SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY: PUTTUR

(AUTONOMOUS)

(Approved by AICTE & Affiliated to JNTUA, Anantapuramu)

(Accredited by NBA for CIVIL, MECH, ECE, CSE, EEE New Delhi)

(Accredited by NAAC with Grade 'A+', an ISO 9001:2008 Certified Institution)

Siddharth Nagar, Narayanavanam Road, Puttur-517583

Department of Computer Science and Engineering

(WITH SPECIALISATION IN ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

(20CS0525) - Design and Analysis of Algorithms Lab

III B.Tech -II Semester

Lab Observation Book

Academic Year: _____

Name : _____

Roll. Number : _____

Year & Branch:_____

Section :_____

SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)
(Approved by AICTE, New Delhi& Affiliated to JNTUA, Ananthapuramu)
(Accredited by NBA for Civil, EEE, Mech., ECE & CSE
Accredited by NAAC with 'A+' Grade)
Puttur -517583, Tirupati District, A.P. (India)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE VISION

To emerge as one of the premier institutions through excellence in education and research, producing globally competent and ethically strong professionals and entrepreneurs

INSTITUTE MISSION

M1: Imparting high-quality technical and management education through the state-of-the- art resources.

M2: Creating an eco-system to conduct independent and collaborative research for the betterment of the society

M3: Promoting entrepreneurial skills and inculcating ethics for the socio-economic development of the nation.

DEPARTMENT VISION

To impart quality education and research in Computer Science and Engineering for producing technically competent and ethically strong IT professionals with contemporary knowledge

DEPARTMENT MISSION

M1: Achieving academic excellence in computer science through effective pedagogy, modern curriculum and state-of-art computing facilities.

M2: Encouraging innovative research in Computer Science and Engineering by collaborating with Industry and Premier Institutions to serve the nation.

M3: Empowering the students by inculcating professional behavior, strong ethical values and leadership abilities

SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)
(Approved by AICTE, New Delhi& Affiliated to JNTUA, Ananthapuramu)
(Accredited by NBA for Civil, EEE, Mech., ECE & CSE
Accredited by NAAC with 'A+' Grade)
Puttur -517583, Tirupati District, A.P. (India)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Program Outcomes

PO1: Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-Long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)
(Approved by AICTE, New Delhi& Affiliated to JNTUA, Ananthapuramu)
(Accredited by NBA for Civil, EEE, Mech., ECE & CSE
Accredited by NAAC with 'A+' Grade)
Puttur -517583, Tirupati District, A.P. (India)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: To provide software solutions for arising problems in diverse areas with strong knowledge in innovative technologies of computer science.

PEO2: To serve in IT industry as professionals and entrepreneurs or in pursuit of higher education and research.

PEO3: To attain professional etiquette, soft skills, leadership, ethical values meld with a commitment for lifelong learning.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: **Analysis & Design:** Ability to design, develop and deploy customized applications in all applicable domains using various algorithms and programming languages.

PSO2: **Computational Logic:** Ability to visualize and configure computational need in terms of hardware and software to provide solutions for various complex applications.

PSO3: **Software Development:** Ability to apply standard procedures, tools and strategies for software development.

SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY
(AUTONOMOUS)
Narayanavanam Road, PUTTUR-517 583

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Name of the Lab: DESIGN & ANALYSIS OF ALGORITHMS LAB(20CS0525)
Year & SEM : II B.TECH – II Sem.

COURSE OBJECTIVES:

The Objectives of this course:

1. Analyze the asymptotic performance of algorithms.
2. Write rigorous correctness proofs for algorithms.
3. Demonstrate a familiarity with major algorithms and data structures.
4. Apply important algorithmic design paradigms and methods of analysis.
5. Synthesize efficient algorithms in common engineering design situations.

COURSE OUTCOMES (COs):

On successful completion of this course, the student will be able to

1. Able to understand the techniques of proof by contradiction, mathematical induction and recurrence relation, and apply them to prove the correctness and to analyze the running time of algorithms.
2. Design new algorithms, prove them correct, and analyze their asymptotic and absolute runtime and memory demands.
3. Analyse an algorithm to solve the problem (create) and prove that the algorithm solves the problem correctly (validate).
4. Understand the mathematical criterion for deciding whether an algorithm is efficient, and know many practically important problems that do not admit any efficient algorithms.
5. Understand basic techniques for designing algorithms, including the techniques of recursion, divide-and-conquer, and greedy.
6. Analyse NP-Completeness, NP-complete problems and synthesize efficient algorithm in common engineering design situations.

LIST OF EXPERIMENTS:

1. Obtain the Topological ordering of vertices in a given digraph.
2. Sort a given set of elements using the Quick sort method and determine the time required to sort the elements.
3. Sort a given set of elements using the Merge sort method and determine the time required to sort the elements.
4. Check whether a given graph is connected or not using DFS method.
5. Print all the nodes reachable from a given starting node in a directed graph using BFS method
6. Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.
7. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.
8. Implement 0/1 Knapsack problem using Dynamic Programming.
9. Write a program to implement Travelling Sales Person problem using Dynamic programming.
10. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
11. Design and implement the presence of Hamiltonian Cycle in an undirected Graph G of n vertices.

TEXT BOOKS:

1. Fundamentals of Computer Algorithms, Ellis Horowitz, S.SatrajSahni and Rajasekharam, Galgotia Publications Pvt. Ltd., 4th Edition, 1998.
2. Design and Analysis Algorithms-ParagHimanshu Dave, HimanshuBhalchandra Dave, Pearson Education India, 2007.

REFERENCES:

1. AnanyLevitin, "Introduction to the Design and Analysis of Algorithms", Third Edition, Pearson Education, 2012.
2. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Third Edition, PHI Learning Private Limited, 2012.
3. Alfred V. Aho, John E. Hopcroft and Jeffrey Ullman, "Data Structures and Algorithms", Pearson Education.

INDEX

Sl.No.	Date	Name of the Experiment	Page No.	Sign.

Sl.No.	Date	Name of the Experiment	Page No.	Sign.

Ex.No-1	Obtain The Topological Ordering of Vertices in a Given Digraph	Date:

Aim:

To obtain the topological ordering of Vertices in a Given Digraph.

Description:

- Topological sort is an ordering of the vertices in a directed acyclic graph, such that, if there is a path from u to v, then v appears after u in the ordering.
- **The graphs should be directed:** otherwise for any edge (u, v) there would be a path from u to v and also from v to u, and hence they cannot be ordered.
- **The graphs should be acyclic:** otherwise for any two vertices u and v on a cycle u would precede v and v would precede u.

Using Source Removal algorithm:

1. Compute the in degrees of all vertices
2. Find a vertex U with in degree 0 and print it (store it in the ordering)
3. If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
4. Remove U and all its edges (U, V) from the graph.
5. Update the indegrees of the remaining vertices.
6. Repeat steps 2 through 4 while there are vertices to be processed.

Source Code:

```
#include<stdio.h>
int temp[10],k=0;
voidtopo(intn,intindegree[10],int a[10][10])
{
inti,j;
for(i=1;i<=n;i++)
{
if(indegree[i]==0)
{
indegree[i]=1;
temp[++k]=i;
for(j=1;j<=n;j++)
{
if(a[i][j]==1&&indegree[j]!=-1)
indegree[j]--;
}
i=0;
}
}
}
void main()
{
```

```
inti,j,n,indegree[10],a[10][10];
printf("enter the number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
indegree[i]=0;
printf("\n enter the adjacency matrix\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&a[i][j]);
if(a[i][j]==1)
indegree[j]++;
}
topo(n,indegree,a);
if(k!=n)
printf("topological ordering is not possible\n");
else
{
printf("\n topological ordering is :\n");
for(i=1;i<=k;i++)
printf("v%d\t",temp[i]);
}
}
```

Using DFS algorithm:

1. Run DFS(G), computing finish time for each vertex
2. As each vertex is finished, insert it onto the front of a list

Source Code:

```
#include<stdio.h>
inti,visit[20],n,adj[20][20],s,topo_order[10];
voiddfs(int v)
{
int w;
visit[v]=1;
for(w=1;w<=n;w++)
if((adj[v][w]==1) && (visit[w]==0))
dfs(w);
topo_order[i--]=v;
}
voidmain ()
{
intv,w;
printf("Enter the number of vertices:\n");
```

```
scanf("%d",&n);
printf("Enter the adjacency matrix:\n");
for(v=1;v<=n;v++)
for(w=1;w<=n;w++)
scanf("%d",&adj[v][w]);
for(v=1;v<=n;v++)
visit[v]=0;
i=n;
for(v=1;v<=n;v++)
{
if(visit[v]==0)
dfs(v);
}
printf("\nTopological sorting is:");
for(v=1;v<=n;v++)
printf("v%d ",topo_order[v]);
}
```

OUTPUT:

RESULT:

Ex.No-2	Sort a given set of elements using the Quick sort method and determine the time required to sort the elements.	Date:

Aim:

To Sort a given set of elements using the Quick sort method and determine the time required to sort the elements.

Description:

1. Quicksort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:
2. The partition phase and the sort phase.
3. As we will see, most of the work is done in the partition phase. The sort phase simply sorts the two smaller problems that are generated in the partition phase.
4. This makes Quicksort a good example of the divide and conquers strategy for solving problems. In quicksort, we divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions, ie we divide the problem into two smaller ones and conquer by solving the smaller ones.

Pseudo code For partition(a, left, right, pivotIndex):

```
pivotValue := a[pivotIndex]
swap(a[pivotIndex], a[right]) // Move pivot to end
storeIndex := left
for i from left to right-1
if a[i] ≤ pivotValue
swap(a[storeIndex], a[i])
storeIndex := storeIndex + 1
swap(a[right], a[storeIndex]) // Move pivot to its final place
returnstoreIndex
```

Pseudo code For quicksort(a, left, right):

```
if right > left
select a pivot value a[pivotIndex]
pivotNewIndex := partition(a, left, right, pivotIndex)
quicksort(a, left, pivotNewIndex-1)
quicksort (a, pivotNewIndex+1, right)
```

ANALYSIS:

The partition routine examines every item in the array at most once, so complexity is clearly O (n).Usually, the partition routine will divide the problem into two roughly equal sized partitions. We know that we can divide n items in half log2n times.

Source code:

```
#include<stdio.h>
#include<conio.h>
void quicksort(int[],int, int);
intpartition (int[],int,int);
```

```
void main()
{
inti,n,a[20],ch=1;
clrscr();
while(ch)
{
printf("\n enter the number of elements\n");
scanf("%d",&n);
printf("\n enter the array elements\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
quicksort(a,0,n-1);
printf("\n\nthe sorted array elements are\n\n");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
printf("\n\n do u wish to continue (0/1)\n");
scanf("%d",&ch);
}
getch();
}
void quicksort(int a[],intlow,int high)
{
int mid;
if(low<high)
{
mid=partition(a,low,high);
quicksort(a,low,mid-1);
quicksort(a,mid+1,high);
}
}
int partition(int a[],intlow,int high)
{
intkey,i,j,temp,k;
key=a[low];
i=low+1;
j=high;
while(i<=j)
{
while(i<=high && key>=a[i])
i=i+1;
while(key<a[j])
j=j-1;
if(i<j)
{
temp=a[i];
a[i]=a[j];
```

```
a[j]=temp;
}
else
{
k=a[j];
a[j]=a[low];
a[low]=k;
}
}
return j;
}
```

OUTPUT:

RESULT:

Ex.No-3	Sort a given set of elements using the Merge sort method and determine the time required to sort the elements.	Date:

Aim:

To Sort a given set of elements using the Merge sort method and determine the time required to sort the elements.

Objective:

Sort a given set of elements using Merge sort method and determine the time taken to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

Algorithm:

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 merges Sort for first half:
    Call merges Sort (arr, l, m)
3. Call merge Sort for second half:
    Call merge Sort (arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
    Calls merge (arr, l, m, r)

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
#define max 20
voidmergesort(int a[],intlow,int high);
void merge(int a[],intlow,intmid,int high);
void main()
{
intn,i,a[max],ch=1;
clock_tstart,end;
clrscr();
while(ch)
{
printf("\n\t enter the number of elements\n");
scanf("%d",&n);
printf("\n\t enter the elements\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
start= clock();
mergesort(a,0,n-1);
```

```
end=clock();
printf("\nthe sorted array is\n");
for(i=0;i<n;i++)
printf("%d\n",a[i]);
printf("\n\ntime taken=%lf",(end-start)/CLK_TCK);
printf("\n\ndo u wish to continue(0/1) \n");
scanf("%d",&ch);
}
getch();
}  voidmergesort(int a[],intlow,int high) {
int mid;
delay(100);
if(low<high) {
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
merge(a,low,mid,high);
}
}
void merge(int a[],intlow,intmid,int high) {
inti,j,k,t[max];
i=low;
j=mid+1;
k=low;
while((i<=mid) && (j<=high))
if(a[i]<=a[j])
t[k++]=a[i++];
else
t[k++]=a[j++];
while(i<=mid)
t[k++]=a[i++];
while(j<=high)
t[k++]=a[j++];
for(i=low;i<=high;i++)
a[i]=t[i];
}
```

OUTPUT:

RESULT:

Ex.No-4	Check whether a given graph is connected or not using DFS method.	Date:

Aim:

To check whether a given graph is connected or not using DFS method.

Description:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Algorithm:

DFS(G,v) (v is the vertex where the search starts)
Stack S := {}; (start with an empty stack)
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
u := pop S;
if (not visited[u]) then
visited[u] := true;
for each unvisited neighbour w of u
push S, w;
end if
end while
END DFS()

Source code:

```
#include<stdio.h>
int visit[20],n,adj[20][20],s,count=0;
voiddfs(int v)
{
int w;
visit[v]=1;
count++;
for(w=1;w<=n;w++)
if((adj[v][w]==1) && (visit[w]==0))
dfs(w);
}
void main()
{
intv,w;
printf("Enter the no.of vertices:");
scanf("%d",&n);
printf("Enter the adjacency matrix:\n");
for(v=1;v<=n;v++)
```

```
for(w=1;w<=n;w++)
scanf("%d",&adj[v][w]);
for(v=1;v<=n;v++)
visit[v]=0;
dfs(1);
if(count==n)
printf("\nThe graph is connected");
else
printf("The graph is not connected");
}
```

OUTPUT:

RESULT:

Ex.No-5	Print all the nodes reachable from a given starting node in a directed graph using BFS method	Date:

Aim:

To print all the nodes reachable from a given starting node in a directed graph using BFS method.

Description:

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors.

Algorithm:

1. Start from node s.
2. Visit all neighbors of node s.
3. Then visit all of their neighbors, if not already visited
4. Continue until all nodes visited

Source code:

```
#include<stdio.h>
#define size 20
#define true 1
#define false 0
int queue[size],visit[20],rear=-1,front=0;
intn,s,adj[20][20],flag=0;
voidinsertq(int v)
{
queue[++rear]=v;
}
intdeleteq()
{
return(queue[front++]);
}
intqempty()
{
if(rear<front)
return 1;
else
return 0;
}
voidbfs(int v)
{
int w;
visit[v]=1;
insertq(v);
```

```
while(!qempty())
{
v=deleteq();
for(w=1;w<=n;w++)

if((adj[v][w]==1) && (visit[w]==0))
{
visit[w]=1;
flag=1;
printf("v%d\t",w);
insertq(w);
}
}
}
void main()
{
intv,w;
printf("Enter the no.of vertices:\n");
scanf("%d",&n);
printf("Enter adjacency matrix:");
for(v=1;v<=n;v++)
{
for(w=1;w<=n;w++)
scanf("%d",&adj[v][w]);
}
printf("Enter the start vertex:");
scanf("%d",&s);
printf("Reachability of vertex %d\n",s);
for(v=1;v<=n;v++)
visit[v]=0;
bfs(s);
if(flag==0)
{
printf("No path found!!\n");
}
}
```

OUTPUT:

RESULT:

Ex.No-6	Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.	Date:

Aim:

To Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Description:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Algorithm:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Source code:

```
# include <stdio.h>
int Prim (int g[20][20], int n, int t[20][20])
{
intu,v, min, mincost;
int visited[20];
inti,j,k;
visited[1] = 1;
for(k=2; k<=n; k++)
visited[k] = 0 ;
mincost = 0;
for(k=1; k<=n-1; k++)
{
min= 99;
u=1;
v=1;
for(i=1; i<=n; i++)
if(visited[i]==1)
for(j=1; j<=n; j++)
if( g[i][j] < min )
{
min = g[i][j];
u = i;    v = j;
}
t[u][v] = t[v][u] = g[u][v] ;
```

```
mincost = mincost + g[u][v] ;
visited[v] = 1;
printf("\n (%d, %d) = %d", u, v, t[u][v]);
for(i=1; i<=n; i++)
for(j=1; j<=n; j++)
if( visited[i] && visited[j] )
g[i][j] = g[j][i] = 99;
}
return(mincost);
}
void main()
{
int n, cost[20][20], t[20][20];
intmincost,i,j;
printf("\nEnter the no of nodes: ");
scanf("%d",&n);
printf("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]=99;
}
for(i=1; i<=n; i++)
for(j=1; j<=n; j++)
t[i][j] = 99;
printf("\nThe order of Insertion of edges:");
mincost = Prim (cost,n,t);
printf("\nMinimum cost = %d\n\n", mincost);
}
```

OUTPUT:

RESULT:

Ex.No-7	Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.	Date:

Aim:

To Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

Description:

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

Algorithm:

Let G = (V, E) be the given graph, with | V| = n
{
Start with a graph T = (V,ϕ) consisting of only the
Vertices of G and no edges;
/* This can be viewed as n connected components, each vertex being one connected component */
Arrange E in the order of increasing costs;
for (i = 1, i≤n - 1, i + +)
{
Select the next smallest cost edge;
if the edge connects two different connected components
add the edge to T;
}
}

Source Code:

```
#include<stdio.h>
int parent[20],min,mincost=0,ne=1,n,cost[20][20];
inta,b,i,j,u,v;
void main()
{
printf("Enter the no of nodes\n");
scanf("%d",&n);
printf("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]=99;
}
while(ne <n)
{
for(i=1,min=99;i<=n;i++)
for(j=1;j<=n;j++)
```

```
if(cost[i][j] < min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
While (parent[u])
u=parent[u];
while(parent[v])
v=parent[v];
if(u!=v)
{
printf("%d\t edge \t (%d,%d)=%d\n",ne++,a,b,min);
mincost+=min;
parent[v]=u;
}
cost[a][b]=cost[b][a]=99;
}
printf("The minimum cost=%d\n",mincost);
}
```

OUTPUT:

RESULT:

Ex.No-8	Implement 0/1 Knapsack problem using Dynamic Programming.	Date:

Aim:

To implement 0/1 Knapsack problem using Dynamic Programming.

Description:

Problem Statement: A thief robbing a store and can carry a maximal weight of W into their knapsack. There are n items and i^{th} item weigh w and is worth v dollars. What items should thief take.

There are two versions of problem

Fractional knapsack problem: The setup is same, but the thief can take fractions of items, meaning that the items can be broken into smaller pieces so that thief may decide to carry only a fraction of xi of item i, where $0 \leq xi \leq 1$.

0-1 knapsack problem: The setup is the same, but the items may not be broken into smaller pieces, so thief may decide either to take an item or to leave it (binary choice), but may not take a fraction of an item.

Algorithm:

```
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
if weight + w[i] ≤ W then
x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
break
return x
```

Source code:

```
#include<stdio.h>
void kpsk(int ,int ,int[],int[],int[][100]);
void opt(int,int,int[],int[][100]);
int max(int,int);
void main()
{
int w[20],p[20],n,m,i,v[10][100];
printf("Enter the no of elements\n");
scanf("%d",&n);
printf("Enter the capacity of knapsack\n");
scanf("%d",&m);
printf("Enter the weight of elements\n");
for(i=1;i<=n;i++)
```

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

printf("Enter the profit of elements\n");
for(i=1;i<=n;i++)
{
scanf("%d",&p[i]);
}
kpsk(n,m,w,p,v);
opt(n,m,w,v);
}
void kpsk(intn,intm,int w[],int p[],int v[][100])
{
inti,j;
for(i=0;i<=n;i++)
for(j=0;j<=m;j++)
{
if(i==0 || j==0)
v[i][j]=0;
else if(j<w[i])
v[i][j]=v[i-1][j];
else
v[i][j]=max(v[i-1][j],p[i]+v[i-1][j-w[i]]);
}
for(i=0;i<=n;i++)
{
for(j=0;j<=m;j++)
{
printf("%d\t",v[i][j]);
}
printf("\n");
}
}
void opt(intn,intm,int w[],int v[][100])
{
inti,j,x[10];
printf("The optimal solution is %d\n",v[n][m]);
for(i=0;i<n;i++)
x[i]=0;
i=n;
j=m;
while((i!=0)&& (j!=0))
{
if(v[i][j]!=v[i-1][j])
{
x[i]=1;
j=j-w[i];
```

```
}
i=i-1;
}
printf("The objects selected are \n");
for(i=1;i<=n;i++)
{
if(x[i]==1)
printf("%d ",i);
}
}
int max(int a, int b)
{
return(a>b? a:b);
}
```

OUTPUT:

RESULT:

Ex.No-9	Write a program to implement Travelling Sales Person problem using Dynamic programming.	Date:

Aim:

To write a program to implement Travelling Sales Person problem using Dynamic programming.

Description:

Problem statement: Given a graph and a source vertex in graph, find the shortest route between a set of points and locations that must be visited. ... Focused on optimization, Travelling Sales Person is often used in computer science to find the most efficient route for data to travel between various nodes.

Algorithm:

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! Permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Source Code:

```
#include<stdio.h>
#include<conio.h>
int a[10][10],visited[10],n,cost=0;
void get()
{
inti,j;
printf("Enter No. of Cities: ");
scanf("%d",&n);
printf("\nEnter 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",&a[i][j]);
visited[i]=0;
}
printf("\n\nThe cost list is:\n\n");
for( i=0;i <n;i++)
{
printf("\n\n");
for(j=0;j <n;j++)
printf("\t%d",a[i][j]);
}
}
voidmincost(int city)
{
```

```
inti,ncity;
visited[city]=1;
printf("%d -->",city+1);
ncity=least(city);
if(ncity==999)
{
ncity=0;
printf("%d",ncity+1);
cost+=a[city][ncity];
return;
}
mincost(ncity);
}
int least(int c)
{
inti,nc=999;
int min=999,kmin;
for(i=0;i <n;i++)
{
if((a[c][i]!=0)&&(visited[i]==0))
if(a[c][i] < min)
{
min=a[i][0]+a[c][i];
kmin=a[c][i];
nc=i;
}
}
if(min!=999)
cost+=kmin;
returnnc;
}
void put()
{
printf("\n\nMinimum cost:");
printf("%d",cost);
}
void main()
{
clrscr();
get();
printf("\n\nThe Path is:\n\n");
mincost(0);
put();
getch();
}
```

OUTPUT:

RESULT:

Ex.No-10	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	Date:

Aim:

To find shortest paths to other vertices using Dijkstra's algorithm in a weighted connected graph.

Description:

Problem statement: Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Algorithm:

- Create a shortest path tree set 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.
- 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.
- While shortest path tree set doesn't include all vertices
- Pick a vertex u which is not there in shortest path tree set and has minimum distance value.
- Include u to shortest path tree set
- 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>
voiddij(int,int);
int min(int);
int a[20][20],dis[20],s[20],i,j;
void main()
{
intv,n;
printf("Enter the no of nodes\n");
scanf("%d",&n);
printf("Enter the cost matrix\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
printf("Enter the vertex\n");
scanf("%d",&v);
dij(v,n);
for(i=1;i<=n;i++)
printf("From %d to %d is %d\n",v,i,dis[i]);
}
voiddij(intv,int n)
```

```
{
intw,u,k;
for(i=1;i<=n;i++)
{
s[i]=0;
dis[i]=a[v][i];
}
s[v]=1;
dis[v]=0;
for(i=2;i<=n;i++)
{
u=min(n);
s[u]=1;
for(w=1;w<=n;w++)
{
if(dis[w] > dis[u]+a[u][w])
dis[w]=dis[u]+a[u][w];
else
dis[w]=dis[w];
}
}
}
int min(int n)
{
inti,p,min=99;
for(i=1;i<=n;i++)
{
if(min > dis[i] && s[i]==0)
{
min=dis[i];
p=i;
}
}
return p;
}
```

OUTPUT:

RESULT:

Ex.No-11	Design and implement the presence of Hamiltonian Cycle in an undirected Graph G of n vertices.	Date:

Aim:

To design and implement the presence of Hamiltonian Cycle in an undirected Graph G of n vertices.

Description:

Problem statement: Given a graph and a source vertex in undirected Graph, find Hamiltonian Cycle from source to all vertices in the given graph.

Algorithms:

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

Source Code:

```
#include<stdio.h>
// Number of vertices in the graph
#define V 5
voidprintSolution(int path[]);
/* A utility function to check if the vertex v can be added at index 'pos' in the Hamiltonian Cycle constructed so far (storedin 'path[]') */
boolisSafe(int v, bool graph[V][V], int path[], intpos)
{
/* 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 */
boolhamCycleUtil(bool graph[V][V], int path[], intpos)
{
/* 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. */
boolhamCycle(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 isa Hamiltonian Cycle, then the path
can be started from any pointof 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 */
voidprintSolution(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()
{
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);
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:

RESULT: