**PRACTICAL NO.1**

**AIM:** Write program for Recursive Binary Search .Write worst and best case time complexity.

**THEORY:**

PROGRAM:

int binarySearch(int arr[], int l, int r, int x)

{

   if (r >= l)

   {

        int mid = l + (r - l)/2;

        // If the element is present at the middle itself

        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present

        // in left subarray

        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray

        return binarySearch(arr, mid+1, r, x);

   }

   // We reach here when element is not present in array

   return -1;

}

int main(void)

{

   int arr[] = {2, 3, 4, 10, 40};

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

   int x = 10;

   int result = binarySearch(arr, 0, n-1, x);

   (result == -1)? printf("Element is not present in array")

                 : printf("Element is present at index %d", result);

   return 0;

}

**OUTPUT**:

**PRACTICAL NO.2**

**AIM:** Write program for Non-Recursive Binary Search. Explain best, average and worst case time complexity for same.

**THEORY:**

PROGRAM:

#include<stdio.h>

int main()

{

    int arr[50],i,n,x,flag=0,first,last,mid;

    printf("Enter size of array:");

    scanf("%d",&n);

    printf("\n Enter array element(ascending order)\n");

    for(i=0;i<n;++i)

        scanf("%d",&arr[i]);

    printf("\nEnter the element to search:");

    scanf("%d",&x);

    first=0;

    last=n-1;

    while(first<=last)

    {

        mid=(first+last)/2;

        if(x==arr[mid]){

            flag=1;

            break;

        }

        else

            if(x>arr[mid])

                first=mid+1;

            else

                last=mid-1;

    }

    if(flag==1)

        printf("\nElement found at position %d",mid+1);

    else

        printf("\nElement not found");

    return 0;

}

**OUTPUT**

**PRACTICAL NO.3**

**AIM:** Write a program for Quick Sort using Divide & Conquer approach take inputs as 9,7,8,10,12,56,38,4,2,88,11. Analyze the average and worst case time complexity and space complexity for same.

**THEORY:**

**Quick sort:**

**PROGRAM:**

#include <stdio.h>

void quick\_sort(int[],int,int);

int partition(int[],int,int);

 int main()

{

int a[50],n,i;

printf("How many elements?");

scanf("%d",&n);

printf("\nEnter array elements:");

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

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

quick\_sort(a,0,n-1);

printf("\nArray after sorting:");

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

printf("%d ",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**

**PRACTICAL 4**

**AIM:** Write a program for Merge Sort using Divide & Conquer approach take inputs as 12, 9,7,8,5,3,2,1. Analyze the worst case time complexity and space complexity for same.

THEORY:

**PROGRAM:**

#include<stdio.h>

void mergesort(int a[],int i,int j);

void merge(int a[],int i1,int j1,int i2,int j2);

int main()

{

int a[30],n,i;

printf("Enter no of elements:");

scanf("%d",&n);

printf("Enter array elements:");

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

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

mergesort(a,0,n-1);

printf("\nSorted array is :");

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

printf("%d ",a[i]);

return 0;

}

void mergesort(int a[],int i,int j)

{

int mid;

if(i<j)

{

mid=(i+j)/2;

mergesort(a,i,mid); //left recursion

mergesort(a,mid+1,j); //right recursion

merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays

}

}

void merge(int a[],int i1,int j1,int i2,int j2)

{

int temp[50]; //array used for merging

int i,j,k;

i=i1; //beginning of the first list

j=i2; //beginning of the second list

k=0;

while(i<=j1 && j<=j2) //while elements in both lists

{

if(a[i]<a[j])

temp[k++]=a[i++];

else

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

}

while(i<=j1) //copy remaining elements of the first list

temp[k++]=a[i++];

while(j<=j2) //copy remaining elements of the second list

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

//Transfer elements from temp[] back to a[]

for(i=i1,j=0;i<=j2;i++,j++)

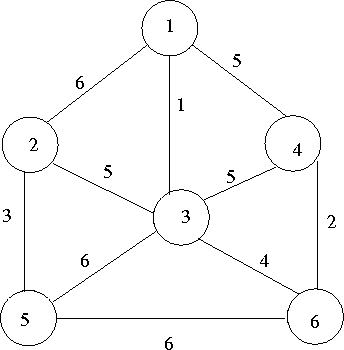
a[i]=temp[j];

}

**OUTPUT**

**PRACTICAL 5**

**AIM:** Write a program for finding minimum spanning tree using prim’s algorithm and calculate the time complexity for same.



**THEORY:**

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#define infinity 9999

#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()

{

int i,j,total\_cost;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

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

total\_cost=prims();

printf("\nspanning tree matrix:\n");

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

{

printf("\n");

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

printf("%d\t",spanning[i][j]);

}

printf("\n\nTotal cost of spanning tree=%d",total\_cost);

return 0;

}

int prims()

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

//create cost[][] matrix,spanning[][]

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

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

{

if(G[i][j]==0)

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning[i][j]=0;

}

//initialise visited[],distance[] and from[]

distance[0]=0;

visited[0]=1;

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

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0; //cost of spanning tree

no\_of\_edges=n-1; //no. of edges to be added

while(no\_of\_edges>0)

{

//find the vertex at minimum distance from the tree

min\_distance=infinity;

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

if(visited[i]==0&&distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

//insert the edge in spanning tree

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

//updated the distance[] array

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

if(visited[i]==0&&cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

from[i]=v;

}

min\_cost=min\_cost+cost[u][v];

}

return(min\_cost);

}

**OUTPUT**

**PRACTICAL 6**

**AIM:** Write a program for finding minimum spanning tree using Kruskal’s Algorithm and calculate the time complexity for same.

**PROGRAM:**

#include<stdio.h>

#define MAX 30

typedef struct edge

{

int u,v,w;

}edge;

typedef struct edgelist

{

edge data[MAX];

int n;

}edgelist;

edgelist elist;

int G[MAX][MAX],n;

edgelist spanlist;

void kruskal();

int find(int belongs[],int vertexno);

void union1(int belongs[],int c1,int c2);

void sort();

void print();

void main()

{

int i,j,total\_cost;

printf("\nEnter number of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

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

kruskal();

print();

}

void kruskal()

{

int belongs[MAX],i,j,cno1,cno2;

elist.n=0;

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

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

{

if(G[i][j]!=0)

{

elist.data[elist.n].u=i;

elist.data[elist.n].v=j;

elist.data[elist.n].w=G[i][j];

elist.n++;

}

}

sort();

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

belongs[i]=i;

spanlist.n=0;

for(i=0;i<elist.n;i++)

{

cno1=find(belongs,elist.data[i].u);

cno2=find(belongs,elist.data[i].v);

if(cno1!=cno2)

{

spanlist.data[spanlist.n]=elist.data[i];

spanlist.n=spanlist.n+1;

union1(belongs,cno1,cno2);

}

}

}

int find(int belongs[],int vertexno)

{

return(belongs[vertexno]);

}

void union1(int belongs[],int c1,int c2)

{

int i;

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

if(belongs[i]==c2)

belongs[i]=c1;

}

void sort()

{

int i,j;

edge temp;

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

for(j=0;j<elist.n-1;j++)

if(elist.data[j].w>elist.data[j+1].w)

{

temp=elist.data[j];

elist.data[j]=elist.data[j+1];

elist.data[j+1]=temp;

}

}

void print()

{

int i,cost=0;

for(i=0;i<spanlist.n;i++)

{

printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);

cost=cost+spanlist.data[i].w;

}

printf("\n\nCost of the spanning tree=%d",cost);

}

**OUTPUT**

PRACTICAL 7

**AIM:** Write a program for finding all pair shortest path for multi-stage graph and write time complexity for same algorithm.

**THEORY:**

**PROGRAM:**

#include<stdio.h>

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough value. This value will be used

  for vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])

{

    /\* dist[][] will be the output matrix that will finally have the shortest

      distances between every pair of vertices \*/

    int dist[V][V], i, j, k;

    /\* Initialize the solution matrix same as input graph matrix. Or

       we can say the initial values of shortest distances are based

       on shortest paths considering no intermediate vertex. \*/

    for (i = 0; i < V; i++)

        for (j = 0; j < V; j++)

            dist[i][j] = graph[i][j];

    /\* Add all vertices one by one to the set of intermediate vertices.

      ---> Before start of an iteration, we have shortest distances between all

      pairs of vertices such that the shortest distances consider only the

      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.

      ----> After the end of an iteration, vertex no. k is added to the set of

      intermediate vertices and the set becomes {0, 1, 2, .. k} \*/

    for (k = 0; k < V; k++)

    {

        // Pick all vertices as source one by one

        for (i = 0; i < V; i++)

        {

            // Pick all vertices as destination for the

            // above picked source

            for (j = 0; j < V; j++)

            {

                // If vertex k is on the shortest path from

                // i to j, then update the value of dist[i][j]

                if (dist[i][k] + dist[k][j] < dist[i][j])

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

            }

        }

    }

    // Print the shortest distance matrix

    printSolution(dist);

}

/\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

    printf ("The following matrix shows the shortest distances"

            " between every pair of vertices \n");

    for (int i = 0; i < V; i++)

    {

        for (int j = 0; j < V; j++)

        {

            if (dist[i][j] == INF)

                printf("%7s", "INF");

            else

                printf ("%7d", dist[i][j]);

        }

        printf("\n");

    }

}

// driver program to test above function

int main()

{

    /\* Let us create the following weighted graph

            10

       (0)------->(3)

        |         /|\

      5 |          |

        |          | 1

       \|/         |

       (1)------->(2)

            3           \*/

    int graph[V][V] = { {0,   5,  INF, 10},

                        {INF, 0,   3, INF},

                        {INF, INF, 0,   1},

                        {INF, INF, INF, 0}

                      };

    // Print the solution

    floydWarshall(graph);

    return 0;

}

**OUTPUT**

**PRACTICAL 8**

**AIM:** Write a program for finding single pair shortest path for multi-stage graph.

**THEORY:**

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#define INFINITY 9999

#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()

{

int G[MAX][MAX],i,j,n,u;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

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

printf("\nEnter the starting node:");

scanf("%d",&u);

dijkstra(G,n,u);

return 0;

}

void dijkstra(int G[MAX][MAX],int n,int startnode)

{

int cost[MAX][MAX],distance[MAX],pred[MAX];

int visited[MAX],count,mindistance,nextnode,i,j;

//pred[] stores the predecessor of each node

//count gives the number of nodes seen so far

//create the cost matrix

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

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

if(G[i][j]==0)

cost[i][j]=INFINITY;

else

cost[i][j]=G[i][j];

//initialize pred[],distance[] and visited[]

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

{

distance[i]=cost[startnode][i];

pred[i]=startnode;

visited[i]=0;

}

distance[startnode]=0;

visited[startnode]=1;

count=1;

while(count<n-1)

{

mindistance=INFINITY;

//nextnode gives the node at minimum distance

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

if(distance[i]<mindistance&&!visited[i])

{

mindistance=distance[i];

nextnode=i;

}

//check if a better path exists through nextnode

visited[nextnode]=1;

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

if(!visited[i])

if(mindistance+cost[nextnode][i]<distance[i])

{

distance[i]=mindistance+cost[nextnode][i];

pred[i]=nextnode;

}

count++;

}

//print the path and distance of each node

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

if(i!=startnode)

{

printf("\nDistance of node%d=%d",i,distance[i]);

printf("\nPath=%d",i);

j=i;

do

{

j=pred[j];

printf("<-%d",j);

}while(j!=startnode);

}

}

**OUTPUT**

**PRACTICAL 9**

**AIM:** Write a program for knapsack algorithm for maximizing profit where n=4, maximum weight capacity=10 and analyze the time complexity.

**THEORY:**

**PROGRAM:**

#include<stdio.h>

int max(int a, int b) { return (a > b)? a : b; }

int knapSack(int W, int wt[], int val[], int n)

{

   int i, w;

   int K[n+1][W+1];

   for (i = 0; i <= n; i++)

   {

       for (w = 0; w <= W; w++)

       {

           if (i==0 || w==0)

               K[i][w] = 0;

           else if (wt[i-1] <= w)

                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);

           else

                 K[i][w] = K[i-1][w];

       }

   }

return K[n][W];

}

int main()

{

    int i, n, val[20], wt[20], W;

    printf("Enter number of items:");

    scanf("%d", &n);

    printf("Enter value and weight of items:\n");

    for(i = 0;i < n; ++i){

     scanf("%d%d", &val[i], &wt[i]);

    }

    printf("Enter size of knapsack:");

    scanf("%d", &W);

    printf("%d", knapSack(W, wt, val, n));

    return 0;

}

**OUTPUT**

**PRACTICAL 10**

**AIM:** Write a program for solving 8-Queens problem**.**

**THEORY:**

**PROGRAM:**

#include<stdio.h>

#include<math.h>

int board[20],count;

int main()

{

int n,i,j;

void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");

printf("\n\nEnter number of Queens:");

scanf("%d",&n);

queen(1,n);

return 0;

}

//function for printing the solution

void print(int n)

{

int i,j;

printf("\n\nSolution %d:\n\n",++count);

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

  printf("\t%d",i);

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

{

  printf("\n\n%d",i);

  for(j=1;j<=n;++j) //for nxn board

  {

   if(board[i]==j)

    printf("\tQ"); //queen at i,j position

   else

    printf("\t-"); //empty slot

  }

}

}

/\*funtion to check conflicts

If no conflict for desired postion returns 1 otherwise returns 0\*/

int place(int row,int column)

{

int i;

for(i=1;i<=row-1;++i)

{

  //checking column and digonal conflicts

  if(board[i]==column)

   return 0;

  else

   if(abs(board[i]-column)==abs(i-row))

    return 0;

}

return 1; //no conflicts

}

//function to check for proper positioning of queen

void queen(int row,int n)

{

int column;

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

{

  if(place(row,column))

  {

   board[row]=column; //no conflicts so place queen

   if(row==n) //dead end

    print(n); //printing the board configuration

   else //try queen with next position

    queen(row+1,n);

  }

}

}

**OUTPUT**

**PRACTICAL 11**

**AIM:** Write a program for getting an optimal binary search tree.

**Input: keys[] = {10, 12,19,17}, freq[] = {7, 5,13,5}**

**THEORY:**

**PROGRAM:**

#include<stdio.h>  
#include<conio.h>  
#define MAX 10  
  
void main()  
{  
  
 char ele[MAX][MAX];  
 int w[MAX][MAX], c[MAX][MAX], r[MAX][MAX], p[MAX], q[MAX];  
 int temp=0, root, min, min1, n;  
 int i,j,k,b;   
 clrscr();  
 printf("Enter the number of elements:");  
 scanf("%d",&n);  
 printf("\n");  
 for(i=1; i <= n; i++)  
 {  
 printf("Enter the Element of %d:",i);  
 scanf("%d",&p[i]);  
 }  
  
        printf("\n");  
 for(i=0; i <= n; i++)  
 {  
 printf("Enter the Probability of %d:",i);  
 scanf("%d",&q[i]);  
 }  
 printf("W\t\tC\t\tR\n");   
 for(i=0; i <= n; i++)  
 {  
 for(j=0; j <= n; j++)  
 {  
 if(i == j)  
 {  
 w[i][j] = q[i];  
 c[i][j] = 0;  
 r[i][j] = 0;  
 printf("W[%d][%d]: %d\tC[%d][%d]: %d\tR[%d][%d]: %d\n",i,j,w[i][j],i,j,c[i][j],i,j,r[i][j]);  
 }  
 }  
  
}  
 printf("\n");  
 for(b=0; b < n; b++)  
 {  
 for(i=0,j=b+1; j < n+1 && i < n+1; j++,i++)  
 {  
  
  
 if(i!=j && i < j)  
 {  
 w[i][j] = p[j] + q[j] + w[i][j-1];  
 min = 30000;  
 for(k = i+1; k <= j; k++)  
 {  
 min1 = c[i][k-1] + c[k][j] + w[i][j];  
 if(min > min1)  
 {  
 min = min1;  
 temp = k;  
 }  
 }  
 c[i][j] = min;  
 r[i][j] = temp;  
 }  
 printf("W[%d][%d]: %d\tC[%d][%d]: %d\tR[%d][%d]: %d\n",i,j,w[i][j],i,j,c[i][j],i,j,r[i][j]);  
  
                }  
 printf("\n");  
 }  
  
  
 printf("Minimum cost = %d\n",c[0][n]);  
 root = r[0][n];  
  
 printf("Root  = %d \n",root);  
 getch();  
}

**OUTPUT**

**PROGRAM 12**

**AIM:** Write a program of Graph Coloring for given data sets.

**THEORY:**

**PROGRAM:**

#include<stdio.h>  
#include<conio.h>  
static int m, n;  
static int c=0;  
static int count=0;  
int g[50][50];  
int x[50];  
void nextValue(int k);  
void GraphColoring(int k);  
void main()  
{  
 int i, j;  
 int temp;  
 clrscr();  
 printf("\nEnter the number of nodes: " );  
 scanf("%d", &n);

/\* printf("\nIf edge exists then enter 1 else enter 0 \n");  
for(i=1; i<=n; i++)  
 {  
  x[i]=0;  
  for(j=1; j<=n; j++)  
  {  
   if(i==j)  
    g[i][j]=0;  
   else

   {

    printf("%d -> %d: " , i, j);

    scanf("%d", &temp);

    g[i][j]=g[j][i]=temp;  
}  
  }  
 }

\*/  
 printf("\nEnter Adjacency Matrix:\n");  
 for(i=1;i<=n;i++)  
 {  
  for(j=1;j<=n;j++)  
  {  
   scanf("%d", &g[i][j]);  
  }

 }  
printf("\nPossible Solutions are\n");  
 for(m=1;m<=n;m++)  
 {  
  if(c==1)  
  {  
   break;  
  }  
  GraphColoring(1);

 }

 printf("\nThe chromatic number is %d", m-1);  
 //in for loop, m gets incremented first and then the condition is checked  
 //so it is m minus 1  
 printf("\nThe total number of solutions is %d", count);

 getch();  
}

void GraphColoring(int k)  
{  
 int i;  
 while(1)  
 {  
  nextValue(k);  
  if(x[k]==0)  
  {  
   return;  
  }

 if(k==n)  
 {  
  c=1;  
  for(i=1;i<=n;i++)  
  {  
   printf("%d ", x[i]);  
  }  
  count++;  
  printf("\n");  
 }  
 else  
  GraphColoring(k+1);  
 }  
}

void nextValue(int k)

{  
int j;  
 while(1)  
 {  
  x[k]=(x[k]+1)%(m+1);  
  if(x[k]==0)  
  {  
   return;  
  }  
  for(j=1;j<=n;j++)  
  {  
   if(g[k][j]==1&&x[k]==x[j])  
  break;

  }

  if(j==(n+1))  
  {  
   return;  
  }  
 }

}

**OUTPUT**

**PRACTICAL 13**

**AIM:** Write a program for getting Hamilton cycle for the given graph

**THEORY:**

**PROGRAM:**

#include<stdio.h>

// 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 )

    {

        printf("\nSolution does not exist");

        return false;

    }

    printSolution(path);

    return true;

}

/\* A utility function to print solution \*/

void printSolution(int path[])

{

    printf ("Solution Exists:"

            " Following is one Hamiltonian Cycle \n");

    for (int i = 0; i < V; i++)

        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle

    printf(" %d ", path[0]);

    printf("\n");

}

// driver program to test above function

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**

**PRACTICAL 14**

**AIM:** Write a program for finding minimum cost and path for travelling n no of cities using travelling sales person problem.

**THEORY**

**PROGRAM:**

#include<stdio.h>

int ary[10][10],completed[10],n,cost=0;

void takeInput()

{

int i,j;

printf("Enter the number of villages: ");

scanf("%d",&n);

printf("\nEnter the Cost Matrix\n");

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

{

printf("\nEnter Elements of Row: %d\n",i+1);

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

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

completed[i]=0;

}

printf("\n\nThe cost list is:");

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

{

printf("\n");

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

printf("\t%d",ary[i][j]);

}

}

void mincost(int city)

{

int i,ncity;

completed[city]=1;

printf("%d--->",city+1);

ncity=least(city);

if(ncity==999)

{

ncity=0;

printf("%d",ncity+1);

cost+=ary[city][ncity];

return;

}

mincost(ncity);

}

int least(int c)

{

int i,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;

}

int main()

{

takeInput();

printf("\n\nThe Path is:\n");

mincost(0); //passing 0 because starting vertex

printf("\n\nMinimum cost is %d\n ",cost);

return 0;

}

**OUTPUT**