SONA COLLEGE OF ARTS AND SCIENCE
Affiliated to Periyar University, Salem -11

DEPARTMENT OF COMPUTER SCIENCE
(CYBER SECURITY)

Practical Record

Name :

Register Number :

Subject Code :

Subject Title :

Year / Sem :

ACADEMIC YEAR: 2023 – 2024

SONA COLLEGE OF ARTS AND SCIENCE
Affiliated to Periyar University, Salem

Certificate

This is to Certify that the Practical Record **“Practical II – Data Structures And Algorithms Lab”** is a bonafide work done by Name & Reg. No

____

submitted to the Department of ComputerScience (Cyber Security), during the academic year 2023 – 2024.

SUBJECT IN-CHARGE **HEAD OF THE DEPARTMENT**

Submitted for University Practical Examination held on ____________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

Ex. No.: 01	**PERFORM STACK OPERATIONS**
Date	

AIM

To Perform Stack Operations

PROCEDURE

Step 1: Start the Process

Step 2: **Initialize Stack:**

Create an array to represent the stack and initialize the top of the stack to -1.

Step 3: **Push Operation:**

Increment the top of the stack and add the given element to the stack.

Step 4: **Pop Operation:**

Remove the element from the top of the stack and decrement the top.

Step 5: **Display Operation:**

Display all elements in the stack.

Step 6: Stop the Process

CODING

```
#include <stdio.h>
#define MAX_SIZE 10
// Function prototypes
void push(int element);
void pop();
void display();

// Global variables
int stack[MAX_SIZE];
int top = -1;
int main() {
    int choice, element;
    do {
        // Display menu
        printf("\nStack Operations Menu:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        // Perform operations based on user choice
        switch (choice) {
            case 1:
                printf("Enter the element to push: ");
                scanf("%d", &element);
                push(element);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);
    return 0;
}
```

```c
// Function to push an element onto the stack
void push(int element) {
    if (top == MAX_SIZE - 1) {
        printf("Stack overflow. Cannot push element.\n");
    } else {
        top++;
        stack[top] = element;
        printf("%d pushed to the stack.\n", element);
    }
}

// Function to pop an element from the stack
void pop() {
    if (top == -1) {
        printf("Stack underflow. Cannot pop element.\n");
    } else {
        printf("%d popped from the stack.\n", stack[top]);
        top--;
    }
}

// Function to display elements of the stack
void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Elements in the stack are:\n");
        for (int i = 0; i <= top; i++) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
```

OUTPUT

```
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the element to push: 5
5 pushed to the stack.

Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the element to push: 10
10 pushed to the stack.

Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Elements in the stack are:
5 10

Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
10 popped from the stack.

Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Elements in the stack are:
```

5

Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting the program.

RESULT

Thus the above perform stack operations program has been created and executed successfully

Ex. No.: 02	**PERFORM QUEUE OPERATIONS**
Date	

AIM

To Perform Queue Operations

PROCEDURE

Step 1: Start the process

Step 2: **Initialize Queue**

i. Create an array to represent the queue.

ii. Initialize front and rear pointers to -1, indicating an empty queue.

Step 3: **Enqueue Operation**

i. Increment the rear pointer.

ii. Add the given element to the queue at the position indicated by the rear pointer.

iii. If it's the first element, update the front pointer.

Step 4: **Dequeue Operation**

i. Remove the element from the front of the queue.

ii. Increment the front pointer.

iii. If the last element is dequeued, reset both front and rear pointers.

Step 5: **Display Operation**

i. Display all elements in the queue, starting from the front and ending at the rear.

Step 6: Stop the process

CODING

```
#include <stdio.h>
#define MAX_SIZE 10
// Function prototypes
void enqueue(int element);
void dequeue();
void display();

// Global variables
int queue[MAX_SIZE];
int front = -1, rear = -1;

int main() {
    int choice, element;

    do {
        // Display menu
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Perform operations based on user choice
        switch (choice) {
            case 1:
                printf("Enter the element to enqueue: ");
                scanf("%d", &element);
                enqueue(element);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
```

```
    } while (choice != 4);

    return 0;
}

// Function to enqueue an element
void enqueue(int element) {
    if (rear == MAX_SIZE - 1) {
        printf("Queue overflow. Cannot enqueue element.\n");
    } else {
        if (front == -1) {
            front = 0;
        }
        rear++;
        queue[rear] = element;
        printf("%d enqueued to the queue.\n", element);
    }
}

// Function to dequeue an element
void dequeue() {
    if (front == -1) {
        printf("Queue underflow. Cannot dequeue element.\n");
    } else {
        printf("%d dequeued from the queue.\n", queue[front]);
        if (front == rear) {
            front = rear = -1;
        } else {
            front++;
        }
    }
}

// Function to display elements of the queue
void display() {
    if (front == -1) {
        printf("Queue is empty.\n");
    } else {
        printf("Elements in the queue are:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
```

OUTPUT

```
Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the element to enqueue: 5
5 enqueued to the queue.

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the element to enqueue: 10
10 enqueued to the queue.

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Elements in the queue are:
5 10

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
5 dequeued from the queue.

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
```

Elements in the queue are:
10

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
Exiting the program.

RESULT

Thus the above perform queue operations program has been created and executed successfully

Ex. No.: 03	**PERFORM TREE TRAVERSAL OPERATIONS**
Date	

AIM

To Perform Tree traversal operations

PROCEDURE

Step 1: Start the process

Step 2: **Define Tree Node**

i. Create a structure to represent a tree node with data, left child, and right child.

Step 3: **Initialize Tree**

i. Create a root node and initialize it to NULL.

Step 4: Insert Nodes

i. Insert nodes into the tree to form a binary tree structure.

Step 4: **Inorder Traversal**

i. Traverse the left subtree.

ii. Visit the root node.

iii. Traverse the right subtree.

Step 5: **Preorder Traversal**

i. Visit the root node.

ii. Traverse the left subtree.

iii. Traverse the right subtree.

Step 6: **Postorder Traversal**

i. Traverse the left subtree.

ii. Traverse the right subtree.

iii. Visit the root node.

Step 7: Display the valid result

Step 8: Stop the process

CODING

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

struct TreeNode {
   int data;
   struct TreeNode *left, *right;
};

struct TreeNode *createNode(int data) {
   struct TreeNode *newNode = (struct TreeNode *)malloc(sizeof(struct TreeNode));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct TreeNode *insertNode(struct TreeNode *root, int data) {
   if (root == NULL)
      return createNode(data);

   if (data < root->data)
      root->left = insertNode(root->left, data);
   else if (data > root->data)
      root->right = insertNode(root->right, data);

   return root;
}

void inorderTraversal(struct TreeNode *root) {
   if (root != NULL) {
      inorderTraversal(root->left);
      printf("%d ", root->data);
      inorderTraversal(root->right);
   }
}

void preorderTraversal(struct TreeNode *root) {
   if (root != NULL) {
      printf("%d ", root->data);
      preorderTraversal(root->left);
      preorderTraversal(root->right);
   }
}
```

```
void postorderTraversal(struct TreeNode *root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

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

    root = insertNode(root, 20);
    insertNode(root, 10);
    insertNode(root, 30);
    insertNode(root, 5);
    insertNode(root, 15);

    printf("Inorder: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

OUTPUT

Inorder: 5 10 15 20 30

Preorder: 20 10 5 15 30

Postorder: 5 15 10 30 20

RESULT

Thus the above perform tree traversal program has been created and executed successfully

Ex. No.: 04	**SEARCH AN ELEMENT IN AN ARRAY USING LINEAR SEARCH**
Date	

AIM

To Search an element an array using linear search method.

PROCEDURE

Step 1: Start the Process

Step 2: Input the size of the array (n) and the array elements.

Step 3: Input the element to be searched (searchElement).

Step 4: Initialize found to 0 (false).

Step 5: For each element (arr[i]) in the array:

i. If arr[i] is equal to searchElement, set found to 1 (true) and break out of the loop.

ii. If found is 1:

iii. Print "Element found at index i" where i is the index of the found element.

Else:

iv. Print "Element not found in the array."

Step 6: Stop the process

CODING

```c
#include <stdio.h>

int main() {
    int n, searchElement, found = 0;

    // Input size of array
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    // Input array elements
    printf("Enter the array elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input element to be searched
    printf("Enter the element to be searched: ");
    scanf("%d", &searchElement);

    // Linear search
    for (int i = 0; i < n; i++) {
        if (arr[i] == searchElement) {
            found = 1;
            break;
        }
    }

    // Output
    if (found) {
        printf("Element found at index %d.\n", i);
    } else {
        printf("Element not found in the array.\n");
    }

    return 0;
}
```

OUTPUT

```
Enter the size of the array: 5
Enter the array elements:
10 20 30 40 50
Enter the element to be searched: 30
Element found at index 2.
```

RESULT

Thus the above Search an element an array using linear search program has been created and executed successfully

Ex. No.: 05	**SEARCH AN ELEMENT IN AN ARRAY USING BINARY SEARCH**
Date	

AIM

To Search an element in an array using binary search method

PROCEDURE

Step 1: Start the Process

Step 2: Input the size of the array (n) and the array elements (sorted in ascending order).

Step 3: Input the element to be searched (searchElement).

Step 4: Initialize low to 0, high to n-1, and found to 0 (false).

Step 5: While low is less than or equal to high:

i. Calculate the mid index as mid = (low + high) / 2.

ii. If arr[mid] is equal to searchElement, set found to 1 (true) and break out of the loop.

iii. If arr[mid] is less than searchElement, update low = mid + 1.

iv. If arr[mid] is greater than searchElement, update high = mid - 1.

If found is 1:

i. Print "Element found at index mid" where mid is the index of the found element.
ii. Else:
iii. Print "Element not found in the array."

Step 6: Stop the process

CODING

```c
#include <stdio.h>
int main() {
    int n, searchElement, found = 0;

    // Input size of array
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    // Input sorted array elements
    printf("Enter the sorted array elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input element to be searched
    printf("Enter the element to be searched: ");
    scanf("%d", &searchElement);

    // Binary search
    int low = 0, high = n - 1, mid;

    while (low <= high) {
        mid = (low + high) / 2;

        if (arr[mid] == searchElement) {
            found = 1;
            break;
        } else if (arr[mid] < searchElement) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    // Output
    if (found) {
        printf("Element found at index %d.\n", mid);
    } else {
        printf("Element not found in the array.\n");
    }
    return 0;
}
```

OUTPUT

Enter the size of the array: 6

Enter the sorted array elements:

10 20 30 40 50 60

Enter the element to be searched: 40

Element found at index 3.

RESULT

Thus the above Search an element an array using binary search program has been created and executed successfully

Ex. No.: 06	SORT THE GIVEN SET OF ELEMENTS USING MERGE SORT
Date	

AIM

To sort the sort the given set of elements using Merge sort method.

PROCEDURE

Step 1: Start the process

Step 2: If the array has one element or is empty, it is already sorted.

Step 3: Otherwise, divide the unsorted array into two halves.

Step 4: Recursively sort each half.

Step 5: Merge the sorted halves back together.

Step 6: Stop the process

CODING

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];

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

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

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);
    }
}

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 arr_size = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Given array: ");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

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

    return 0;
}
```

OUTPUT

Given array: 12 11 13 5 6 7

Sorted array: 5 6 7 11 12 13

RESULT

Thus the above sort the given set of elements using Merge sort method program has been created and executed successfully

Ex. No.: 07	SORT THE GIVEN SET OF ELEMENTS USING QUICK SORT
Date	

AIM

To sort the given set of elements using quick sort

ALGORITHM

Step 1: Start the process

Step 2: If the array has one element or is empty, it is already sorted.

Step 3: Choose a "pivot" element from the array.

Step 4: Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.

Step 5: Recursively sort the sub-arrays.

Step 6: Stop the process

CODING

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];

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

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

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);
    }
}

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 arr_size = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Given array: ");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

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

    return 0;
}
```

OUTPUT

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

RESULT

Thus the above sort the given set of elements using quick sort program has been created and executed successfully

Ex. No.: 08	**SEARCH THE K^{TH} SMALLEST ELEMENT USING SELECTION SORT**
Date	

AIM

To search the Kth smallest element using selection sort.

PROCEDURE

Step 1: Start the process

Step 2: Input the size of the array (n).

Step 3: Input array elements.

Step 4: Input the value of K.

Step 5: If K is less than 1 or greater than n, print an error message and exit.

Step 6: Perform Selection Sort to sort the array in ascending order.

Step 7: Print the Kth smallest element.

Step 8: Stop the process

CODING

```
#include <stdio.h>
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        // Swap arr[i] and arr[minIndex]
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

// Function to search the Kth smallest element
int kthSmallest(int arr[], int n, int k) {
    if (k < 1 || k > n) {
        printf("Invalid value of K. K should be between 1 and %d.\n", n);
        return -1; // Error
    }

    // Perform Selection Sort
    selectionSort(arr, n);

    // Print the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Return the Kth smallest element
```

```
    return arr[k - 1];
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    int K = 2; // Example: Search for the 2nd smallest element

    int result = kthSmallest(arr, arr_size, K);

    if (result != -1)
        printf("The %dth smallest element is: %d\n", K, result);

    return 0;
}
```

OUTPUT

Sorted array: 5 6 11 12 13

The 2nd smallest element is: 6

RESULT

Thus the above search the Kth smallest element using selection sort program has been created and executed successfully

Ex. No.: 09	**FIND THE OPTIMAL SOLUTION FOR THE GIVEN KNAPSACK PROBLEM USING GREEDY METHOD.**
Date	

AIM

To find the optimal solution for the given Knapsack problem using Greedy method

PROCEDURE

Step 1: Start the process

Step 2: Input the weights and values of items.

Step 3: Calculate the value-to-weight ratio for each item.

Step 4: Sort the items based on their value-to-weight ratio in descending order.

Step 5: Initialize the current weight in the knapsack as 0.

Step 6: Initialize the total value in the knapsack as 0.

Step 7: For each item in the sorted list:

i. If adding the entire item doesn't exceed the knapsack capacity, add the entire item.

ii. Otherwise, add a fraction of the item to fill the remaining capacity.

Step 8: Stop the process

CODING

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent an item
struct Item {
    int value;
    int weight;
};
// Function to compare items based on their value-to-weight ratio
int compare(const void* a, const void* b) {
    double ratioA = ((struct Item*)a)->value / (double)((struct Item*)a)->weight;
    double ratioB = ((struct Item*)b)->value / (double)((struct Item*)b)->weight;
    return (ratioB > ratioA) ? 1 : -1;
}
// Function to find the optimal solution for the Knapsack Problem using the
Greedy Method
void knapsackGreedy(struct Item items[], int n, int capacity) {
    // Sort items based on value-to-weight ratio
    qsort(items, n, sizeof(struct Item), compare);
    int currentWeight = 0; // Current weight in the knapsack
    double totalValue = 0.0; // Total value in the knapsack
    // Iterate through sorted items
    for (int i = 0; i < n; i++) {
        // If adding the entire item doesn't exceed the capacity, add the entire item
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            // Add a fraction of the item to fill the remaining capacity
            double fraction = (double)(capacity - currentWeight) / items[i].weight;
            currentWeight += fraction * items[i].weight;
            totalValue += fraction * items[i].value;
            break; // Knapsack is full
        }
    }
```

```
    // Print the result
    printf("Optimal value in Knapsack: %.2lf\n", totalValue);
}
int main() {
    struct Item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int n = sizeof(items) / sizeof(items[0]);
    int capacity = 50;
    // Find the optimal solution using the Greedy Method
    knapsackGreedy(items, n, capacity);
    return 0;
}
```

OUTPUT

Optimal value in Knapsack: 240.00

RESULT

Thus the above optimal solution for the given Knapsack problem using Greedy method program has been created and executed successfully

Ex. No.: 10	FIND ALL PAIRS SHORTEST PATH FOR THE GIVEN GRAPH USING DYNAMIC PROGRAMMING METHOD
Date	

AIM

To find all pairs shortest path for the given Graph using Dynamic Programming method

PROCEDURE

Step 1: Start the Process

Step 2: **Define the Graph**: Represent the graph using an adjacency matrix, where graph[i][j] represents the weight of the edge from vertex i to vertex j.

Step 3: **Initialize Distances**: Create a 2D array dist[][] to store the shortest distances between all pairs of vertices. Initialize dist[i][j] with the weight of the edge for every edge (i, j) if it exists, otherwise initialize it to infinity.

Step 4: **Dynamic Programming Algorithm**:

a) Iterate through all vertices k from 0 to V-1 (where V is the number of vertices).

b) For each pair of vertices (i, j):

If dist[i][k] and dist[k][j] are not INT_MAX, and the sum of dist[i][k] and dist[k][j] is less than dist[i][j], update dist[i][j] with their sum.

Step 5: **Print the Result**: Display the shortest distances between all pairs of vertices stored in the dist[][] array.

Step 6: Stop the Process

CODING

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

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

// Function to find all pairs shortest path using Dynamic Programming
void find_all_pairs_shortest_path(int graph[][V])
{
    int dist[V][V]; // Array to store shortest distances between all pairs of vertices

    // Initialize distances based on the graph edges
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++) {
            dist[u][v] = graph[u][v];
        }
    }

    // Dynamic programming to find the shortest path between all pairs of vertices
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                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 shortest distances between all pairs of vertices
    printf("Shortest distances between all pairs 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\t");
            } else {
                printf("%d\t", dist[i][j]);
            }
        }
        printf("\n");
    }
}
```

```
// Driver code
int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 3, 0, 7},
        {8, 0, 2, 0},
        {5, 0, 0, 1},
        {2, 0, 0, 0}
    };

    find_all_pairs_shortest_path(graph);

    return 0;
}
```

OUTPUT

Shortest distances between all pairs of vertices:

```
0   3   5   4
8   0   2   3
5   8   0   1
2   5   7   0
```

RESULT

Thus the above optimal solution for the given find all pairs shortest path for the given graph using dynamic programming method program has been created and executed successfully

Ex. No.: 11	**FIND THE SINGLE SOURCE SHORTEST PATH FOR THE GIVEN TRAVELLING SALESMAN PROBLEM USING DYNAMIC PROGRAMMING METHOD**
Date	

AIM

To find the single source shortest path for the given travelling salesman problem using dynamic programming method

PROCEDURE

Step 1: Start the Process

Step 2: **Define the Graph**: Represent the graph with weighted edges as an adjacency matrix.

Step 3: **Initialize**: Create a 2D array dp[][] of size 2^N x N (where N is the number of vertices) to store the minimum cost of visiting all vertices starting from each vertex.

Step 4: **Base Case**: Set dp[1 << src][src] = 0, where src is the source vertex.

Step 5: **Dynamic Programming**: Use bit manipulation to generate all possible subsets of vertices. For each subset mask:

a) If mask includes the source vertex src, skip the iteration.

b) For each vertex u in the subset mask, calculate the minimum cost to reach u from any vertex v in the subset mask - {u}. Update dp[mask][u] accordingly.

Step 6: **Find the Minimum Cost**: Iterate through all vertices u (except the source vertex src) and find the minimum cost to reach the source vertex src from vertex u. This will be the minimum cost of the TSP.

Step 7: Stop the process

CODING

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

#define N 4 // Number of vertices

int min(int a, int b) {
    return (a < b) ? a : b;
}

int tsp(int graph[][N], int src) {
    int dp[1 << N][N];

    // Initialize dp array
    for (int i = 0; i < (1 << N); i++) {
        for (int j = 0; j < N; j++) {
            dp[i][j] = INT_MAX;
        }
    }

    dp[1 << src][src] = 0; // Base case

    // Dynamic Programming
    for (int mask = 0; mask < (1 << N); mask++) {
        for (int u = 0; u < N; u++) {
            if (mask & (1 << u)) {
                for (int v = 0; v < N; v++) {
                    if (mask & (1 << v)) {
                        dp[mask][u] = min(dp[mask][u], graph[v][u] + dp[mask ^ (1 << u)][v]);
                    }
                }
            }
        }
    }

    // Find the minimum cost of TSP
    int minCost = INT_MAX;
    for (int u = 0; u < N; u++) {
        if (u != src) {
            minCost = min(minCost, graph[u][src] + dp[(1 << N) - 1][u]);
        }
    }

    return minCost;
}
```

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

    int src = 0; // Source vertex

    printf("Minimum cost of TSP starting from vertex %d: %d\n", src, tsp(graph, src));

    return 0;
}
```

OUTPUT

Minimum cost of TSP starting from vertex 0: 80

RESULT

Thus the above find the single source shortest path for the given travelling salesman problem using dynamic programming method program has been created and executed successfully

Ex. No.: 12	FIND ALL POSSIBLE SOLUTION FOR AN N QUEEN PROBLEM USING BACKTRACKING METHOD
Date	

AIM

To find all possible solution for an N Queen problem using backtracking method

PROCEDURE

Step 1: Start the Process

Step 2: **Initialize the Chessboard**: Create an N x N chessboard grid, where N represents the number of queens to be placed.

Step 3**: Initialize Positions**: Start with the first row and place the first queen in the first column.

Step 4: **Backtracking Algorithm:**

a) Try placing the next queen in the next column of the current row.

b) If it is a valid position (not under attack by any previously placed queens), move to the next row and repeat the process.

c) If there are no valid positions in the current row, backtrack to the previous row and try placing the queen in the next valid column.

d) Repeat this process until all queens are placed on the board.

Step 5: Check Valid Positions:

Step 6: Use functions to check whether placing a queen in a particular position results in conflicts with already placed queens.

Step 7: Check for conflicts in the same row, same column, and diagonals.

Step 8: Print Solutions: Once all queens are placed on the board without conflicts, print the solution.

Step 9: Stop the Process

CODING

```c
#include <stdio.h>
#define N 4
int board[N][N];

// Function to print the board
void printBoard() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
    printf("\n");
}

// Function to check if a queen can be placed in the given position
int isSafe(int row, int col) {
    int i, j;

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

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

```
        return 0;

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

    return 1;
}

// Function to solve N Queen problem using backtracking
int solveNQueen(int col) {
    if (col >= N)
        return 1;

    for (int i = 0; i < N; i++) {
        if (isSafe(i, col)) {
            board[i][col] = 1;
            if (solveNQueen(col + 1))
                return 1;
            board[i][col] = 0; // Backtrack
        }
    }
    return 0;
}

// Main function
int main() {
```

```
    // Initialize the board with all 0s
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = 0;

    if (!solveNQueen(0)) {
        printf("Solution does not exist\n");
        return 0;
    }

    // Print the board with the solution
    printf("Solution for N Queen problem:\n");
    printBoard();

    return 0;
}
```

OUTPUT

Solution for N Queen problem:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

RESULT

Thus the above find all possible solution for an N Queen problem using backtracking method program has been created and executed successfully

Ex. No.: 13	FIND ALL POSSIBLE HAMILTONIAN CYCLE FOR THE GIVEN GRAPH USING BACKTRACKING METHOD
Date	

AIM

To Find all possible Hamiltonian Cycle for the given graph using backtracking method

PROCEDURE

Step 1: Start the Process

Step 2**: Initialize the Graph**: Represent the graph using an adjacency matrix or adjacency list.

Step 3: **Backtracking Algorithm**:

a) Start with an empty path and add the first vertex to the path.

b) For each vertex not yet included in the path:

  i) Try adding the vertex to the path if it is adjacent to the last vertex in the path and not already visited.

  ii) If adding the vertex creates a cycle and all vertices are included in the path, print the path as a Hamiltonian Cycle.

  iii) If adding the vertex does not create a cycle, recursively explore further by adding more vertices to the path.

c) Backtrack and remove the added vertex from the path to explore other possibilities.

Step 4: **Check for Hamiltonian Cycle**:

a) A Hamiltonian Cycle is a closed path that visits every vertex exactly once.

b) After exploring all possible paths, print the Hamiltonian Cycles found.

Step 5: Stop the Process

CODING

```
#include <stdio.h>

#include <stdbool.h>

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

int graph[V][V]; // Adjacency matrix to represent the graph

int path[V];     // Array to store the Hamiltonian Cycle


// Function to check if the vertex can be added to the path

bool isSafe(int v, int pos, int path[], int graph[V][V]) {

    if (graph[path[pos - 1]][v] == 0) // Check if vertex v is adjacent to the last vertex in the path

        return false;


    // Check if the vertex has already been visited

    for (int i = 0; i < pos; i++)

        if (path[i] == v)

            return false;


    return true;

}


// Function to print the Hamiltonian Cycle

void printPath(int path[]) {

    printf("Hamiltonian Cycle: ");

    for (int i = 0; i < V; i++)

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

    printf("%d\n", path[0]); // Print the first vertex again to complete the cycle

}
```

```
// Function to find Hamiltonian Cycle using backtracking
bool hamiltonianCycleUtil(int graph[V][V], int path[], int pos) {
    if (pos == V) {
        // Check if the last vertex in the path is adjacent to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1) {
            printPath(path);
            return true;
        }
        return false;
    }

    // Try adding vertices to the path
    for (int v = 1; v < V; v++) {
        if (isSafe(v, pos, path, graph)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1))
                return true;
            path[pos] = -1; // Backtrack
        }
    }

    return false;
}

// Function to find Hamiltonian Cycle in the given graph
void findHamiltonianCycle(int graph[V][V]) {
    int path[V];
```

```
    // Initialize all vertices as not visited
    for (int i = 0; i < V; i++)
        path[i] = -1;

    // Start from the first vertex (0)
    path[0] = 0;

    // Try to find Hamiltonian Cycle starting from the first vertex
    if (!hamiltonianCycleUtil(graph, path, 1))
        printf("No Hamiltonian Cycle exists\n");
}
// Main function
int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };

    // Find and print all Hamiltonian Cycles in the graph
    findHamiltonianCycle(graph);

    return 0;
}
```

OUTPUT

Hamiltonian Cycle: 0 1 2 4 3 0

Hamiltonian Cycle: 0 3 4 2 1 0

Hamiltonian Cycle: 0 3 4 1 2 0

Hamiltonian Cycle: 0 2 4 3 1 0

Hamiltonian Cycle: 0 2 4 1 3 0

RESULT

Thus the above Find all possible Hamiltonian Cycle for the given graph using backtracking method program has been created and executed successfully