DESIGN AND ANALYSIS OF ALGORITHMS LAB RECORD

Roll No	**25MSCSCKK0001**
Name	**JIDHUN.P.P**
Class	**MSC. COMPUTER SCIENCE**

Department of Computer Science
School of Engineering & Technology
Pondicherry University
Karaikal -609605, Puducherry, India

Department of Compute Science School of Engineering &Technology Pondicherry University Karaikal-609 605, Puducherry, India

BONAFIDE CERTIFICATE

Name: ____________________

Reg. No: ____________________**Class:** __________

Course Title: ____________________

Course Code: ____________________

Certified that this is the bonafide record of work done by me during **Odd / Even** Semester of **2025 – 2027** and submitted for the Practical Examination on:

Date: ____________________

Staff In-Charge **Head of the Department**

Examiners:

1. ____________________

2. ____________________

Index of Lab Programs/Exercises

Exercise 1: Write a program that implements Binary Search

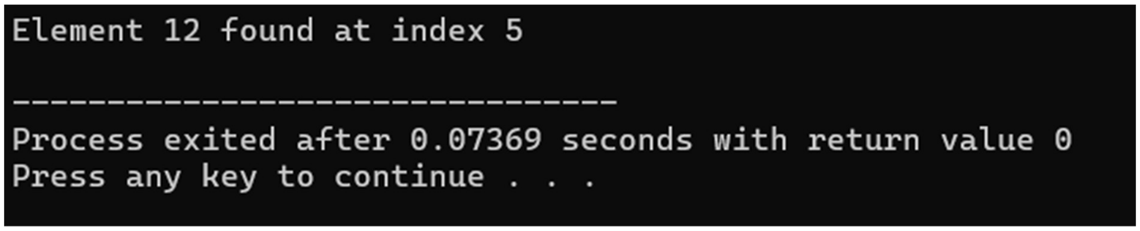
Aim:

Algorithm:

Code Section:

```c
#include <stdio.h>

// Recursive binary search function
int binarySearchRecursive(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if the target is present at the middle
        if (arr[mid] == target)
            return mid;

        // If the target is greater, search in the right half
        if (arr[mid] < target)
            return binarySearchRecursive(arr, mid + 1, right, target);

        // If the target is smaller, search in the left half
        return binarySearchRecursive(arr, left, mid - 1, target);
    }

    // If the target is not present in the array
    return -1;
}

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 12;

    int result = binarySearchRecursive(arr, 0, size - 1, target);

    if (result != -1)
        printf("Element %d found at index %d\n", target, result);
    else
        printf("Element %d not found in the array\n", target);

    return 0;
}
```

Output Section:



```
Element 12 found at index 5

--------------------------------
Process exited after 0.07369 seconds with return value 0
Press any key to continue . . .
```

Exercise 2: Write a program that implements Quick Sort

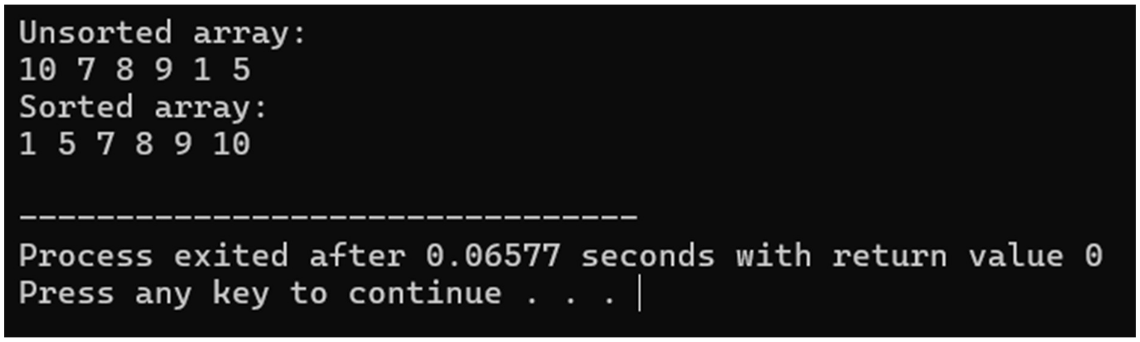
Aim:

Algorithm:

Code Section:

```c
#include <stdio.h>

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

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choosing the pivot element
    int i = (low - 1);      // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++;  // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function that implements Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

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

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

// Main function to test the Quick Sort implementation
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output Section:



```
Unsorted array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10

--------------------------------
Process exited after 0.06577 seconds with return value 0
Press any key to continue . . .
```

Exercise 3: Write a program that implements Strassen's matrix multiplication

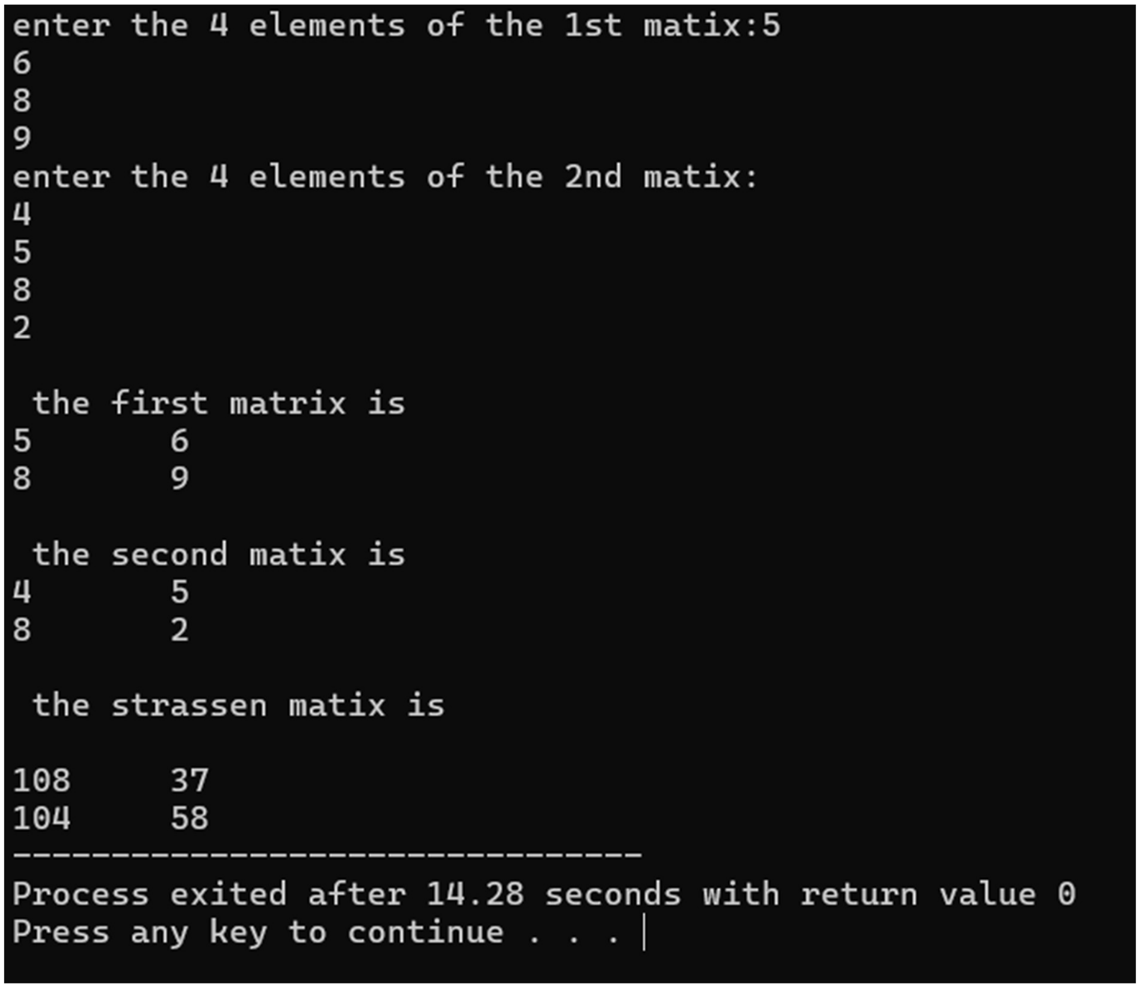
Aim:

Algorithm:

Code Section:

```
#include<stdio.h>
int main() {
      int a[2][2],b[2][2],c[2][2],i,j;
      int m1,m2,m3,m4,m5,m6,m7;
      printf("enter the 4 elements of the 1st matix:");
        for(i=0;i<2;i++){
                     for(j=0;j<2;j++){
            scanf("%d",&a[i][j]);
          }
      }
      printf("enter the 4 elements of the 2nd matix:\n");
      for(i=0;i<2;i++){
         for(j=0;j<2;j++){
                scanf("%d",&b[i][j]);
          }
      }
      printf("\n the first matrix is \n");
        for(i=0;i<2;i++){
            for(j=0;j<2;j++){
            printf("%d\t",a[i][j]);
             }
            printf("\n");
      }
               printf("\n the second matix is \n");
      for(i=0;i<2;i++){
              for(j=0;j<2;j++){
            printf("%d\t",b[i][j]);
            }
            printf("\n");
      }

       m1=(a[0][0]+a[1][1])*(b[0][0]+b[1][1]);
       m2=(a[1][0]+a[1][1])*b[0][0];
       m3=a[0][0]*(b[0][1]-b[1][1]);
       m4=a[1][1]*(b[1][0]-b[0][0]);
       m5=(a[0][0]+a[0][1])*b[1][1];
       m6=(a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
       m7=(a[0][1]-a[0][0])*(b[1][0]+b[1][1]);

       c[0][0]=m1+m4-m5+m7;
       c[0][1]=m3+m5;
       c[1][0]=m2+m4;
       c[1][1]=m1-m2+m3+m6;

      printf("\n the strassen matix is \n");
      for(i=0;i<2;i++){
          printf("\n");
               for(j=0;j<2;j++){
            printf("%d\t",c[i][j]);
            }
      }
      return 0;
}
```

Output Section:



```
enter the 4 elements of the 1st matix:5
6
8
9
enter the 4 elements of the 2nd matix:
4
5
8
2

 the first matrix is
5       6
8       9

 the second matix is
4       5
8       2

 the strassen matix is

108     37
104     58
--------------------------------
Process exited after 14.28 seconds with return value 0
Press any key to continue . . .
```

Exercise 4: Write a program that implements Prim’s Algorithm

Aim:

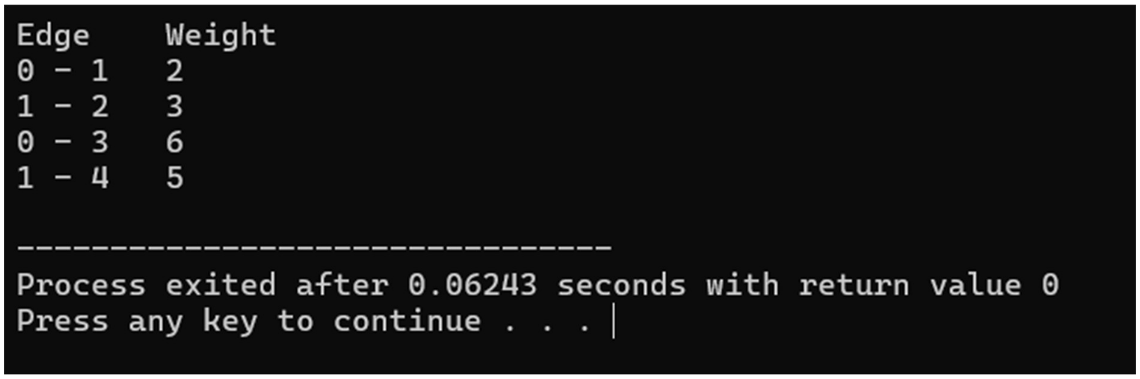
Algorithm:

Code Section:

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

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

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Function to compare edges for qsort
int compare(const void *a, const void *b) {
    return ((struct Edge *)a)->weight - ((struct Edge *)b)->weight;
}

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

// Function to implement Prim's algorithm using an edge list
void primMST(struct Edge edges[], int edgeCount) {
    int parent[V]; // Array to store the constructed MST
    int key[V];    // Key values used to pick the minimum weight edge
    int mstSet[V]; // To represent the set of vertices included in the MST

    // Initialize all keys as infinite and mstSet as false
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    // Always include the first vertex in MST
    key[0] = 0;      // Make the key 0 so that this vertex is picked first
    parent[0] = -1;  // First node is always the root of MST

    // 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] = 1;

        // Update the key value and parent index of the adjacent vertices of
the picked vertex
```

```
        for (int i = 0; i < edgeCount; i++) {
            // Check if the edge connects to the picked vertex
            if (edges[i].src == u && mstSet[edges[i].dest] == 0) {
                if (edges[i].weight < key[edges[i].dest]) {
                    parent[edges[i].dest] = u;
                    key[edges[i].dest] = edges[i].weight;
                }
            } else if (edges[i].dest == u && mstSet[edges[i].src] == 0) {
                if (edges[i].weight < key[edges[i].src]) {
                    parent[edges[i].src] = u;
                    key[edges[i].src] = edges[i].weight;
                }
            }
        }
    }

    // Print the constructed MST
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d\n", parent[i], i, key[i]);
}

int main() {
    // Example graph represented as an edge list
    struct Edge edges[] = {
        {0, 1, 2},
        {0, 3, 6},
        {1, 2, 3},
        {1, 3, 8},
        {1, 4, 5},
        {2, 4, 7},
        {3, 4, 9}
    };

    int edgeCount = sizeof(edges) / sizeof(edges[0]);

    // Call the Prim's algorithm function
    primMST(edges, edgeCount);

    return 0;
}
```

Output Section:



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

--------------------------------
Process exited after 0.06243 seconds with return value 0
Press any key to continue . . .
```

Exercise 5: Write a program that implements Kruskal’s Algorithm

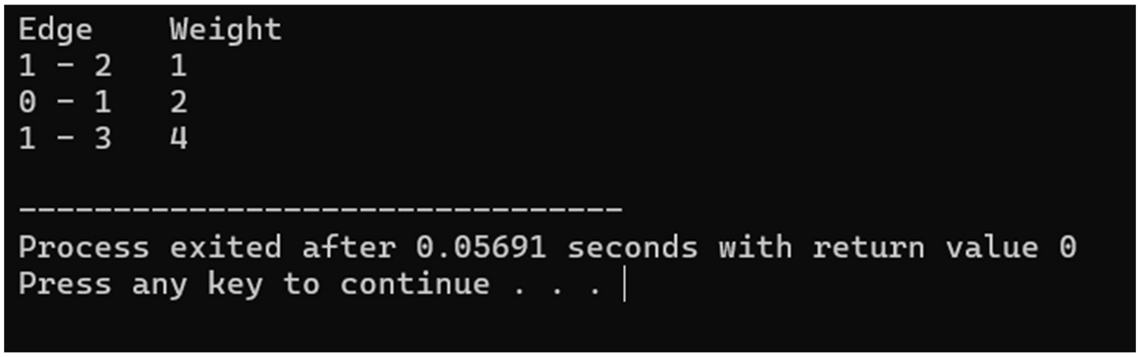
Aim:

Algorithm:

Code Section:

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

#define MAX_VERTICES 100

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edges;
};

struct Subset {
    int parent, rank;
};

// Function to create a graph
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int index, int src, int dest, int weight) {
    graph->edges[index].src = src;
    graph->edges[index].dest = dest;
    graph->edges[index].weight = weight;
}

// Function to find the subset of an element i (uses path compression
technique)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Function to do union of two subsets x and y (uses union by rank)
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Union by rank
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Comparison function for qsort to sort edges by weight
int compareEdges(const void* a, const void* b) {
    struct Edge* edgeA = (struct Edge*)a;
    struct Edge* edgeB = (struct Edge*)b;
    return edgeA->weight - edgeB->weight;
}

// Function to perform Kruskal's algorithm to find MST
void kruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V];  // Array to store the result MST
    int e = 0;  // Count of edges in MST
    int i = 0;  // Index variable to traverse edges

    // Step 1: Sort all edges in non-decreasing order of their weight
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);

    // Allocate memory for creating V subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct
Subset));

    // Step 2: Create V subsets with single elements
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Step 3: Process edges, adding them to the MST if they don't form a
cycle
    while (e < V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause a cycle, include it in the
result
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Step 4: Print the constructed MST
    printf("Edge \tWeight\n");
    for (int i = 0; i < e; i++) {
        printf("%d - %d \t%d\n", result[i].src, result[i].dest,
result[i].weight);
    }

    // Free the allocated memory for subsets
    free(subsets);
}

// Main function
int main() {
    int V = 4;  // Number of vertices
    int E = 5;  // Number of edges

    struct Graph* graph = createGraph(V, E);

    // Adding edges to the graph
    addEdge(graph, 0, 0, 1, 2);
    addEdge(graph, 1, 0, 2, 3);
    addEdge(graph, 2, 1, 2, 1);
    addEdge(graph, 3, 1, 3, 4);
    addEdge(graph, 4, 2, 3, 5);

    // Perform Kruskal's algorithm to find the MST
    kruskalMST(graph);

    // Free allocated memory
    free(graph->edges);
    free(graph);
return 0;
}
```

Output Section:



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

--------------------------------
Process exited after 0.05691 seconds with return value 0
Press any key to continue . . .
```

Exercise 6: Write a program that implements All pair shortest path problem

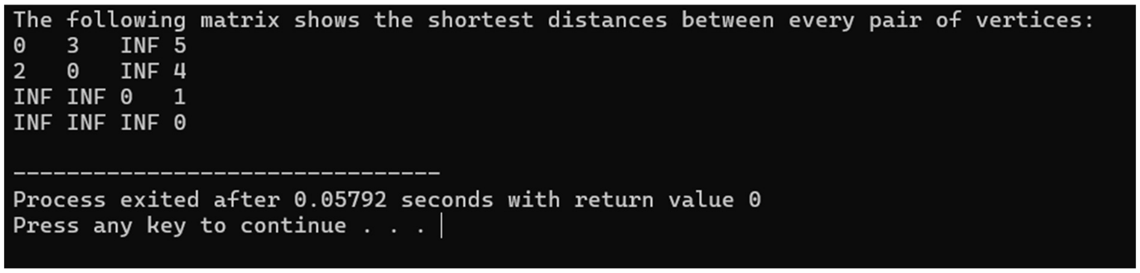
Aim:

Algorithm:

Code Section:

```
#include <stdio.h>
#include <limits.h>  // Include the limits.h header for INT_MAX

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

// Function to print the solution matrix
void printSolution(int dist[][V]);

// Implementation of the Floyd-Warshall algorithm
void floydWarshall(int graph[][V]) {
    int dist[V][V];

    // Initialize the solution matrix with the same values as the input graph
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update the solution matrix by considering all vertices as intermediate
vertices
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                // If vertex k is on the shortest path from i to j, update
the value
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX &&
dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the solution
    printSolution(dist);
}

// Function to print the solution matrix
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] == INT_MAX)
                printf("INF ");
            else
                printf("%d   ", dist[i][j]);
        }
        printf("\n");
    }
}

int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 3, INT_MAX, 5},
        {2, 0, INT_MAX, 4},
        {INT_MAX, INT_MAX, 0, 1},
        {INT_MAX, INT_MAX, INT_MAX, 0}
    };

    // Apply the Floyd-Warshall algorithm
    floydWarshall(graph);

    return 0;
}
```

Output Section:



```
The following matrix shows the shortest distances between every pair of vertices:
0   3   INF 5
2   0   INF 4
INF INF 0   1
INF INF INF 0

--------------------------------
Process exited after 0.05792 seconds with return value 0
Press any key to continue . . .
```

Exercise 7: Write a program that implements N-Queen Problem

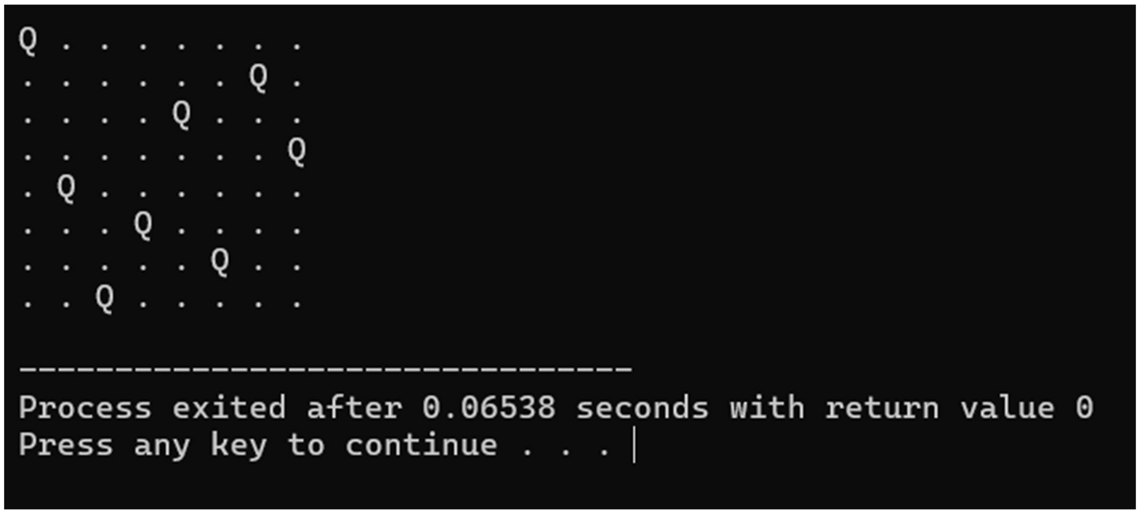
Aim:

Algorithm:

Code Section:

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

#define N 8  // Define the board size (e.g., 8 for an 8x8 board)

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%s ", board[i][j] ? "Q" : ".");
        }
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check the row on the left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

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

    // Check the lower diagonal on the left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col) {
    // If all queens are placed, return true
    if (col >= N)
        return true;

    // Try placing a queen in all rows one by one
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;  // Place the queen

            // Recursively place the queen in the next column
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in current position doesn't lead to a
solution,
            // remove queen (backtrack)
            board[i][col] = 0;
        }
    }
    // If the queen cannot be placed in any row, return false
    return false;
}

bool solveNQ() {
    int board[N][N] = {0};  // Initialize the chessboard with 0s

    // Start solving from the first column
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist\n");
        return false;
    }

    // If a solution is found, print it
    printSolution(board);
    return true;
}

int main() {
    solveNQ();
    return 0;
}
```

Output Section:



```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

--------------------------------
Process exited after 0.06538 seconds with return value 0
Press any key to continue . . .
```

Exercise 8: Write a program that implements Heapsort

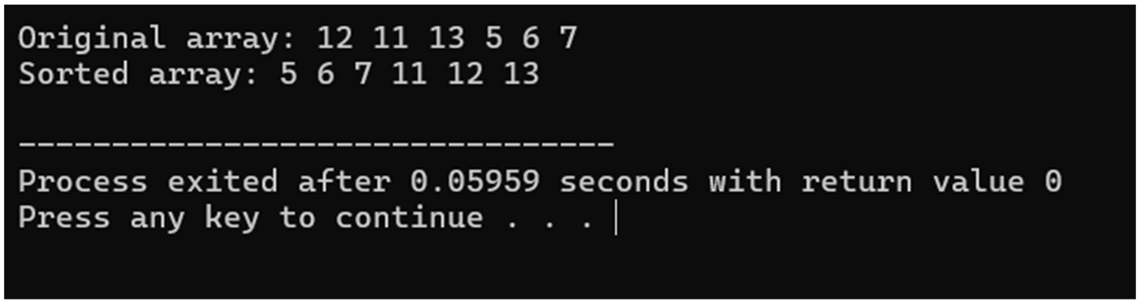
Aim:

Algorithm:

Code Section:

```
#include <stdio.h>

// Function to heapify a subtree rooted at node i of the array arr[] of size
n
void heapify(int arr[], int n, int i) {
    int largest = i;     // Initialize largest as root
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        // Swap the root with the largest element
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform heapsort on an array
void heapsort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from the heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

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

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    heapsort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output Section:



```
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13

--------------------------------
Process exited after 0.05959 seconds with return value 0
Press any key to continue . . .
```

Exercise 9: Write a program that implements Travelling Salesperson Problem

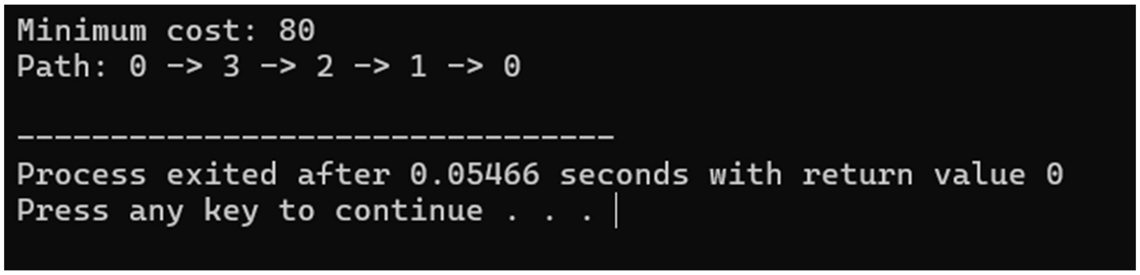
Aim:

Algorithm:

Code Section:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 4  // Number of vertices in the graph

int tsp(int graph[][V], int path[], bool visited[], int pos, int n, int
count, int cost, int ans) {
    if (count == n && graph[pos][0]) {
        if (cost + graph[pos][0] < ans) {
            path[count] = 0;
            return cost + graph[pos][0];
        }
        return ans;
    }

    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[pos][i]) {
            visited[i] = true;
            path[count] = i;
            ans = tsp(graph, path, visited, i, n, count + 1, cost +
graph[pos][i], ans);
            visited[i] = false;
        }
    }
    return ans;
}

void printPath(int path[], int n) {
    printf("Path: ");
    for (int i = 0; i < n; i++) {
        printf("%d -> ", path[i]);
    }
    printf("0\n");
}

int main() {
    int graph[V][V] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int path[V + 1];
    bool visited[V] = {false};
    visited[0] = true;
    path[0] = 0;

    int ans = tsp(graph, path, visited, 0, V, 1, 0, INT_MAX);

    printf("Minimum cost: %d\n", ans);
    printPath(path, V);

    return 0;
}
```

Output Section:



```
Minimum cost: 80
Path: 0 -> 3 -> 2 -> 1 -> 0

--------------------------------
Process exited after 0.05466 seconds with return value 0
Press any key to continue . . .
```

Exercise 10: Write a program that implements Knapsack using greedy Method

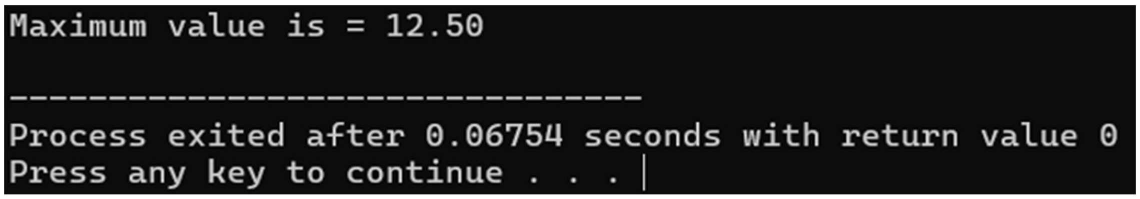
Aim:

Algorithm:

Code Section:

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

typedef struct {
    int weight;
    int value;
} item;

int compare(const void *a, const void *b) {
    double r1 = (double)((item*)a)->value / (double)((item*)a)->weight;
    double r2 = (double)((item*)b)->value / (double)((item*)b)->weight;
    if (r1 < r2) return 1;
    if (r1 > r2) return -1;
    return 0;
}

double knapsack(item items[], int n, int w) {
    qsort(items, n, sizeof(item), compare);

    int current_weight = 0;
    double final_value = 0.0;

    for (int i = 0; i < n; i++) {
        if (current_weight + items[i].weight <= w) {
            current_weight += items[i].weight;
            final_value += items[i].value;
        } else {
            int remain = w - current_weight;
            final_value += items[i].value * ((double)remain /
items[i].weight);
            break;
        }
    }

    return final_value;
}

int main() {
    int w = 50;
    item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int n = sizeof(items) / sizeof(items[0]);

    double maxvalue = knapsack(items, n, w);
    printf("Maximum value is = %.2f\n", maxvalue);

    return 0;
}
```

Output Section:



```
Maximum value is = 12.50

--------------------------------
Process exited after 0.06754 seconds with return value 0
Press any key to continue . . .
```