NEAR ITPB, CHANNASANDRA, BENGALURU – 560 067
Affiliated to VTU, Belagavi
Approved by AICTE, New Delhi
Recognized by UGC under 2(f) &12(B)
Accredited by NBA & NAAC

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IV SEMESTER

ANALYSIS AND DESIGN OF ALGORITHM LAB MANUAL
MVJ22CSL44

ACADEMIC YEAR 2024 – 2025 [EVEN]

LABORATORY MANUAL

NAME OF THE STUDENT : ----------

BRANCH : ----------

UNIVERSITY SEAT NO. : ----------

SEMESTER & SECTION : ----------

BATCH : ----------

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION:

To create an ambience in excellence and provide innovative emerging programs in Computer Science and Engineering and to bring out future ready engineers equipped with technical expertise and strong ethical values.

MISSION:

1. **Concepts of computing discipline**: To educate students at under graduate, postgraduate and doctoral levels in the fundamental and advanced concepts of computing discipline.

2. **Quality Research**: To provide strong theoretical and practical background across the Computer Science and Engineering discipline with the emphasis on computing technologies, quality research, consultancy and training's.

3. **Continuous Teaching Learning:** To promote a teaching learning process that brings advancements in Computer Science and Engineering discipline leading to new technologies and products.

4. **Social Responsibility and Ethical Values:** To inculcate professional behavior, innovative research Capabilities, leadership abilities and strong ethical values in the young minds so as to work with the commitment for the betterment of the society.

Programme Educational Objectives (PEOs):

PEO01: **Current Industry Practices**: Graduates will analyze real world problems and give solution using current industry practices in computing technology.

PEO02: **Research & Higher Studies:** Graduates with strong foundation in mathematics and engineering fundamentals will pursue higher learning, R&D activities and consultancy.

PEO03: **Social Responsibility:** Graduates will be professionals with ethics, who will provide industry growth and social transformation as responsible citizens.

Programme Outcomes (POs):

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

2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **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.

4. **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.

5. **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.

6. **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.

7. **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.

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

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **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.

11. **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.

12. **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.

Program Specific Outcome:

PSO1 : Programming: Ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, DBMS, and networking for efficient design of computer-based systems of varying complexity.

PSO2: Practical Solution: Ability to practically provide solutions for real world problems with a broad range of programming language and open source platforms in various computing domains.

Course objectives:

- To Design and implement various algorithms in C/C++ programming using suitable development tools to address different computational challenges.
- To Apply diverse design strategies for effective problem solving
- To measure and compare the performance of different algorithms to determine their efficiency and sustainability for specific tasks.

Course outcomes:

At the end of the course the student will be able to:
C204.1. Develop programsto solve computational problems using suitable algorithm design strategy.
C204.2. Compare algorithm design strategies by developing equivalent programs and observing running times for analysis (Empirical).
C204.3. Make use of suitable integrated development tools to develop programs
C204.4. Choose appropriate algorithm design techniques to develop solution to the computational and complex problems.
C204.5. Demonstrate and present the development of program, its execution and running time(s) and record the results/inferences.

Prerequisites: Basic programming Languages like
C,C++,Java.

CONTENTS

	Laboratory Experiment	Revised Bloom's Taxonomy Levels (RBT Level)
1.	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.	**L3**
2.	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.	**L3**
3.	a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm. b. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.	**L3**
4.	Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.	**L3**
5.	Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.	**L3**
6.	Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.	**L3**
7.	Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.	**L3**
8.	Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.	**L3**
9.	Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	**L3**
10.	Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	**L3**
11.	Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator	**L3**
12.	Design and implement C/C++ Program for N Queen's problem using Backtracking	**L3**

1. **Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.**

PROGRAM:

```
// C code to implement Kruskal's algorithm

#include <stdio.h>
#include <stdlib.h>

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
const int(*x)[3] = p1;
const int(*y)[3] = p2;

return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
for (int i = 0; i < n; i++) {
parent[i] = i;
rank[i] = 0;
}
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
if (parent[component] == component)
return component;

return parent[component]
= findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
// Finding the parents
u = findParent(parent, u);
v = findParent(parent, v);

if (rank[u] < rank[v]) {
```

```
parent[u] = v;
}
else if (rank[u] > rank[v]) {
parent[v] = u;
}
else {
parent[v] = u;

// Since the rank increases if
// the ranks of two sets are same
rank[u]++;
}
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
// First we sort the edge array in ascending order
// so that we can access minimum distances/cost
qsort(edge, n, sizeof(edge[0]), comparator);

int parent[n];
int rank[n];

// Function to initialize parent[] and rank[]
makeSet(parent, rank, n);

// To store the minimun cost
int minCost = 0;

printf(
"Following are the edges in the constructed MST\n");
for (int i = 0; i < n; i++) {
int v1 = findParent(parent, edge[i][0]);
int v2 = findParent(parent, edge[i][1]);
int wt = edge[i][2];

// If the parents are different that
// means they are in different sets so
// union them
if (v1 != v2) {
unionSet(v1, v2, parent, rank, n);
minCost += wt;
printf("%d -- %d == %d\n", edge[i][0],
```

```
edge[i][1], wt);
}
}

printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
{
int edge[5][3] = { { 0, 1, 10 },
{ 0, 2, 6 },
{ 0, 3, 5 },
{ 1, 3, 15 },
{ 2, 3, 4 } };

kruskalAlgo(5, edge);

return 0;
}
```

Output:

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

2. **Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.**

```
// A C program for Prim's Minimum
```

```
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
// Initialize min value
int min = INT_MAX, min_index;

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

return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
printf("Edge \tWeight\n");
for (int i = 1; i < V; i++)
printf("%d - %d \t%d \n", parent[i], i,
graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
// Array to store constructed MST
int parent[V];
// Key values used to pick minimum weight edge in cut
int key[V];
```

```
// To represent set of vertices included in MST
bool mstSet[V];

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
key[i] = INT_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.
// Make key 0 so that this vertex is picked as first
// vertex.
key[0] = 0;

// First node is always root of MST
parent[0] = -1;

// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the
// set of vertices not yet included in MST
int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set
mstSet[u] = true;

// Update key value and parent index of
// the adjacent vertices of the picked vertex.
// Consider only those vertices which are not
// yet included in MST
for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent
// vertices of m mstSet[v] is false for vertices
// not yet included in MST Update the key only
// if graph[u][v] is smaller than key[v]
if (graph[u][v] && mstSet[v] == false
&& graph[u][v] < key[v])
parent[v] = u, key[v] = graph[u][v];
}

// print the constructed MST
printMST(parent, graph);
}
```

```
// Driver's code
int main()
{
int graph[V][V] = { { 0, 2, 0, 6, 0 },
{ 2, 0, 3, 8, 5 },
{ 0, 3, 0, 0, 7 },
{ 6, 8, 0, 0, 9 },
{ 0, 5, 7, 9, 0 } };

// Print the solution
primMST(graph);

return 0;
}
```

OUTPUT:

```
Edge   Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

3. a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

```
// C Program for Floyd Warshall Algorithm
#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 dist[][V])
{
   int i, j, k;

   /* 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's code
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 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

OUTPUT:

The following matrix shows the shortest distances between every pair of vertices

```
  0     5     8     9
INF     0     3     4
INF   INF     0     1
INF   INF   INF     0
```

3.b. **Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.**

```
// Program for transitive closure
// using Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

// A function to print the solution matrix
```

```
void printSolution(int reach[][V]);

// Prints transitive closure of graph[][]
// using Floyd Warshall algorithm
void transitiveClosure(int graph[][V])
{
/* reach[][] will be the output matrix
// that will finally have the
shortest distances between
every pair of vertices */
int reach[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++)
                reach[i][j] = graph[i][j];

/* Add all vertices one by one to the
set of intermediate vertices.
---> Before start of a iteration,
        we have reachability values for
        all pairs of vertices such that
        the reachability values
        consider only the vertices in
        set {0, 1, 2, .. k-1} as
        intermediate vertices.
----> After the end of a iteration,
        vertex no. k is added to the
                set of intermediate vertices
                and the set becomes {0, 1, .. 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 a path
                        // from i to j,
                        // then make sure that the value
                        // of reach[i][j] is 1
                        reach[i][j] = reach[i][j] ||
                        (reach[i][k] && reach[k][j]);
                }
        }
}

// Print the shortest distance matrix
printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[][V])
{
printf ("Following matrix is transitive");
printf("closure of the given graph\n");
for (int i = 0; i < V; i++)
{
        for (int j = 0; j < V; j++)
        {
                /* because "i==j means same vertex"
                and we can reach same vertex
                from same vertex. So, we print 1....
                and we have not considered this in
                Floyd Warshall Algo. so we need to
                make this true by ourself
                while printing transitive closure.*/
                if(i == j)
                        printf("1 ");
                else
                        printf ("%d ", reach[i][j]);
        }
        printf("\n");
}
}

// Driver Code
int main()
{
```

```
/* Let us create the following weighted graph
                10
(0)------->(3)
          |                 /|\
5 |                 |
          |                 | 1
\|/                 |
(1)------->(2)
                3                 */
int graph[V][V] = { {1, 1, 0, 1},
                                        {0, 1, 1, 0},
                                        {0, 0, 1, 1},
                                        {0, 0, 0, 1}
                                    };

// Print the solution
transitiveClosure(graph);
return 0;
}
```

OUTPUT:

Following matrix is transitive closure of the given graph
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1

4. **Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.**

```
// C program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9
```

```
// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;

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

return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
printf("Vertex \t\t Distance from Source\n");
for (int i = 0; i < V; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
int dist[V]; // The output array. dist[i] will hold the
                        // shortest
// distance from src to i

bool sptSet[V]; // sptSet[i] will be true if vertex i is
                                // included in shortest
// path tree or shortest distance from src to i is
// finalized

// 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 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. u is always equal to
        // src in the first iteration.
        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 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);
}

// driver's code
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
```

```
// Function call
dijkstra(graph, 0);

return 0;
}
```

OUTPUT:

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

5. **Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.**

```
/*To obtain the topological order in of vertices in a digraph.*/
#include<stdio.h>
void findindegree(int [10][10],int[10],int);
void topological(int,int [10][10]);
void main()
{int a[10][10],i,j,n;
        clrscr();
        printf("Enter the number of nodes:");
        scanf("%d",&n);
        printf("\nEnter the adjacency matrix\n");
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
                scanf("%d",&a[i][j]);
```

```
        printf("\nThe adjacency matirx is:\n");
        for(i=1;i<=n;i++)
        {for(j=1;j<=n;j++)
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
        }
        topological(n,a);
        getch();
}
void findindegree(int a[10][10],int indegree[10],int n)
{
int i,j,sum;
for(j=1;j<=n;j++)
{
        sum=0;
        for(i=1;i<=n;i++)
        {
                sum=sum+a[i][j];
        }
        indegree[j]=sum;
}}
void topological(int n,int a[10][10])
{
int k,top,t[100],i,stack[20],u,v,indegree[20];
k=1;
top=-1;
findindegree(a,indegree,n);
for(i=1;i<=n;i++)
{
        if(indegree[i]==0)
        {
                stack[++top]=i;
        }
}
while(top!=-1)
{
        u=stack[top--];
        t[k++]=u;
        for(v=1;v<=n;v++)
        {
                if(a[u][v]==1)
                {
```

```
                        indegree[v]--;
                        if(indegree[v]==0)
                        {
                                stack[++top]=v;
                        }
                }
        }}
printf("\nTopological sequence is\n");
for(i=1;i<=n;i++)
        printf("%d\t",t[i]);
}
```

OUTPUT:

```
Enter the number of nodes:2

Enter the adjacency matrix
4 5
9 6

The adjacency Matirx is:
4    5
9    6

Topological sequence is
31637        -532947266
```

6. **Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.**

```
#include <stdio.h>

// Function to calculate the maximum value using Dynamic Programming
int knapsack(int W, int weights[], int values[], int n) {
   int dp[n+1][W+1];  // dp[i][w] stores the maximum value for the first i items and a knapsack of capacity w

   // Initialize the dp table with 0 for base cases
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0) {
            dp[i][w] = 0;
         } else if (weights[i-1] <= w) {
            // If the weight of the current item is less than or equal to the capacity w, we have two options:
```

```
            // 1. Include the item (add its value to the remaining capacity)
            // 2. Exclude the item (take the best value without including the item)
            dp[i][w] = (values[i-1] + dp[i-1][w - weights[i-1]]) > dp[i-1][w] ?
                    (values[i-1] + dp[i-1][w - weights[i-1]]) : dp[i-1][w];
        } else {
            dp[i][w] = dp[i-1][w];  // Exclude the item
        }
    }
  }

  return dp[n][W];  // The value in the bottom-right corner is the maximum value we can achieve
}

int main() {
  int n, W;

  // Taking input for number of items and the knapsack capacity
  printf("Enter number of items: ");
  scanf("%d", &n);
  printf("Enter the capacity of the knapsack: ");
  scanf("%d", &W);

  int weights[n], values[n];

  // Input the weights and values of the items
  printf("Enter the weights of the items:\n");
  for (int i = 0; i < n; i++) {
    scanf("%d", &weights[i]);
  }

  printf("Enter the values of the items:\n");
  for (int i = 0; i < n; i++) {
    scanf("%d", &values[i]);
  }

  // Call the knapsack function
  int maxValue = knapsack(W, weights, values, n);
  printf("The maximum value that can be obtained is: %d\n", maxValue);

  return 0;
}
```

OUTPUT :

```
Enter number of items: 4
Enter the capacity of the knapsack: 30
Enter the weights of the items:
```

```
14
15
19
20
Enter the values of the items:
15
6
8
7
The maximum value that can be obtained is: 21
```

7. **Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.**

```
#include<stdio.h>
int w[10],p[10],n;
int max(int a,int b)
{
   return a>b?a:b;
}
int knap(int i,int m)
{
   if(i==n) return w[i]>m?0:p[i];
   if(w[i]>m) return knap(i+1,m);
   return max(knap(i+1,m),knap(i+1,m-w[i])+p[i]);
}
int main()
{
   int m,i,max_profit;
   printf("\nEnter the no. of objects:");
   scanf("%d",&n);
   printf("\nEnter the knapsack capacity:");
   scanf("%d",&m);
   printf("\nEnter profit followed by weight:\n");
   for(i=1; i<=n; i++)
```

```
    scanf("%d %d",&p[i],&w[i]);
  max_profit=knap(1,m);
  printf("\nMax profit=%d",max_profit);
  return 0;
}
```

OUTPUT:

```
Enter the no. of objects:5

Enter the knapsack capacity:40

Enter profit followed by weight:
10 14
14 15
14 16
12 7
12 15

Max profit=40
```

8. Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

```
#include<stdio.h>
//#include<conio.h>
void subset(int,int,int);
int x[10],w[10],d,count=0;
void main()
{
int i,n,sum=0;
//clrscr();
printf("Enter the no. of elements: ");
scanf("%d",&n);
printf("\nEnter the elements in ascending order:\n");
for(i=0;i<n;i++)
scanf("%d",&w[i]);
printf("\nEnter the sum: ");
scanf("%d",&d);
for(i=0;i<n;i++)
sum=sum+w[i];
if(sum<d)
{
printf("No solution\n");
```

```
//getch();
return;
}
subset(0,0,sum);
if(count==0)
{
printf("No solution\n");
//getch();
return;
}
//getch();
}
void subset(int cs,int k,int r)
{
int i;
x[k]=1;
if(cs+w[k]==d)
{
printf("\n\nSubset %d\n",++count);
for(i=0;i<=k;i++)
if(x[i]==1)
printf("%d\t",w[i]);
}
else if(cs+w[k]+w[k+1]<=d)
subset(cs+w[k],k+1,r-w[k]);
if(cs+r-w[k]>=d && cs+w[k]<=d)
{
x[k]=0;
subset(cs,k+1,r-w[k]);
}
}
```

OUTPUT :

Enter the no. of elements: 4

Enter the elements in ascending order:
10
20
30
40

Enter the sum: 10

Subset 1
10

9. Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

```
// C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void selectionSort(int arr[], int n)
{
   int i, j, min_idx;

   // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
```

```
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
           if(min_idx != i)
            swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
      }
```

OUTPUT:

Sorted array:
11 12 22 25 64

10. Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to swap two elements
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}

// Partition function for Quick Sort
int partition(int arr[], int low, int high)
{
   int pivot = arr[high]; // Pivot element
   int i = (low - 1); // Index of smaller element

   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j] < pivot)
      {
         i++; // Increment index of smaller element
```

```
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to generate random numbers
void generateRandomNumbers(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100000; // Generate random numbers between 0 and 99999
    }
}

int main()
{
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n); // Read the number of elements from the user

    if (n <= 5000)
    {
        printf("Please enter a value greater than 5000\n");
        return 1; // Exit if the number of elements is not greater than 5000
    }

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL)
    {
        printf("Memory allocation failed\n");
        return 1; // Exit if memory allocation fails
```

```
    }

    // Generate random numbers and store them in the array
    generateRandomNumbers(arr, n);

    // Measure the time taken to sort the array
    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();

    // Calculate and print the time taken to sort the array
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    // Free the allocated memory
    free(arr);
    return 0;
}
```

OUTPUT :

```
Enter number of elements: 5689
Time taken to sort 5689 elements: 0.000568 seconds
```

11. Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to merge two sorted arrays
void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
```

```
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

// Function to generate random integers
void generateRandomArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100000; // Generate random integers between 0 and 99999
```

```
}

int main()
{
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   if (n <= 5000)
   {
      printf("Please enter a value greater than 5000\n");
      return 1; // Exit if the number of elements is not greater than 5000
   }

   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL)
   {
      printf("Memory allocation failed\n");
      return 1; // Exit if memory allocation fails
   }

   generateRandomArray(arr, n);

   // Repeat the sorting process multiple times to increase duration for timing
   clock_t start = clock();
   for (int i = 0; i < 1000; i++)
   {
      mergeSort(arr, 0, n - 1);
   }
   clock_t end = clock();

   // Calculate the time taken for one iteration
   double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;

   printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

   free(arr);
   return 0;
}
```

OUTPUT:

```
Enter the number of elements: 5689
Time taken to sort 5689 elements: 0.000531 seconds
```

12. Design and implement C/C++ Program for N Queen's problem using Backtracking

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

#define MAX 20

int board[MAX][MAX];

// Function to print the chessboard
void printBoard(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == 1)
                printf(" Q ");
            else
                printf(" . ");
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed on board[row][col]
// It checks if there is a queen in the same column, upper left diagonal or upper right diagonal
bool isSafe(int row, int col, int n) {
    int i, j;

    // Check this column on the upper side
    for (i = 0; i < row; i++)
        if (board[i][col] == 1)
            return false;
```

```
    // Check upper left diagonal
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;

    // Check upper right diagonal
    for (i = row, j = col; i >= 0 && j < n; i--, j++)
        if (board[i][j] == 1)
            return false;

    return true;
}

// Backtracking function to solve N Queens problem
bool solveNQueens(int row, int n) {
    // If all queens are placed, return true
    if (row == n)
        return true;

    for (int col = 0; col < n; col++) {
        // Check if it's safe to place the queen in the current cell
        if (isSafe(row, col, n)) {
            // Place the queen
            board[row][col] = 1;

            // Recur to place queens in the next row
            if (solveNQueens(row + 1, n))
                return true;

            // If placing queen in current cell doesn't lead to a solution, backtrack
            board[row][col] = 0;
        }
    }

    return false; // No solution found
}

int main() {
    int n;

    // Input the size of the chessboard (number of queens)
    printf("Enter the value of N (for N x N board): ");
    scanf("%d", &n);
```

```
    // Initialize the board with 0 (no queens placed)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            board[i][j] = 0;

    // Solve the N Queens problem
    if (solveNQueens(0, n)) {
        printf("Solution found:\n");
        printBoard(n);
    } else {
        printf("No solution exists for N = %d\n", n);
    }

    return 0;
}
```

OUTPUT:

```
Enter the value of N (for N x N board): 4
Solution found:
 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .
```

VIVA QUESTIONS

General Questions:

1. What is an algorithm?
2. What are algorithm design techniques?
3. How is an algorithm's time efficiency measured?
4. What is Big 'Oh', Theta and Omega notation?
5. Define order of an algorithm
6. What is recursive call?
7. What do you mean by time complexity and space complexity of an algorithm?
8. What are the characteristics of an algorithm?
9. Define best case, average case and worst case efficiency.
10. Write the difference between the Greedy method and Dynamic programming.
11. Define Branch-and-Bound method.
12. What is class P and NP?
13. What are NP- hard and Np-complete problems?
14. What is a decision problem?
15. What is approximate solution?
16. What is Master's theorem?
17. What is a directed graph?
18. What do you mean by transitive closure?
19. How does Warshall's algorithm work?
20. What is the time complexity of Warshall's algorithm?
21. What is the difference between path and edge?
22. What is path matrix?
23. Which are the two graph traversal techniques?
24. What do you mean by BFS? How does it work?
25. Which algorithm design technique, does it come under?
26. What are the applications of BFS?
27. Explain the differences between BFS and DFS.
28. How does Decrease and Conquer design methodology work?
29. What are the 3 variants of Decrease and Conquer?
30. What is the time complexity of BFS?

31. What is tree edge and cross edge?
32. What is backtracking?
33. Name few problems that can be solved using backtracking.
34. What is traveling salesman problem? Show with an example.
35. What is the time complexity of TSP, using dynamic programming?
36. What are approximation algorithms?
37. What is N-Queen's problem?
38. What is 4-Queen's problem? Show how the 4 queens can be placed.
39. What is state-space tree? Construct the state space tree for 4-Queens problem.
40. What is Backtracking?
41. Name few problems that come under Backtracking design technique.

SAFETY INSTRUCTIONS

Do's

1. Do wear ID card and follow dress code.
2. Be on time to Lab sessions.
3. Do log off the computers when you finish.
4. Do ask the staff for assistance if you need help.
5. Do keep your voice low when speaking to others in the LAB.
6. Do ask for assistance in downloading any software.
7. Do make suggestions as to how we can improve the LAB.
8. In case of any hardware related problem, ask LAB in charge for solution.
9. If you are the last one leaving the LAB , make sure that the staff in charge of the LAB is informed to close the LAB.
10. Be on time to LAB sessions.
11. Do keep the LAB as clean as possible.

Don'ts

1. Do not use mobile phone inside the lab.
2. Don't do anything that can make the LAB dirty (like eating, throwing waste papers etc).
3. Do not carry any external devices without permission.
4. Don't move the chairs of the LAB.
5. Don't interchange any part of one computer with another.
6. Don't leave the computers of the LAB turned on while leaving the LAB.
7. Do not install or download any software or modify or delete any system files on any lab computers.
8. Do not damage, remove, or disconnect any labels, parts, cables, or equipment.
9. Don't attempt to bypass the computer security system.
10. Do not read or modify other user's file.

11. If you leave the lab, do not leave your personal belongings unattended. We are not responsible for any theft.