# INDEX

| S No. | Objective | Date | Signature |
|---|---|---|---|
| 1. | WAP to implement Quick sort algorithm for sorting a list of integers in ascending order | | |
| 2. | WAP to implement Merge sort algorithm for sorting a list of integers in ascending order. | | |
| 3. | WAP to implement the dfs algorithm for a graph. | | |
| 4. | WAP to implement the bfs algorithm for a graph. | | |
| 5. | WAP to implement backtracking algorithm for the N-queens problem | | |
| 6. | WAP to implement the backtracking algorithm for the sum of subsets problem. | | |
| 7. | WAP to implement the backtracking algorithm for the Hamiltonian Circuits problem | | |
| 8. | WAP to implement Knapsack Problem. | | |
| 9. | WAP to implement Dijkstra's algorithm for the Single source shortest path problem. | | |
| 10. | WAP that implements Prim's algorithm to generate minimum cost spanning tree. | | |

## 1. Write a program to implement Quick sort algorithm for sorting a list of integers in ascending order.

**THEORY – Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes O(nlogn) (big O notation) comparisons to sort n items. However, in the worst case, it makes O(n2) comparisons. Typically, quicksort is significantly faster in practice than other O(nlogn) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the probability of requiring quadratic time.**

**Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.**

**Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.**

**The steps are:**

- **Pick an element, called a pivot, from the list.**
- **Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.**
- **Recursively sort the list of lesser elements and the list of greater elements in sequence.**

**The base cases of the recursion are lists of size zero or one, which are always sorted.**

### Program –

```
#include <iostream>

using namespace std;

void quick_sort(int[],int,int);
int partition(int[],int,int);
int main()
{
   int a[50],n,i;
   cout<<"How many elements?";
  cin>>n;
   cout<<"\nEnter array elements:";

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

```
        cin>>a[i];

    quick_sort(a,0,n-1);
    cout<<"\nArray after sorting:";

    for(i=0;i<n;i++)
        cout<<a[i]<<" ";

    return 0;
}
void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
int partition(int a[],int l,int u)
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;

    do
    {
        do
            i++;

        while(a[i]<v&&i<=u);

        do
            j--;
        while(v<a[j]);

        if(i<j)
        {


  temp=a[i];
            a[i]=a[j];
```

```
            a[j]=temp;
          }
      }while(i<j);

      a[l]=a[j];
      a[j]=v;

      return(j);
}
```

## Output

*How many elements?6*

*Enter array elements:9 15 6 7 10 12*

*Array after sorting:6 7 9 10 12 15*

## 2. Write a program to implement Merge sort algorithm for sorting a list of integers in ascending order.

**Theory:- Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.**
**MergeSort(arr[], l, r)**
**If r > l**

1. **Find the middle point to divide the array into two halves:**
   **middle m = (l+r)/2**
2. **Call mergeSort for first half:**
   **Call mergeSort(arr, l, m)**
3. **Call mergeSort for second half:**
   **Call mergeSort(arr, m+1, r)**
4. **Merge the two halves sorted in step 2 and 3:**
   **Call merge(arr, l, m, r)**

**Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.**
**T(n) = 2T(n/2) +**
**The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .**
**Time complexity of Merge Sort is in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.**
**Auxiliary Space: O(n)**
**Algorithmic Paradigm: Divide and Conquer**
**Sorting In Place: No in a typical implementation**
**Stable: Yes**

### Program:-

```
#include <iostream>

using namespace std;

// A function to merge the two half into a sorted data.
```

```
void Merge(int *a, int low, int high, int mid)
{
        // We have low to mid and mid+1 to high already sorted.
        int i, j, k, temp[high-low+1];
        i = low;
        k = 0;
        j = mid + 1;

        // Merge the two parts into temp[].
        while (i <= mid && j <= high)
        {
                if (a[i] < a[j])
                {
                        temp[k] = a[i];
                        k++;
                        i++;
                }
                else
                {
                        temp[k] = a[j];
                        k++;
                        j++;
                }
        }

        // Insert all the remaining values from i to mid into temp[].
        while (i <= mid)
        {
                temp[k] = a[i];
                k++;
                i++;
        }

        // Insert all the remaining values from j to high into temp[].
        while (j <= high)
        {

                temp[k] = a[j];
                k++;
                j++;
```

```
        }
        // Assign sorted data stored in temp[] to a[].
        for (i = low; i <= high; i++)
        {
                a[i] = temp[i-low];
        }
}
 // A function to split array into two parts.
void MergeSort(int *a, int low, int high)
{
        int mid;
        if (low < high)
        {
                mid=(low+high)/2;
                // Split the data into two half.
                MergeSort(a, low, mid);
                MergeSort(a, mid+1, high);

                // Merge them to get sorted output.
                Merge(a, low, high, mid);
        }
}

int main()
{
        int n, i;
        cout<<"\nEnter the number of data element to be sorted: ";
        cin>>n;

        int arr[n];
        for(i = 0; i < n; i++)
        {
                cout<<"Enter element "<<i+1<<": ";
                cin>>arr[i];
        }

        MergeSort(arr, 0, n-1);

        // Printing the sorted data.
        cout<<"\nSorted Data ";
        for (i = 0; i < n; i++)
```

```
        cout<<"->"<<arr[i];

            return 0;
}
```

## Output:

```
Enter the number of data element to be sorted: 10
Enter element 1: 23
Enter element 2: 987
Enter element 3: 45
Enter element 4: 65
Enter element 5: 32
Enter element 6: 9
Enter element 7: 475
Enter element 8: 1
Enter element 9: 17
Enter element 10: 3

Sorted Data ->1->3->9->17->23->32->45->65->475->987
```

# 3. Write a program to implement the dfs algorithm for a graph.

## Depth First Search Algorithm

**A standard DFS implementation puts each vertex of the graph into one of two categories:**

1. **Visited**
2. **Not Visited**

**The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.**

**The DFS algorithm works as follows:**

1. **Start by putting any one of the graph's vertices on top of a stack.**
2. **Take the top item of the stack and add it to the visited list.**
3. **Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.**
4. **Keep repeating steps 2 and 3 until the stack is empty.**

## Complexity Analysis:

**Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.**

**Space Complexity: O(V).**

**Since an extra visited array is needed of size V.**

## Program:

```
#include <iostream>

#include <list>

using namespace std;

class Graph {

 int numVertices;

 list<int> *adjLists;

 bool *visited;

  public:

 Graph(int V);
```

```
  void addEdge(int src, int dest);
  void DFS(int vertex);
};// Initialize graph
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
  visited = new bool[vertices];
}// Add edges
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_front(dest);}// DFS algorithm
void Graph::DFS(int vertex) {
  visited[vertex] = true;
  list<int> adjList = adjLists[vertex];
  cout << vertex << " ";
  list<int>::iterator i;
  for (i = adjList.begin(); i != adjList.end(); ++i)
    if (!visited[*i])
      DFS(*i);
}int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);
  g.DFS(2);
  return 0;
}
```

## 4. Write a program to implement the bfs algorithm for a graph.

**A standard BFS implementation puts each vertex of the graph into one of two categories:**

- **Visited**
- **Not Visited**

**The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.**

**The algorithm works as follows:**

- **Start by putting any one of the graph's vertices at the back of a queue.**
- **Take the front item of the queue and add it to the visited list.**
- **Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.**
- **Keep repeating steps 2 and 3 until the queue is empty.**

### BFS Algorithm Complexity

**The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.**

**The space complexity of the algorithm is O(V).**

### BFS Algorithm Applications

- **To build index by search index**
- **For GPS navigation**
- **Path finding algorithms**
- **In Ford-Fulkerson algorithm to find maximum flow in a network**
- **Cycle detection in an undirected graph**
- **In minimum spanning tree**

### Program:

```
// BFS algorithm in C++

#include <iostream>

#include <list>

using namespace std;

class Graph {

  int numVertices;

  list<int>* adjLists;

  bool* visited;

   public:

  Graph(int vertices);

  void addEdge(int src, int dest);

  void BFS(int startVertex);

};// Create a graph with given vertices,

// and maintain an adjacency list

Graph::Graph(int vertices) {

  numVertices = vertices;

  adjLists = new list<int>[vertices];

}// Add edges to the graph

void Graph::addEdge(int src, int dest) {

  adjLists[src].push_back(dest);

  adjLists[dest].push_back(src);

}
```

```
// BFS algorithm
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;
  list<int> queue;
  visited[startVertex] = true;
  queue.push_back(startVertex);
  list<int>::iterator i;
  while (!queue.empty()) {
    int currVertex = queue.front();
    cout << "Visited " << currVertex << " ";
    queue.pop_front();
    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}
```

```
int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);
  g.BFS(2);
  return 0;
}
```

## 5. Write a program to implement backtracking algorithm for the N-queens problem.

**THEORY – We have discussed Knight's tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.**

**The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.**

**The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.**

**{ 0, 1, 0, 0}**

**{ 0, 0, 0, 1}**

**{ 1, 0, 0, 0}**

**{ 0, 0, 1, 0}**

**The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.**

**1) Start in the leftmost column**

**2) If all queens are placed**

**return true**

**3) Try all rows in the current column. Do following for every tried row.**

**a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.**

**b) If placing the queen in [row, column] leads to a solution then return true.**

**c) If placing queen doesn't lead to a solution then umark this [row,column] (Backtrack) and go to step (a) to try other rows.**

**3) If all rows have been tried and nothing worked, return false to trigger backtracking.**

**PROGRAM**

```
/* C/C++ program to solve N Queen Problem using backtracking */

#define N 4

#include<stdio.h>

/* A utility function to print solution */

void printSolution(int board[N][N])

{

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

   {

      for (int j = 0; j < N; j++)

         printf(" %d ", board[i][j]);

      printf("n");

   }

}

 /* A utility function to check if a queen can be placed on board[row][col]. Note that this function is called when "col" queens are already placed in columns from 0 to col -1.

  So we need to check only left side for attacking queens */

bool isSafe(int board[N][N], int row, int col)
```

```
{
   int i, j;
   /* Check this row on 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;
}
 /* A recursive utility function to solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
   /* base case: If all queens are placed  then 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 queen from
board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }
     /* If the queen can not be placed in any row in this colum col  then return false */
    return false;
}
```

```
/* This function solves the N Queen problem usingBacktracking. It mainly uses solveNQUtil() to solve the problem. It returns false if queens cannot be placed, otherwise, return true and prints placement of queens in the form of 1s. Please note that there may be more than one solutions, this function prints one of the feasible solutions.*/

bool solveNQ()

{

   int board[N][N] = { {0, 0, 0, 0},

      {0, 0, 0, 0},

      {0, 0, 0, 0},

      {0, 0, 0, 0}

   };

    if ( solveNQUtil(board, 0) == false )

   {printf("Solution does not exist");

     return false;

   } printSolution(board);

   return true;}

 // driver program to test above function

int main(){

   solveNQ();

   return 0;}
```

## 6. Write a program to implement the backtracking algorithm for the sum of subsets problem.

**PROGRAM LOGIC:**

**Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.**

**SOURCE CODE:**

```
#include<stdio.h>

#define TRUE 1

#define FALSE 0

int inc[50],w[50],sum,n;

voidsumset(int ,int ,int);

int promising(inti,intwt,int total) {

return (((wt+total)>=sum)&&((wt==sum)||(wt+w[i+1]<=sum)));

}

void main() {

inti,j,n,temp,total=0;

printf("\n Enter how many numbers: ");

scanf("%d",&n);

printf("\n Enter %d numbers : ",n);

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

scanf("%d",&w[i]);

total+=w[i];
```

```
}
printf("\n Input the sum value to create sub set: ");
scanf("%d",&sum);
for (i=0;i<=n;i++)
for (j=0;j<n-1;j++)
if(w[j]>w[j+1]) {
temp=w[j];
w[j]=w[j+1];
w[j+1]=temp;
}
printf("\n The given %d numbers in ascending order: ",n);
for (i=0;i<n;i++)
printf("%3d",w[i]);
if((total<sum))
printf("\n Subset construction is not possible");
else{
for (i=0;i<n;i++)
inc[i]=0;
printf("\n The solution using backtracking is:\n");
sumset(-1,0,total);
}
}
```

```
voidsumset(inti,intwt,int total){
int j;
if(promising(i,wt,total)) {
if(wt==sum){
printf("\n{");
for (j=0;j<=i;j++)
if(inc[j])
printf("%3d",w[j]);
printf(" }\n");
} else {
inc[i+1]=TRUE;
sumset(i+1,wt+w[i+1],total-w[i+1]);
inc[i+1]=FALSE;
sumset(i+1,wt,total-w[i+1]);
 }
}
}
```

## 7. Write a program to implement the backtracking algorithm for the Hamiltonian Circuits problem.

**Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.**

### Backtracking Algorithm

**Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.**

```
#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

void printSolution(int path[]);

/* A utility function to check if

the vertex v can be added at index 'pos'

in the Hamiltonian Cycle constructed

so far (stored in 'path[]') */

bool isSafe(int v, bool graph[V][V],

        int path[], int pos)
```

```
{
    /* Check if this vertex is an adjacent
    vertex of the previously added vertex. */
    if (graph [path[pos - 1]][ v ] == 0)
        return false;

    /* Check if the vertex has already been included.
    This step can be optimized by creating
    an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function
to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V],
                  int path[], int pos)
{
    /* base case: If all vertices are
```

```
included in Hamiltonian Cycle */
if (pos == V)
{
   // And if there is an edge from the
   // last included vertex to the first vertex
   if (graph[path[pos - 1]][path[0]] == 1)
      return true;
   else
      return false;
}

// Try different vertices as a next candidate
// in Hamiltonian Cycle. We don't try for 0 as
// we included 0 as starting point in hamCycle()
for (int v = 1; v < V; v++)
{
   /* Check if this vertex can be added
   // to Hamiltonian Cycle */
   if (isSafe(v, graph, path, pos))
   {
      path[pos] = v;
```

```
            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos + 1) == true)
               return true;

            /* If adding vertex v doesn't lead to a solution,
            then remove it */
            path[pos] = -1;
         }
      }

      /* If no vertex can be added to
      Hamiltonian Cycle constructed so far,
      then return false */
      return false;
}

/* This function solves the Hamiltonian Cycle problem
using Backtracking. It mainly uses hamCycleUtil() to
solve the problem. It returns false if there is no
Hamiltonian Cycle possible, otherwise return true
and prints the path. Please note that there may be
more than one solutions, this function prints one
```

```
of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
   int *path = new int[V];
   for (int i = 0; i < V; i++)
      path[i] = -1;

   /* Let us put vertex 0 as the first vertex in the path.
   If there is a Hamiltonian Cycle, then the path can be
   started from any point of the cycle as the graph is undirected */
   path[0] = 0;
   if (hamCycleUtil(graph, path, 1) == false )
   {
      cout << "\nSolution does not exist";
      return false;
   }

   printSolution(path);
   return true;
}
/* A utility function to print solution */
void printSolution(int path[])
```

```
{   cout << "Solution Exists:"
          " Following is one Hamiltonian Cycle \n";
   for (int i = 0; i < V; i++)
      cout << path[i] << " ";
  // Let us print the first vertex again
   // to show the complete cycle
   cout << path[0] << " ";
   cout << endl;
}// Driver Code
int main()
{  /* Let us create the following graph
      (0)--(1)--(2)
      | / \ |
      | / \ |
      | / \ |
      (3)-------(4) */
   bool graph1[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0}};
      // Print the solution
```

```
  hamCycle(graph1);

    /* Let us create the following graph

  (0)--(1)--(2)

  | / \ |

  | / \ |

  | / \ |

  (3) (4) */

  bool graph2[V][V] = {{0, 1, 0, 1, 0},

                {1, 0, 1, 1, 1},

                {0, 1, 0, 0, 1},

                {1, 1, 0, 0, 0},

                {0, 1, 1, 0, 0}};

 // Print the solution

  hamCycle(graph2);

 return 0;}
```

## Output:

```
Solution Exists: Following is one Hamiltonian Cycle

 0   1  2  4  3  0
```

## 8. Write a program to implement Knapsack Problem.

**In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.**

**Input :**

**Same as above**

**Output :**

**Maximum possible value = 240**

**By taking full items of 10 kg, 20 kg and**

**2/3rd of last item of 30 kg**

**An efficient solution is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.**
**A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.**
**After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.**

**As main time taking step is sorting, the whole problem can be solved in O(n log n) only.**

***PROGRAM***

```
// C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>

using namespace std;

 // Structure for an item which stores weight and corresponding

// value of Item

struct Item

{

   int value, weight;
```

```
    // Constructor
    Item(int value, int weight) : value(value), weight(weight)
    {}
};
 // Comparison function to sort Item according to val/weight ratio
bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}
double fractionalKnapsack(int W, struct Item arr[], int n)
{
        sort(arr, arr + n, cmp);
     //    Uncomment to see new order of Items with their ratio    /*
    for (int i = 0; i < n; i++)
    {
        cout << arr[i].value << "  " << arr[i].weight << " : "
             << ((double)arr[i].value / arr[i].weight) << endl;
    }
    */
     int curWeight = 0;  // Current weight in knapsack
    double finalvalue = 0.0; // Result (value in Knapsack)
     for (int i = 0; i < n; i++)
    {
         if (curWeight + arr[i].weight <= W)
        {
```

```
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }
        // If we can't add current Item, add fractional part of it
        else
        {
            int remain = W - curWeight;
            finalvalue += arr[i].value * ((double) remain / arr[i].weight);
            break;
        }
    }
        return finalvalue;
}
// driver program to test above function
int main()
{
    int W = 50;   //    Weight of knapsack
    Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum value we can obtain = "
        << fractionalKnapsack(W, arr, n);
    return 0;
}
```

## 9. Write a program to implement Dijkstra's algorithm for the Single source shortest path problem.

### Dijkstra's algorithm.

**1) Create a set S that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.**

**2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE.Assign distance value as 0 for the source vertex so that it is picked first.**

**3) While S doesn't include all vertices**

**a) Pick a vertex u which is not there in S and has minimum distance value.**

**b)Include u to S.**

**c) Update distance value of all adjacent vertices of u.**

**To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if**

**sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v,**

**then update the distance value of v.**

### SOURCE CODE:

```
#include<stdio.h>

#define infinity 999

void dij(int n, int v,int cost[20][20], int dist[]){
```



```
int i,u,count,w,flag[20],min;
```

```
for(i=1;i<=n;i++)
flag[i]=0, dist[i]=cost[v][i];
count=2;
while(count<=n){
min=99;
for(w=1;w<=n;w++)
if(dist[w]<min && !flag[w]) {
min=dist[w];
u=w;
}
flag[u]=1;
count++;
for(w=1;w<=n;w++)
if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
dist[w]=dist[u]+cost[u][w];
}
}
int main(){
int n,v,i,j,cost[20][20],dist[20];
printf("enter the number of nodes:");
scanf("%d",&n);
printf("\n enter the cost 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]=infinity;

}

printf("\n enter the source matrix:");

scanf("%d",&v);

dij(n,v,cost,dist);

printf("\n shortest path : \n");

for(i=1;i<=n;i++)

if(i!=v)

printf("%d->%d,cost=%d\n",v,i,dist[i]);

}
```

## 10. Write a program that implements Prim's algorithm to generate minimum cost spanning tree.

**PROGRAM LOGIC:**

**1) Create a set S that keeps track of vertices already included in MST.**

**2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE.**

**Assign key value as 0 for the first vertex so that it is picked first.**

**3) While S doesn't include all vertices.**

**a) Pick a vertex u which is not there in Sand has minimum key value.**

**b) Include u to S.**

**c) Update key value of all adjacent vertices of u.**

**To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if**

**weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v**

**The idea of using key values is to pick the minimum weight edge from cut. The key values are used**

**only for vertices which are not yet included in MST, the key value for these vertices indicate the**

**minimum weight edges connecting them to the set of vertices included in MST.**

```
#include<stdio.h>

inta,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()
```

```
{
printf("\n Enter the number of
nodes:"); scanf("%d",&n);
printf("\n Enter the 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;
}
visited[1]=1;
printf("\n");
while(ne<n)
{
for(i=1,min=999;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]<min)
if(visited[i]!=0)
{
min=cost[i][j];
a=u=i;
b=v=j;
```

```
}
if(visited[u]==0 || visited[v]==0)
{
printf("\n Edge %d:(%d %d)
cost:%d",ne++,a,b,min); mincost+=min;
visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n Minimun cost=%d",mincost);
}
```