JEPPIAAR INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

Self-Belief | Self Discipline | Self Respect

Kunnam, Sunguvarchatram, Sriperumbudur-631604

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

ACS301-DATA STRUCTURES AND ALGORITHMS LABORATORY

NAME	:	
REG NO.	:	
YEAR	:	**I**
SEMESTER	:	**02**
BRANCH	:	

JEPPIAAR INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

Self-Belief | Self Discipline | Self Respect

Kunnam, Sunguvarchatram, Sriperumbudur-631604

BONAFIDE CERTIFICATE

This is a certified Bonafide Record Work of Mr./Ms.______________________________

Register No.____________________submitted for the Anna University Practical Examination held on______________in ACS301 **DATA STRUCTURES AND ALGORITHMS LABORATORY** during the year 2024-2025.

Signature of the Lab In-charge

Head of the Department

Internal Examiner

External Examiner

INSTITUTE VISION:

Jeppiaar Institute of Technology aspires to provide technical education in futuristic technologies with the perspective of innovative, industrial and social application for the betterment of humanity.

INSTITUTE MISSION:

- **M1:** To produce competent and disciplined high-quality professionals with the practical skills necessary to excel as innovative professionals and entrepreneurs for the benefit of the society.
- **M2:** To improve the quality of education through excellence in teaching and learning, research, leadership and by promoting the principles of scientific analysis, and creative thinking.
- **M3:** To provide excellent infrastructure, serene and stimulating environment that is most conducive to learning.
- **M4:** To strive for productive partnership between the Industry and the Institute for research and development in the emerging fields and creating opportunities for employability.
- **M5:** To serve the global community by instilling ethics, values and life skills among the students needed to enrich their lives.

VISION

- ❖ The department will serve as a centre of excellence in practicing, training and implementing AI and AI associated techniques that will enable /support innovative thoughts and ideas across industries and society

MISSION

- ❖ M1: To collaborate with industry and provide the state of the art infrastructural Facilities to meet the global requirements and societal needs for AI.
- ❖ M2: Promote learning and development of students in Artificial Intelligence thought leadership, by providing them a suitable infrastructure and Environment, enabling them to grow into successful entrepreneurs.
- ❖ M3: To encourage students to pursue higher education and research in the field of AI.
- ❖ M4: To impart moral and ethical values in their profession

PROGRAMME EDUCATIONAL OBJECTIVES

- ❖ PEO 1: Utilize their proficiencies in the fundamental knowledge of basic sciences, mathematics, Artificial Intelligence, data science and statistics to build systems that require management and analysis of large volumes of data.

- ❖ PEO 2: Advance their technical skills to pursue pioneering research in the field of AI and Data Science and create disruptive and sustainable solutions for the welfare of ecosystems.

- ❖ PEO 3: Think logically, pursue lifelong learning and collaborate with an ethical attitude in a multidisciplinary team.

- ❖ PEO 4: Design and model AI based solutions to critical problem domains in the real world

- ❖ PEO 5: Exhibit innovative thoughts and creative ideas for effective contribution towards economy building

PROGRAM OUTCOMES

Engineering Graduates will be able to:

1. **Engineering knowledge:** (K3) Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis:** (K4) Identify, formulate, review 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:** (K4) 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:** (K5) 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:** (K3, K5, K6) Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** (A3) 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:** (A2) 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:** (A3) Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** (A3) Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** (A3) 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:** (A3) 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:** (A2) 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 OUTCOMES

PSO 1: To evolve AI based efficient domain specific processes for effective decision making in several domains such as business and governance domains.

PSO 2: To arrive at actionable Foresight, Insight, hindsight from data for solving business and engineering problems

PSO 3: To create, select and apply the theoretical knowledge of AI and Data Analytics along with practical industrial tools and techniques to manage and solve wicked societal problems

PSO 4: To develop data analytics and data visualization skills, skills pertaining to knowledge acquisition, knowledge representation and knowledge engineering, and hence be capable of coordinating complex projects.

PSO 5: To able to carry out fundamental research to cater the critical needs of the society through cutting edge technologies of AI.

SYLLABUS

ACS301 DATA STRUCTURES AND ALGORITHMS LABORATORY

1.Implement Linear Search and recursive Binary Search. Determine the time required to search for an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

2. Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt [ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m

3. Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required 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.

4. Develop a program to implement graph traversal using Breadth First Search and Depth First Search.

5. From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.

6. Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

7. Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique,

8. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. 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.

9. Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.

Content Beyond the syllabus

1.Perform the Towers of Hanoi Problem to transfer the disk from one tower to another.

2.Implement the Red-Black Tree with different operation Process.

TABLE OF CONTENTS

-

Course Outcome :

CO1	Implement Linear data structure algorithms using arrays and Linked lists.	K3
CO2	Analyze the efficiency of algorithms using various frameworks	K3
CO3	Analyze the various searching and sorting algorithms.	K4
CO4	Apply graph algorithms to solve problems and analyze their efficiency.	K2
CO5	Make use of algorithm design techniques like divide and conquer, dynamic programming and greedy techniques to solve problems.	K3
Mapping of Cos with Pos and PSOs		

COs/ Pos	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO 1**	**PSO 2**	**PSO3**
CO1	2	1	-	-	2	2	3	1	2	2	2	2	3	3	3
CO2	3	2	-	-	3	3	3	1	2	2	2	2	3	3	3
CO3	3	-	1	-	2	2	2	1	2	2	2	2	3	3	3
CO4	3	2	1	1	2	2	2	1	2	2	2	2	3	3	3
CO5	3	2	1	-	2	2	2	1	2	2	2	1	3	3	3
1–Slight,2–Moderate,3–Substantial,BT-Bloom'sTaxonomy															

Ex.No:1 **Date:**	**Implement Linear Search and recursive Binary Search. Determine the time required to search for an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.**

Aim:

To implement **Linear Search** and **Recursive Binary Search**, measure the time required to search for an element, and analyze the performance by plotting a graph of time taken versus the number of elements in the list.

Algorithm:

1. Linear Search Algorithm
Step 1: Start with an unsorted list and the key (element to search).
Step 2: Traverse the list sequentially.
Step 3: If an element matches the key, return its index.
Step 4: If the end of the list is reached without finding the key, return -1.
Step 5: Stop.

Program :

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
#include<stdlib.h>
#define max 20
int pos;
int binsearch (int,int[],int,int,int);
int linsearch (int,int[],int);
void main()
{
int ch=1;
double t;
int n,i,a [max],k,op,low,high,pos;
clock_tbegin,end;
clrscr();
while(ch)
{
printf("\n.......MENU.......\n 1.BinarySearch \n 2.Linear search \n 3.Exit \n");
printf("\n enter your choice\n");
scanf("%d",&op);
switch(op)
{
case 1:printf("\n enter the number of elments\n");
scanf("%d",&n);
printf("\n enter the number of an array in the order \n");
for(i=0;i<n;i++)
```

```
    scanf("%d",&a[i]);
    printf("\n enter the elements to be searched \n");
    scanf("%d",&k);
    low=0;high=n-1;
    begin=clock();
    pos=binsearch(n,a,k,low,high);
    end=clock();
    if(pos==-1)
    printf("\n\nUnsuccessful search");
      else
   printf("\n element %d is found at position %d",k,pos+1);
   printf("\n Time Taken is %lf CPU1 cycles \n",(end-begin)/CLK_TCK);
   getch();
   break;
   case 2:printf("\n enter the number of elements \n");
   scanf("%d",&n);
   printf("\n enter the elements of an array\n");
   for(i=0;i<n;i++)
   scanf("%d",&a[i]);
   printf("\n enter the element to be searched \n");
   scanf("%d",&k);
   begin=clock();
   pos=linsearch(n,a,k);
   end=clock();
   if(pos==-1)
   printf("\n\n Unsuccessful search");
  else
   printf("element %d is found at position %d",k,pos+1);
   printf("\n Time taken is %lf CPU cycles \n",(end-begin)/CLK_TCK);
   getch();
   break;
   default:printf("Invalid choice entered \n");
   exit(0);
   }
  printf("\n Do you wish to run again(1/0) \n");
  scanf("%d",&ch);
  }
 getch();
 }
 int binsearch(intn,int a[],intk,intlow,int high)
{
 int mid;
 delay(1000);
 mid=(low+high)/2;
 if(low>high)
 return -1;
 if(k==a[mid])
 return(mid);
else
 if(k<a[mid])
 return binsearch(n,a,k,low,mid-1);
```

```
else
returnbinsearch(n,a,k,mid+1,high);
}
intlinsearch(intn,int a[],int k)
{
delay(1000);
if(n<0)
return -1;
if(k==a[n-1])
return (n-1);
else
returnlinsearch(n-1,a,k);
}
```

OUTPUT:

```
-----------MENU-----------
1.BINARY SEARCH
2.LINEAR SEARCH
3.EXIT
Enter your choice
1
 Enter the number of  elements
3
Enter the number of array inorder
25    69      98
Enter the elemants to be searched
98
Element 98 is found at the position 3
Time taken is 1.978022 CPU1 cycles

    -----------MENU-----------
1.BINARY SEARCH
2.LINEAR SEARCH
3.EXIT
Enter your choice
1
 Enter the number of  elements
3
Enter the number of array inorder
98 22 46
Enter the elemants to be searched
22
Element 22 is found at the position 2
Time taken is 1.978022 CPU cycles
```

Result:

Thus the given program has been executed and verified successfully.

Ex.No:2 **Date:**	**Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt [ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m.**

Aim:

To implement a C program that searches for all occurrences of a given pattern in a text string using a simple string-matching algorithm.

Algorithm:

1. Get the input text txt[] of length n and pattern pat[] of length m.
2. Traverse the text from index i = 0 to n - m.
3. For each index i, compare the substring txt[i ... i+m-1] with pat[0 ... m-1].
4. If a match is found, print the index i.
5. Continue checking for other occurrences.

Program:

```
#include <stdio.h>
#include <string.h>
// Function to search for pattern in text
void search(char pat[], char txt[])
 {
 int m = strlen(pat);
 int n = strlen(txt);
// Loop to slide pat[] one by one
for (int i = 0; i <= n - m; i++)
{
int j;
// Check for pattern match at current position
for (j = 0; j < m; j++)
{
if (txt[i + j] != pat[j])
break;
}
 // If pattern matched completely,
 print index if (j == m)
{
printf("Pattern found at index %d\n", i);
}
}
}
 int main()
{
char txt[] = "ABABABCABABABCAB";
char pat[] = "ABABC";
printf("Text: %s\n", txt);
printf("Pattern: %s\n", pat);
printf("Occurrences of pattern:\n");
search(pat, txt);
return 0;
}
```

OUTPUT:

```
Text: ABABABCABABABCAB
Pattern: ABABC
 Occurrences of pattern:
Pattern found at index 2
Pattern found at index 9
```

RESULT:

Thus the given program has been executed and verified successfully.

Ex.No:3 **Date:**	**Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required 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.**

Aim:

To implement Insertion Sort and Heap Sort in C, measure their execution time for different values of n, and compare their performance.

Algorithm:

Insertion Sort Algorithm

1. Start from the second element (index 1).
2. Compare it with the previous elements.
3. Insert it into its correct position by shifting elements.
4. Repeat until the list is sorted.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}
// Function to perform Heap Sort
void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

// Function to generate random array
void generateArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;  // Random numbers between 0 and 999
    }
}

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

int main() {
    int n;
    clock_t start, end;
    double time_taken;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr1[n], arr2[n];
    // Generate random array
    generateArray(arr1, n);
    // Copy array for heap sort
    for (int i = 0; i < n; i++)
        arr2[i] = arr1[i];
    // Measure time for Insertion Sort
    start = clock();
    insertionSort(arr1, n);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Insertion Sort Time: %f seconds\n", time_taken);
    // Measure time for Heap Sort
    start = clock();
    heapSort(arr2, n);
```

```
end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Heap Sort Time: %f seconds\n", time_taken);
    return 0;
}
```

Output:

```
Enter number of elements: 1000
Insertion Sort Time: 0.081234 seconds
Heap Sort Time: 0.003256 seconds
```

Result:

Thus the program has been executed successfully.

Ex.No:4 Date:	**Develop a program to implement graph traversal using Breadth First Search and Depth First Search.**

Aim:

To implement Graph Traversal using BFS (Breadth-First Search) and DFS (Depth-First Search) in C.

Algorithm:

1. Breadth-First Search (BFS) Algorithm
1. Start from a given node (source).
2. Enqueue the starting node and mark it as visited.
3. Dequeue a node from the queue.
4. Visit all unvisited adjacent nodes, mark them visited, and enqueue them.
5. Repeat until the queue is empty.

2.Depth-First Search (DFS) Algorithm
1. Start from a given node (source).
2. Mark the node as visited.
3. Recursively visit all unvisited adjacent nodes.
4. Backtrack when no unvisited adjacent nodes are left.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100  // Maximum number of vertices
// Graph structure
typedef struct {
    int vertices;
    int adj[MAX][MAX]; // Adjacency matrix
} Graph;
// Queue structure for BFS
typedef struct {
    int items[MAX];
    int front, rear;
} Queue;
// Stack structure for DFS
typedef struct {
    int items[MAX];
    int top;
} Stack;
// Function to initialize the graph
void initGraph(Graph *g, int v) {
    g->vertices = v;
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++) {
            g->adj[i][j] = 0;  // Initialize adjacency matrix with 0
        }
```

```
  }
}

// Function to add an edge to the graph
void addEdge(Graph *g, int src, int dest) {
    g->adj[src][dest] = 1;
    g->adj[dest][src] = 1;  // For an undirected graph
}
// Function to initialize a queue
void initQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}
// Function to check if the queue is empty
int isQueueEmpty(Queue *q) {
    return q->front == -1;
}
// Function to enqueue an element
void enqueue(Queue *q, int value) {
    if (q->rear == MAX - 1) return;
    if (q->front == -1) q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
}
// Function to dequeue an element
int dequeue(Queue *q) {
    if (isQueueEmpty(q)) return -1;
    int item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    return item;
}
// Function for Breadth-First Search (BFS)
void BFS(Graph *g, int startVertex) {
    int visited[MAX] = {0};
    Queue q;
    initQueue(&q);
    printf("BFS Traversal: ");
    visited[startVertex] = 1;
    enqueue(&q, startVertex);
    while (!isQueueEmpty(&q)) {
        int currentVertex = dequeue(&q);
        printf("%d ", currentVertex);
        for (int i = 0; i < g->vertices; i++) {
            if (g->adj[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(&q, i);
            }
  }
    }
    printf("\n");
}

// Function to initialize a stack
void initStack(Stack *s) {
    s->top = -1;
}
// Function to check if the stack is empty
int isStackEmpty(Stack *s) {
    return s->top == -1;
}
// Function to push an element onto the stack
void push(Stack *s, int value) {
    if (s->top == MAX - 1) return;
    s->top++;
    s->items[s->top] = value;
}
// Function to pop an element from the stack
int pop(Stack *s) {
    if (isStackEmpty(s)) return -1;
    return s->items[s->top--];
}
// Function for Depth-First Search (DFS)
void DFS(Graph *g, int startVertex) {
    int visited[MAX] = {0};
    Stack s;
    initStack(&s);
    printf("DFS Traversal: ");
    push(&s, startVertex);
    while (!isStackEmpty(&s)) {
        int currentVertex = pop(&s);
        if (!visited[currentVertex]) {
            printf("%d ", currentVertex);
            visited[currentVertex] = 1;
        }
        for (int i = g->vertices - 1; i >= 0; i--) {
            if (g->adj[currentVertex][i] == 1 && !visited[i]) {
                push(&s, i);
            }
        }
    }
    printf("\n");
}
    int main() {
    int vertices, edges, src, dest, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
```

```
Graph g;
    initGraph(&g, vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    printf("Enter edges (src dest):\n");
    for (int i = 0; i < edges; i++) {
    scanf("%d %d", &src, &dest);
        addEdge(&g, src, dest);
    }
    printf("Enter the starting vertex for traversal: ");
    scanf("%d", &start);
    BFS(&g, start);
    DFS(&g, start);
    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the number of edges: 6
Enter edges (src dest):
0 1
0 2
1 3
1 4
2 4
3 4
Enter the starting vertex for traversal: 0

BFS Traversal: 0 1 2 3 4
DFS Traversal: 0 2 4 3 1
```

Result:

Thus the given program has been executed and verified successfully.

Ex.No:5 **Date:**	**From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.**

Aim:

To implement Dijkstra's Algorithm in C to find the shortest path from a source vertex to all other vertices in a weighted graph.

Algorithm (Dijkstra's Algorithm):

1. Initialize:
   - Set the source vertex distance to 0 and all other vertices to infinity.
   - Mark all vertices as unvisited.
2. Find the Minimum Distance Vertex:
   - Select the unvisited vertex with the smallest distance.
3. Update Distances:
   - Update the distances of its adjacent vertices if the current path is shorter.
4. Mark Vertex as Visited:
   - Mark the selected vertex as visited.
5. Repeat:
   - Repeat until all vertices have been visited or no more updates are possible.

Program:

```
#include <stdio.h>
 #include <limits.h>
#include <stdbool.h>
#define MAX 100
#define INF INT_MAX
// Function to find the vertex with the minimum distance value
int minDistance(int dist[], bool visited[], int vertices)
{
 int min = INF, min_index = -1;
for (int v = 0; v < vertices; v++)
{
 if (!visited[v] && dist[v] <= min)
{
min = dist[v];
min_index = v;
}
 }
 return min_index;
}
 // Function to print the shortest distances
void printSolution(int dist[], int vertices)
 {
 printf("Vertex \t Distance from Source\n");
 for (int i = 0; i < vertices; i++)
```

```
{
 printf("%d \t %d\n", i, dist[i]);
}
 }
 // Function to implement Dijkstra's algorithm
void dijkstra(int graph[MAX][MAX], int vertices, int src)
{
 int dist[MAX];
 // Stores shortest distances from src to i bool visited[MAX] = {false};
// Track visited vertices
// Initialize all distances as infinite and visited as false
for (int i = 0; i < vertices; i++)
{
 dist[i] = INF;
}
 dist[src] = 0;
 // Distance to itself is 0
 // Find shortest path for all vertices
 for (int count = 0; count < vertices - 1; count++)
{
int u = minDistance(dist, visited, vertices);
if (u == -1) break;
// If no vertex is found, break the loop visited[u] = true;
// Mark vertex as processed
// Update distance of adjacent vertices
for (int v = 0; v < vertices; v++)
{
 if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v])
 {
 dist[v] = dist[u] + graph[u][v];
}

}
 }
 // Print the shortest distances printSolution(dist, vertices);
}
 int main()
 {
int vertices, src;
int graph[MAX][MAX];
printf("Enter the number of vertices: ");
scanf("%d", &vertices);
printf("Enter the adjacency matrix (use 0 for no edge, or the weight of the edge if it exis
for (int i = 0; i < vertices; i++)
{
 for (int j = 0; j < vertices; j++)
{
 scanf("%d", &graph[i][j]); if (i != j && graph[i][j] == 0)
{
 graph[i][j] = INF;
 // Set no connection to INF
}
```

```
 }
 }
 printf("Enter the source vertex: ");
scanf("%d", &src); dijkstra(graph, vertices, src);
return 0;
}
```

Output :

```
Enter the number of vertices: 5
Enter the adjacency matrix:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Enter the source vertex: 0

Vertex   Distance from Source
0        0
1        10
2        50
3        30
4        60
```

Result :

Thus the program has been executed successfully.

Ex.No:6 Date:	**Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.**

Aim:

To find the **Minimum Cost Spanning Tree (MST)** of a given **undirected weighted graph** using **Prim's Algorithm**.

Algorithm (Prim's Algorithm):

Step 1: Start with an empty Minimum Spanning Tree (MST).
Choose any **arbitrary node** as the starting node and mark it as part of the MST.
Step 2: Repeat until all nodes are included in the MST:

- Select the **edge with the smallest weight** that connects a node **inside** the MST to a node **outside** the MST.
- Add this edge to the MST.
- Mark the newly included node as part of the MST.

Step 3: Stop when all nodes are included in the MST.

Program:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 5  // Number of vertices
// Function to find the vertex with minimum key value
int minKey(int key[], bool mstSet[]) {
    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;
}
// Function to print the constructed MST
void printMST(int parent[], int graph[V][V]) {
    printf("Edge   Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d    %d\n", parent[i], i, graph[i][parent[i]]);
}
// Function to construct and print MST using Prim's Algorithm
void primMST(int graph[V][V]) {
    int parent[V];     // Array to store MST
    int key[V];        // Key values used to pick minimum weight edge
    bool mstSet[V];    // To represent the set of vertices included in MST
    // Initialize all keys as INFINITE and mstSet as false
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
        key[0] = 0;        // Start from the first vertex
        parent[0] = -1;    // First node is always root of MST
for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);  // Pick the minimum key vertex
        mstSet[u] = true;             // Include in MST
        // Update key values of adjacent vertices
        for (int v = 0; v < V; 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);
}
    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}
    };
    // Function call
    primMST(graph);
    return 0;
}
```

Output:

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

Result:

Thus the program has been executed successfully.

Ex.No:7 Date:	**Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.**

Aim:

To develop a C program to find the maximum and minimum numbers in a given list of nnn numbers using the Divide and Conquer technique.

Algorithm:

Step 1: If the list contains only one element, return that element as both the maximum and minimum.
Step 2: If the list contains two elements,
- Compare them and return the larger as maximum and the smaller as minimum.

Step 3: Otherwise,
- Divide the list into two halves.
- Recursively find the maximum and minimum in both halves.
- Merge the results:
  - The overall maximum is the greater of the two maximums.
  - The overall minimum is the smaller of the two minimums.

Step 4: Return the final maximum and minimum values.

Program:

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

// Structure to store min and max values
struct MinMax {
    int min;
    int max;
};

// Function to find min and max using Divide and Conquer
struct MinMax findMinMax(int arr[], int low, int high) {
    struct MinMax result, left, right;
    // If only one element
    if (low == high) {
        result.min = result.max = arr[low];
        return result;
    }
    // If two elements
    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            result.min = arr[low];
            result.max = arr[high];
        } else {
            result.min = arr[high];
            result.max = arr[low];
        }
```

```
  return result;
    }
    // Divide: Find the middle index
    int mid = (low + high) / 2;
    // Recursively find min and max in left and right halves
    left = findMinMax(arr, low, mid);
    right = findMinMax(arr, mid + 1, high);
    // Conquer: Merge results
    result.min = (left.min < right.min) ? left.min : right.min;
    result.max = (left.max > right.max) ? left.max : right.max;
    return result;
}
int main() {
    int arr[] = {7, 2, 10, 5, 1, 8, 6, 3, 9, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    struct MinMax result = findMinMax(arr, 0, n - 1);
    printf("Minimum element: %d\n", result.min);
    printf("Maximum element: %d\n", result.max);
    return 0;
}
```

Output:

```
Minimum element: 1
Maximum element: 10
```

Result :

Thus the program has been executed successfully.

Ex.No:8 **Date:**	Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. 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.

Aim :

To implement Merge Sort and Quick Sort algorithms in C, measure their execution times for different values of nnn (number of elements), and plot a graph of time taken versus nnn.

Algorithm:

Merge Sort Algorithm:

1. If the array has only one element, return.
2. Divide the array into two halves.
3. Recursively sort both halves.
4. Merge the sorted halves.

Quick Sort Algorithm:

1. Select a pivot element.
2. Partition the array into two halves (elements less than pivot and elements greater than pivot).
3. Recursively sort both halves.

Program:

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

// Function to merge two subarrays
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[n1], R[n2];
    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++;
    }
}
// Merge Sort function
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);
    }
}

// Partition function for Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
// Function to generate an array with random values
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Random numbers between 0 and 9999
    }
}

// Function to copy array
void copyArray(int source[], int destination[], int n) {
    for (int i = 0; i < n; i++) {
        destination[i] = source[i];
    }
}

    int main() {
    int n_values[] = {1000, 5000, 10000, 20000, 50000}; // Different values of n
    int num_cases = sizeof(n_values) / sizeof(n_values[0]);
    printf("n\tMerge Sort Time(ms)\tQuick Sort Time(ms)\n");
    for (int i = 0; i < num_cases; i++) {
        int n = n_values[i];
        int arr[n], arr_copy[n];
        generateRandomArray(arr, n);
        copyArray(arr, arr_copy, n);

        clock_t start, end;
        double time_taken;

        // Merge Sort timing
        start = clock();
        mergeSort(arr, 0, n - 1);
        end = clock();
        time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
        printf("%d\t%.2f\t\t", n, time_taken);

        // Quick Sort timing
        start = clock();
        quickSort(arr_copy, 0, n - 1);
        end = clock();
        time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
        printf("%.2f\n", time_taken);
    }

    return 0;
}
```

Output:

```
n       Merge Sort Time(ms)   Quick Sort Time(ms)
1000    1.50                  0.90
5000    7.80                  4.30
10000   16.40                 8.50
20000   35.60                 19.20
50000   92.30                 48.60
```

Result :

Thus the program has been executed successfully.

Ex.No:9 **Date:**	**Implement Floyd’s algorithm for the All-Pairs- Shortest-Paths problem.**

Aim:

To implement Floyd’s Algorithm for the All-Pairs Shortest Paths problem in C and find the shortest path between all pairs of vertices in a weighted graph.

Algorithm:

Floyd’s algorithm (also called Floyd-Warshall algorithm) is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph.
Steps:

1. Initialize the distance matrix:
    - If there is a direct edge between vertices iii and jjj, set the distance as the weight of the edge.
    - If there is no direct edge, set the distance to **infinity**.
    - Distance from a vertex to itself is **0**.
2. Update the distance matrix using each vertex kkk as an intermediate vertex:
    - For each pair of vertices (i,j)(i, j)(i,j), update: dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])
    - This checks whether the path from iii to jjj via kkk is shorter than the previously known path.
3. Repeat the process for all vertices.

Program:

```
#include <stdio.h>
#define INF 99999 // Representing infinity
#define V 4       // Number of vertices in the graph

// Function to print the solution matrix
void printSolution(int dist[][V]) {
    printf("The shortest distance matrix:\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");
    }
}

// Function to implement Floyd's Algorithm
void floydWarshall(int graph[][V]) {
    int dist[V][V];
    // Initialize distance matrix with input graph weights
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update the distance matrix using Floyd's algorithm
    for (int k = 0; k < V; k++) { // Consider each vertex as an intermediate
        for (int i = 0; i < V; i++) { // Iterate over each row
            for (int j = 0; j < V; j++) { // Iterate over each column
                // Update dist[i][j] if a shorter path exists via vertex k
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the shortest path matrix
    printSolution(dist);
}

int main() {
    int graph[V][V] = {
        {0, 3, INF, 7},
        {8, 0, 2, INF},
        {5, INF, 0, 1},
        {2, INF, INF, 0}
    };

    floydWarshall(graph);
    return 0;
}
```

Output:

```
The shortest distance matrix:
      0      3      5      6
      5      0      2      3
      3      6      0      1
      2      5      7      0
```

Result :

Thus the program has been executed successfully.

CONTENT BEYOND THE SYLLABUS

Ex.No:10 **Date:**	**Perform the Towers of Hanoi Problem to transfer the disk from one tower to another.**

Aim :

To implement the **Towers of Hanoi** problem in C to transfer disks from one tower (source) to another (destination) using an auxiliary tower, following the rules of the problem.

Algorithm:

The **Towers of Hanoi** problem involves three towers (pegs) and nnn disks. The goal is to move all the disks from the **source peg** to the **destination peg**, using an **auxiliary peg** while following these rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. A disk can only be moved if it is the topmost disk on a peg.

Recursive Algorithm:

1. Move **n−1n-1n−1** disks from **source** to **auxiliary** peg.
2. Move the **nthn^{th}nth** (largest) disk from **source** to **destination** peg.
3. Move the **n−1n-1n−1** disks from **auxiliary** to **destination** peg.

Program:

```
#include <stdio.h>

// Function to solve Towers of Hanoi
void towersOfHanoi(int n, char source, char auxiliary, char destination) {
   if (n == 1) {
      printf("Move disk 1 from %c to %c\n", source, destination);
      return;
   }

   // Move n-1 disks from source to auxiliary using destination
   towersOfHanoi(n - 1, source, destination, auxiliary);

   // Move the nth (largest) disk from source to destination
   printf("Move disk %d from %c to %c\n", n, source, destination);

   // Move n-1 disks from auxiliary to destination using source
   towersOfHanoi(n - 1, auxiliary, source, destination);
}


int main() {
   int n; // Number of disks
   printf("Enter the number of disks: ");
   scanf("%d", &n);
```

```
    // Calling the function with A as source, B as auxiliary, and C as destination
    printf("The sequence of moves are:\n");
    towersOfHanoi(n, 'A', 'B', 'C');

    return 0;
}
```

Output:

```
Enter the number of disks: 3
The sequence of moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Result :

Thus the program has been executed successfully.

Ex.No:11 **Date:**	**Implement the Red-Black Tree with different operation Process.**

Aim :

To implement a Red-Black Tree (RBT) in C with insertion, deletion, and search operations while maintaining its balanced property.

Algorithm:

A Red-Black Tree is a self-balancing binary search tree with the following properties:

1. Every node is either red or black.
2. The root node is always black.
3. No two consecutive red nodes exist (i.e., a red node cannot have a red parent).
4. Every path from a node to its NULL children must have the same number of black nodes.
5. Newly inserted nodes are initially red.

Basic Operations:

1. Insertion:
   - Insert a node like in a Binary Search Tree (BST) and color it red.
   - If the parent is also red, perform rotation and recoloring to maintain balance.
2. Rotation:
   - Left Rotation: When a node is inserted into the right child of a right-heavy node.
   - Right Rotation: When a node is inserted into the left child of a left-heavy node.
3. Deletion:
   - Remove the node as in BST deletion.
   - If the deleted node or its replacement is black, fix violations using rotations and recoloring.

Program:

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

// Define node structure
struct Node {
   int data;
   char color;
   struct Node *left, *right, *parent;
};

// Helper function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->color = 'R';  // New nodes are always red
   newNode->left = newNode->right = newNode->parent = NULL;
   return newNode;
}
```

```
// Left Rotate function
void leftRotate(struct Node** root, struct Node* x) {
    struct Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;

    y->parent = x->parent;

    if (x->parent == NULL)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    y->left = x;
    x->parent = y;
}

// Right Rotate function
void rightRotate(struct Node** root, struct Node* y) {
    struct Node* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;

    x->parent = y->parent;

    if (y->parent == NULL)
        *root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    x->right = y;
    y->parent = x;
}

// Fix violations after insertion
void fixInsert(struct Node** root, struct Node* z) {
    while (z->parent != NULL && z->parent->color == 'R') {
        struct Node* grandparent = z->parent->parent;

        if (z->parent == grandparent->left) {
            struct Node* uncle = grandparent->right;
```

```
 if (uncle != NULL && uncle->color == 'R') { // Case 1: Uncle is red
                z->parent->color = 'B';
                uncle->color = 'B';
                grandparent->color = 'R';
                z = grandparent;
            } else {
                if (z == z->parent->right) { // Case 2: z is right child
                    z = z->parent;
                    leftRotate(root, z);
                }
                // Case 3: z is left child
                z->parent->color = 'B';
                grandparent->color = 'R';
                rightRotate(root, grandparent);
            }
        } else { // Mirror cases for right subtree
            struct Node* uncle = grandparent->left;

            if (uncle != NULL && uncle->color == 'R') {
                z->parent->color = 'B';
                uncle->color = 'B';
                grandparent->color = 'R';
                z = grandparent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(root, z);
                }
                z->parent->color = 'B';
                grandparent->color = 'R';
                leftRotate(root, grandparent);
            }
        }
    }
    (*root)->color = 'B'; // Ensure root is always black
}

// Insert a node into Red-Black Tree
void insert(struct Node** root, int data) {
    struct Node* z = createNode(data);
    struct Node* y = NULL;
    struct Node* x = *root;

    while (x != NULL) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if (y == NULL)
        *root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    fixInsert(root, z);
}

// In-order traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d(%c) ", root->data, root->color);
    inorder(root->right);
}

int main() {
    struct Node* root = NULL;

    int choice, value;
    while (1) {
        printf("\nRed-Black Tree Operations:\n");
        printf("1. Insert\n2. Inorder Traversal\n3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(&root, value);
                break;
            case 2:
                printf("In-order Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```

Output:

```
Red-Black Tree Operations:
1. Insert
2. Inorder Traversal
3. Exit
Enter choice: 1
Enter value to insert: 20

Enter choice: 1
Enter value to insert: 15

Enter choice: 1
Enter value to insert: 25

Enter choice: 2
In-order Traversal: 15(R) 20(B) 25(R)
```

Result:

Thus the program has been executed successfully.