

      for (i = 0; i < 7; i++)

      {

         for (j = 0; j < 7; j++)

         {

if ((b[i][k] * b[k][j] != 0) && (i != j))

            {

               if ((b[i][k] + b[k][j] < b[i][j]) || (b[i][j] == 0))

               {

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

               }

            }

         }

      }

   }

   for (i = 0; i < 7; i++)

   {

cout<<"\nMinimum Cost With Respect to Node:"<<i<<endl;

      for (j = 0; j < 7; j++)

      {
```

```
cout<<b[i][j]<<"\t";
        }


    }
}
int main()
{
int b[7][7];
cout<<"ENTER VALUES OF ADJACENCY MATRIX\n\n";
    for (int i = 0; i < 7; i++) {
cout<<"enter values for "<<(i+1)<<" row"<<endl;
        for (int j = 0; j < 7; j++)
        {
cin>>b[i][j];
        }  }
floyds(b);
getch();
return(0);
}
```

Output :

```
enter values for 2 row
enter values for 3 row
enter values for 4 row
enter values for 5 row
enter values for 6 row
enter values for 7 row

Minimum Cost With Respect to Node:0
4       -76     -96     -68     -72     -140    -87
Minimum Cost With Respect to Node:1
-73     6       -97     -69     -73     -141    -88
Minimum Cost With Respect to Node:2
-53     -57     4945    -49     -53     -121    -68
Minimum Cost With Respect to Node:3
-50     -54     -74     3292    -50     -118    -65
Minimum Cost With Respect to Node:4
-77     -81     -101    -73     520     -145    -92
Minimum Cost With Respect to Node:5
1       -3      -23     5       1       1300    -14
Minimum Cost With Respect to Node:6
15      11      -9      19      15      -53     8115    ENTER VALUES OF ADJACENC
Y MATRIX

enter values for 1 row
```

Result :

Thus the given Program is executed and output is verified successfully.

Exp.No : 05 **Date :18.04.2024**

TRAVELLING SALESPERSON PROBLEM

AIM:

To write C++ program Find the Optimal solution for travelling salesperson problem using approximation algorithm.

ALGORITHM:

STEP-1: Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.

STEP-2:The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.

STEP-3:The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).

STEP-4:Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.

STEP-5:Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.

STEP-6:However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

PROGRAM:

```
#include<iostream>
using namespace std;
intary[10][10],completed[10],n,cost=0;
voidtakeInput()
{
inti,j;
cout<<"Enter the number of villages: ";
cin>>n;
cout<<"\nEnter the Cost Matrix\n";
for(i=0;i <n;i++)
{
cout<<"\nEnter Elements of Row: "<<i+1<<"\n";
for( j=0;j <n;j++)
cin>>ary[i][j];
completed[i]=0;
}
cout<<"\n\nThe cost list is:";
for( i=0;i <n;i++)
{
cout<<"\n";

for(j=0;j <n;j++)
cout<<"\t"<<ary[i][j];
}
}
int least(int c)
```

```
{
inti,nc=999;
int min=999,kmin;
for(i=0;i <n;i++)
{
if((ary[c][i]!=0)&&(completed[i]==0))
if(ary[c][i]+ary[i][c] < min)
{
min=ary[i][0]+ary[c][i];
kmin=ary[c][i];
nc=i;
}
}
if(min!=999)
cost+=kmin;
return nc;
}
voidmincost(int city)
{
inti,ncity;
completed[city]=1;
cout<<city+1<<"--->";
ncity=least(city);
if(ncity==999)
{
ncity=0;
cout<<ncity+1;
```

```
cost+=ary[city][ncity];

return;
}
mincost(ncity);
}
int main()
{
takeInput();
cout<<"\n\nThe Path is:\n";
mincost(0); //passing 0 because starting vertex
cout<<"\n\nMinimum cost is "<<cost;
return 0;
}
```

Output :

```
Enter the Cost Matrix

Enter Elements of Row: 1
4 2 1 0

Enter Elements of Row: 2
2 3 2 1

Enter Elements of Row: 3
1 4 3 2

Enter Elements of Row: 4
2 1 3 6


The cost list is:
        4       2       1       0
        2       3       2       1
        1       4       3       2
        2       1       3       6

The Path is:
1--->3--->4--->2--->1

Minimum cost is 6Enter the number of villages:
```

Result :

Thus the given program is executed and output is verified successfully.

Exp.No : 06 **Date :25.04.2024**

KNAPSACK PROBLEM USING GREEDY METHOD

Aim :

To write a C++ program to solve the knapsack problem using greedy method

Algorithm :

1.Sort the array in decreasing order using the value/weight ratio

2.Start taking the element having the maximum value/weight ratio

3.If the weight of the current item is less than the current knapsack capacity, add the whole item, or else add the portion of the item to the knapsack

4.Stop adding the elements when the capacity of the knapsack becomes 0

5.Choose the item that has the maximum value from the remaining items

6.Choose the lightest item from the remaining items which uses up capacity as slowly as possible

7.Put items into the bag until the next item on the list cannot fit

8.Try to fill any remaining capacity with the next item on the list that can fit

Program :

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

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
intknapSack(int W, intwt[], intval[], int n)
{

    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more
    // than Knapsack capacity W, then
    // this item cannot be included
    // in the optimal solution
if (wt[n - 1] > W)
        returnknapSack(W, wt, val, n - 1);

    // Return the maximum of two cases:
    // (1) nth item included
```

```
    // (2) not included
    else
            return max(
            val[n - 1]
                        + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
int profit[] = { 60, 100, 120 };
int weight[] = { 10, 20, 30 };
    int W = 50;
int n = sizeof(profit) / sizeof(profit[0]);
cout<<knapSack(W, weight, profit, n);
    return 0;
}
```

Output :

Result :

Thus given program travelling salesperson problem is executed successfully and output is verified

Exp.No: 07 **Date :02.05.2024**

Shortest path to other vertices using Dijkstra's Algorithm

AIM :

To implement shortest path to other vertices using Dijkstras algorithm by using C++ Language

ALGORITHM:

Step 1: minDistance function: Finds the vertex with the minimum distance value not yet included in the shortest path tree.

Step 2: printSolution function: Prints the distances of all vertices from the source vertex.

Step 3: dijkstra function: Initializes distances and shortest path tree set, then iteratively selects the vertex with the minimum distance to update adjacent vertices' distances.

Step 4: main function: Sets up the graph as an adjacency matrix.

Step 5: main function (continued): Calls `Dijkstra` to compute and print the shortest paths from the source vertex.

PROGRAM:

```
#include <iostream.h>

#include <conio.h>

#include <limits.h>


// Number of vertices in the graph

#define V 9


// Function to find the vertex with the minimum distance value

intminDistance(intdist[], bool sptSet[]) {

int min = INT_MAX, min_index;


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

if (!sptSet[v] &&dist[v] <= min)

min = dist[v], min_index = v;


returnmin_index;

}


// Function to print the constructed distance array

voidprintSolution(intdist[]) {

cout<< "Vertex \t Distance from Source\n";

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

cout<< i << " \t\t " <<dist[i] << "\n";

}


// Function that implements Dijkstra's single source shortest path algorithm
```

```
voiddijkstra(int graph[V][V], intsrc) {
intdist[V]; // Output array. dist[i] will hold the shortest distance from src to i
boolsptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree

   // Initialize all distances as INFINITE and sptSet[] as false
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 the shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
      // Pick the minimum distance vertex from the set of vertices not yet processed
int u = minDistance(dist, sptSet);

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

      // Update dist value of the adjacent vertices of the picked vertex
for (int v = 0; v < V; 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];
   }

   // Print the constructed distance array
printSolution(dist);
```

```
}

void main() {
clrscr(); // Clear the screen

   // Example graph represented as an adjacency matrix
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); // Call Dijkstra's algorithm with source vertex 0

getch(); // Wait for a key press
}
```

Output:

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

Result :

Thus the program is executed and the output is verified successfully.

Exp.No : 08 **Date : 9.05.2024**

MINIMUM COST SPANNING TREE USING KRUSKALS ALGORITHM

AIM:

To find the minimum cost spanning tree of a given undirected graph using kruskal's algorithm.

ALGORITHM:

STEP-1:.A ← Ø.

STEP-2:for each vertex v V[G].

STEP-3:do MAKE-SET(v).

STEP-4:sort the edges of E into nondecreasing order by weight w.

STEP-5:For each edge (u, v) E, taken in nondecreasing order by weight.

STEP-6:do if FIND-SET(u) ≠ FIND-SET(v).

STEP-7:then A ← A {(u, v)}.

STEP-8:UNION(u, v).

STEP-9:return A.

STEP-10:display the mincost as Minimum cost.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
inti,j,k,a,b,u,v,n,ne=1;
intmin,mincost=0,cost[9][9],parent[9];
int find(int);
intuni(int,int);
void main()
{
clrscr();
printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n)
{
```

```
for(i=1,min=999;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
u=find(u);
v=find(v);
if(uni(u,v))
{
printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
getch();
}
int find(int i)
{
while(parent[i])
```

```
i=parent[i];
return i;
}
intuni(inti,int j)
{
if(i!=j)
{
parent[j]=i;
return 1;
}
```

Output :

```
        Implementation of Kruskal's algorithm


Enter the no. of vertices
2

Enter the cost adjacency matrix
100 200
200 300

The edges of Minimum Cost Spanning Tree are


1 edge (1,2) =200

        Minimum cost = 200

_
```

RESULT:

Thus the program is executed and the output is verified successfully.

Exp.No:09 **Date:16.05.2024**

Implementation of the N Queens problem

Aim:

Implementation of the N Queens problem using C++ programming language.

ALGORITHM :

Step 1: printSolution function: Prints the current chessboard configuration, where 1s represent queens' positions and 0s represent empty squares.

Step 2 :isSafe function: Checks if it's safe to place a queen at a given position, ensuring no other queens threaten it horizontally, vertically, or diagonally.

Step 3: solveNQUtil function: Recursively solves the N Queens problem by trying to place queens column by column, ensuring each placement is safe.

Step 4: solveNQ function: Initializes the chessboard and invokes solveNQUtil to find a solution. If found, prints it; otherwise, indicates no solution exists.

Step 5 :main function: Initializes the chessboard with empty squares, calls solveNQ to find the solution, and manages the program flow.

Program:

```
#include <iostream.h>
#include <conio.h>

#define N 8

voidprintSolution(int board[N][N]) {
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++)
cout<< board[i][j] << " ";
cout<<endl;
    }
}

boolisSafe(int board[N][N], int row, int col) {
int i, j;

    // Check this row on the left side
for (i = 0; i < col; i++)
if (board[row][i])
return false;

    // Check upper diagonal on left side
for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
if (board[i][j])
return false;
```

```
    // Check lower diagonal on left side
for (i = row, j = col; j >= 0 && i < N; i++, j--)
if (board[i][j])
return false;

return true;
}

boolsolveNQUtil(int board[N][N], int col) {
    // If all queens are placed, return true
if (col >= N)
return true;

    // Consider this column and try placing this queen in all rows one by one
for (int i = 0; i < N; i++) {
        // Check if the queen can be placed on board[i][col]
if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
board[i][col] = 1;

            // Recur to place rest of the queens
if (solveNQUtil(board, col + 1))
return true;

            // If placing queen in board[i][col] doesn't lead to a solution, then remove the queen
from board[i][col]
board[i][col] = 0; // BACKTRACK
```

```
        }
    }

    // If the queen cannot be placed in any row in this column, return false
return false;
}

boolsolveNQ() {
int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0, 0} };

if (solveNQUtil(board, 0) == false) {
cout<< "Solution does not exist";
return false;
    }

printSolution(board);
return true;
}

void main() {
```

```
clrscr(); // Clear the screen


solveNQ();


getch(); // Wait for a key press
}
```

Output :

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

RESULT:

Thus the program is executed and the output is verified successfully