Catalog

**Exp_01 - LInear and Binary Search (Recursive)\LinearSearch.c**

```c
#include <stdio.h>

int linearSearch(int arr[], int size, int key, int index) {
    if (index >= size)
        return -1;

    if (arr[index] == key)
        return index;

    return linearSearch(arr, size, key, index + 1);
}

int main() {
    int n, key, result;

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

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

    printf("Enter element to search: ");
    scanf("%d", &key);

    result = linearSearch(arr, n, key, 0);

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

    return 0;
}
```

**Exp_01 - LInear and Binary Search (Recursive)\binSearch.c**

```c
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int x)
{
    if (right >= left)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, left, mid - 1, x);

        return binarySearch(arr, mid + 1, right, x);
    }

    return -1;
}

int main()
{
    int arr[] = {2, 3, 4, 10, 40, 50, 70, 80};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x;

    printf("Enter element to search: ");
    scanf("%d", &x);

    int result = binarySearch(arr, 0, n - 1, x);

    if (result == -1)
        printf("Element is not present in array\n");
    else
        printf("Element is present at index %d\n", result);

    return 0;
}
```

**Exp_02_Heap_Sort\Heapsort.c**

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

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

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

    int arr[n];
    printf("Enter elements: ");
    for (int i = 0; i < n; i++)
```

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

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

    heapSort(arr, n);

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

    return 0;
}
```

**Exp_03_Merge_Sort\MergeSort.c**

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

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

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

    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) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

    free(L);
    free(R);
}

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);
```

```c
    }
}

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

int main() {
    int n;

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

    int *arr = (int*)malloc(n * sizeof(int));

    printf("Enter elements: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

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

    mergeSort(arr, 0, n - 1);

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

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

**Exp_04_Selection Sort\selectionsort.c**

```c
#include <stdio.h>

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

    for (i = 0; i < n - 1; i++) {
        minIdx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        if (minIdx != i) {
            temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}

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

int main() {
    int n;

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

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

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

    selectionSort(arr, n);

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

    return 0;
}
```

**Exp_05_Insertion_Sort\InsertionSort.c**

```c
#include <stdio.h>

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;
    }
}

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

int main() {
    int n;

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

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

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

    insertionSort(arr, n);

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

    return 0;
}
```

**Exp_06_Quick_Sort\QuickSort.c**

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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

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);
    }
}

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

int main() {
    int n;

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

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

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

```c

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

**Exp_07\KnapsackGreedy.c**

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

typedef struct {
    int weight;
    int value;
    double ratio;
    int index;
} Item;

int compare(const void *a, const void *b) {
    Item *itemA = (Item *)a;
    Item *itemB = (Item *)b;
    if (itemB->ratio > itemA->ratio) return 1;
    if (itemB->ratio < itemA->ratio) return -1;
    return 0;
}

void knapsackGreedy(int weights[], int values[], int n, int capacity) {
    Item items[n];

    // Calculate value-to-weight ratio
    for (int i = 0; i < n; i++) {
        items[i].weight = weights[i];
        items[i].value = values[i];
        items[i].ratio = (double)values[i] / weights[i];
        items[i].index = i + 1;
    }

    // Sort by ratio in descending order
    qsort(items, n, sizeof(Item), compare);

    double totalValue = 0.0;
    int remainingCapacity = capacity;
    double fractions[n];

    for (int i = 0; i < n; i++) {
        fractions[i] = 0.0;
    }

    printf("\nItem\tWeight\tValue\tRatio\t\tFraction\n");
    printf("-------------------------------------------------------------\n");

    for (int i = 0; i < n; i++) {
        if (remainingCapacity >= items[i].weight) {
            // Take full item
            fractions[items[i].index - 1] = 1.0;
            totalValue += items[i].value;
            remainingCapacity -= items[i].weight;
            printf("%d\t%d\t%d\t%.2f\t\t%.2f\n", items[i].index, items[i].weight,
                   items[i].value, items[i].ratio, 1.0);
```

```c
        } else if (remainingCapacity > 0) {
            // Take fractional item
            double fraction = (double)remainingCapacity / items[i].weight;
            fractions[items[i].index - 1] = fraction;
            totalValue += items[i].value * fraction;
            printf("%d\t%d\t%d\t%.2f\t\t%.2f\n", items[i].index, items[i].weight,
                   items[i].value, items[i].ratio, fraction);
            remainingCapacity = 0;
            break;
        }
    }

    printf("-----------------------------------------------------------\n");
    printf("Maximum value: %.2f\n", totalValue);
}

int main() {
    int n, capacity;

    printf("Enter number of items: ");
    scanf("%d", &n);

    int weights[n], values[n];

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

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

    printf("Enter knapsack capacity: ");
    scanf("%d", &capacity);

    knapsackGreedy(weights, values, n, capacity);

    return 0;
}
```

**Exp_08\TravellingSalesman.c**

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

#define MAX 20

int n;
int graph[MAX][MAX];
int dp[MAX][1 << MAX];

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

int tsp(int pos, int mask) {
    if (mask == (1 << n) - 1) {
        return graph[pos][0];
    }

    if (dp[pos][mask] != -1) {
        return dp[pos][mask];
    }

    int ans = INT_MAX;

    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) {
            int newAns = graph[pos][city] + tsp(city, mask | (1 << city));
            ans = min(ans, newAns);
        }
    }

    return dp[pos][mask] = ans;
}

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

    printf("Enter cost matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < (1 << MAX); j++) {
            dp[i][j] = -1;
        }
    }
```

```c

    int result = tsp(0, 1);

    printf("Minimum cost: %d\n", result);

    return 0;
}
```

**Exp_09\minSpaningTree.c**

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

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

typedef struct {
    int parent, rank;
} Subset;

int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void unionSets(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    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++;
    }
}

int compareEdges(const void* a, const void* b) {
    return ((Edge*)a)->weight - ((Edge*)b)->weight;
}

void kruskalMST(Edge edges[], int V, int E) {
    qsort(edges, E, sizeof(Edge), compareEdges);

    Subset* subsets = (Subset*)malloc(V * sizeof(Subset));
    for (int i = 0; i < V; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    Edge result[V];
    int e = 0, i = 0;

    printf("Edges in MST:\n");
    while (e < V - 1 && i < E) {
        Edge next_edge = edges[i++];
        int x = find(subsets, next_edge.src);
```

```c
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            printf("%d -- %d == %d\n", next_edge.src, next_edge.dest,
next_edge.weight);
            unionSets(subsets, x, y);
        }
    }

    int totalWeight = 0;
    for (int i = 0; i < e; i++)
        totalWeight += result[i].weight;
    printf("Total weight of MST: %d\n", totalWeight);

    free(subsets);
}

int main() {
    int V, E;
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);

    Edge* edges = (Edge*)malloc(E * sizeof(Edge));
    printf("Enter edges (src dest weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }

    kruskalMST(edges, V, E);

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

**Exp_10\N_Queen.c**

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

bool isSafe(int** board, int row, int col, int n) {
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    for (int i = row, j = col; i < n && j >= 0; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQueenUtil(int** board, int col, int n) {
    if (col >= n)
        return true;

    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            board[i][col] = 1;

            if (solveNQueenUtil(board, col + 1, n))
                return true;

            board[i][col] = 0;
        }
    }

    return false;
}

void printSolution(int** board, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
}

bool solveNQueen(int n) {
    int** board = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        board[i] = (int*)calloc(n, sizeof(int));
    }
```

```
52
53      if (!solveNQueenUtil(board, 0, n)) {
54          printf("Solution does not exist\n");
55          for (int i = 0; i < n; i++)
56              free(board[i]);
57          free(board);
58          return false;
59      }
60
61      printSolution(board, n);
62
63      for (int i = 0; i < n; i++)
64          free(board[i]);
65      free(board);
66      return true;
67  }
68
69  int main() {
70      int n;
71      printf("Enter the value of N: ");
72      scanf("%d", &n);
73
74      solveNQueen(n);
75
76      return 0;
77  }
```

**Exp_11\0_1Knapsack.c**

```c
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][W];
}

int main() {
    int n, W;

    printf("Enter number of items: ");
    scanf("%d", &n);

    int val[n], wt[n];

    printf("Enter values: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &val[i]);

    printf("Enter weights: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &wt[i]);

    printf("Enter knapsack capacity: ");
    scanf("%d", &W);

    int maxValue = knapsack(W, wt, val, n);
    printf("Maximum value: %d\n", maxValue);

    return 0;
}
```

**Exp_12\Dijkastra.c**

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

#define MAX 100
#define INF INT_MAX

int minDistance(int dist[], bool visited[], int n) {
    int min = INF, minIndex;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && dist[i] < min) {
            min = dist[i];
            minIndex = i;
        }
    }
    return minIndex;
}

void dijkstra(int graph[MAX][MAX], int n, int src) {
    int dist[MAX];
    bool visited[MAX];

    for (int i = 0; i < n; i++) {
        dist[i] = INF;
        visited[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, visited, n);
        visited[u] = true;

        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printf("Vertex\t\tDistance from Source (%d)\n", src);
    for (int i = 0; i < n; i++) {
        printf("%d\t\t", i);
        if (dist[i] == INF)
            printf("INF\n");
        else
            printf("%d\n", dist[i]);
    }
}

```

```c
int main() {
    int n, src;
    int graph[MAX][MAX];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix (0 for no edge):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter source vertex: ");
    scanf("%d", &src);

    dijkstra(graph, n, src);

    return 0;
}
```

**Exp_13\0_1Knapsack_DP_Greedy.c**

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

// Structure to represent an item
typedef struct {
    int weight;
    int value;
    double ratio; // value-to-weight ratio for greedy approach
} Item;

// Function to find maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Dynamic Programming approach for 0/1 Knapsack
int knapsackDP(int weights[], int values[], int n, int capacity) {
    int dp[n + 1][capacity + 1];

    // Build table dp[][] in bottom-up manner
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weights[i - 1] <= w)
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]],
                               dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][capacity];
}

// Comparison function for qsort (descending order of ratio)
int compare(const void *a, const void *b) {
    Item *item1 = (Item *)a;
    Item *item2 = (Item *)b;
    if (item2->ratio > item1->ratio)
        return 1;
    else if (item2->ratio < item1->ratio)
        return -1;
    return 0;
}

// Greedy approach for 0/1 Knapsack (approximation)
int knapsackGreedy(int weights[], int values[], int n, int capacity) {
    Item items[n];

    // Create items array with value-to-weight ratio
```

```c
    for (int i = 0; i < n; i++) {
        items[i].weight = weights[i];
        items[i].value = values[i];
        items[i].ratio = (double)values[i] / weights[i];
    }

    // Sort items by ratio in descending order
    qsort(items, n, sizeof(Item), compare);

    int totalValue = 0;
    int currentWeight = 0;

    // Pick items based on greedy choice
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        }
    }

    return totalValue;
}

int main() {
    int n, capacity;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    int weights[n], values[n];

    printf("Enter the weights of items:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d", &weights[i]);
    }

    printf("Enter the values of items:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d", &values[i]);
    }

    printf("Enter the knapsack capacity: ");
    scanf("%d", &capacity);

    // Dynamic Programming approach
    int dpResult = knapsackDP(weights, values, n, capacity);
    printf("\n--- Dynamic Programming Approach ---\n");
    printf("Maximum value (DP): %d\n", dpResult);

    // Greedy approach
    int greedyResult = knapsackGreedy(weights, values, n, capacity);
    printf("\n--- Greedy Approach ---\n");
```

```c
    printf("Maximum value (Greedy): %d\n", greedyResult);

    printf("\nNote: Greedy approach may not always give optimal solution for 0/1 Knapsack.\n");

    return 0;
}
```

**Exp_14\Kruskal.c**

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

#define MAX 30

typedef struct edge {
    int u, v, w;
} edge;

typedef struct edge_list {
    edge data[MAX];
    int n;
} edge_list;

edge_list elist;
int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskal();
int find(int belongs[], int vertexno);
void union1(int belongs[], int c1, int c2);
void sort();
void print();

void kruskal() {
    int belongs[MAX], i, j, cno1, cno2;
    elist.n = 0;

    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++) {
            if (Graph[i][j] != 0) {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = Graph[i][j];
                elist.n++;
            }
        }

    sort();

    for (i = 0; i < n; i++)
        belongs[i] = i;

    spanlist.n = 0;

    for (i = 0; i < elist.n; i++) {
        cno1 = find(belongs, elist.data[i].u);
        cno2 = find(belongs, elist.data[i].v);

        if (cno1 != cno2) {
            spanlist.data[spanlist.n] = elist.data[i];
```

```
            spanlist.n++;
            union1(belongs, cno1, cno2);
        }
    }
}

int find(int belongs[], int vertexno) {
    return belongs[vertexno];
}

void union1(int belongs[], int c1, int c2) {
    int i;
    for (i = 0; i < n; i++)
        if (belongs[i] == c2)
            belongs[i] = c1;
}

void sort() {
    int i, j;
    edge temp;

    for (i = 1; i < elist.n; i++)
        for (j = 0; j < elist.n - 1; j++)
            if (elist.data[j].w > elist.data[j + 1].w) {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
}

void print() {
    int i, cost = 0;
    printf("\nMinimum Spanning Tree Edges:\n");
    for (i = 0; i < spanlist.n; i++) {
        printf("\nEdge %d: (%d, %d) with weight %d",
               i + 1, spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
        cost += spanlist.data[i].w;
    }
    printf("\n\nTotal Cost of Spanning Tree: %d\n", cost);
}

int main() {
    int i, j, total_cost;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("\nEnter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &Graph[i][j]);

    kruskal();
    print();
```

```
106
107     return 0;
108 }
```

**Exp_15\a_Floyd.c**

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

#define V 4
#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall(int graph[][V]) {
    int dist[V][V], i, j, k;

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

    // Add all vertices one by one to the set of intermediate vertices
    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from i to j,
                // then update the value of dist[i][j]
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    printSolution(dist);
}

void printSolution(int dist[][V]) {
    printf("Following matrix shows the shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

int main() {
```

```
    int graph[V][V] = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };

    floydWarshall(graph);

    return 0;
}
```

**Exp_15\b_TravellingSalesman_DP.c**

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

#define N 4  // Number of cities
#define INF INT_MAX

int dist[N][N];  // Distance matrix
int dp[1 << N][N];  // DP table: dp[mask][i] = min cost to visit cities in mask ending at city i

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

// Function to solve TSP using Dynamic Programming
int tsp(int mask, int pos) {
    // If all cities have been visited
    if (mask == (1 << N) - 1) {
        return dist[pos][0];  // Return to starting city
    }

    // If already computed
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }

    int ans = INF;

    // Try visiting all unvisited cities
    for (int city = 0; city < N; city++) {
        // If city is not visited and there is a path
        if ((mask & (1 << city)) == 0 && dist[pos][city] != INF) {
            int newAns = dist[pos][city] + tsp(mask | (1 << city), city);
            ans = min(ans, newAns);
        }
    }

    return dp[mask][pos] = ans;
}

int main() {
    // Initialize distance matrix
    printf("Enter the distance matrix (%d x %d):\n", N, N);
    printf("(Enter %d for no direct path)\n", INF);

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &dist[i][j]);
        }
    }

    // Initialize DP table with -1
```

```
    for (int i = 0; i < (1 << N); i++) {
        for (int j = 0; j < N; j++) {
            dp[i][j] = -1;
        }
    }

    // Start from city 0 with only city 0 visited
    int minCost = tsp(1, 0);

    if (minCost >= INF) {
        printf("\nNo valid tour exists!\n");
    } else {
        printf("\nMinimum cost of TSP tour: %d\n", minCost);
    }

    return 0;
}
```

**Exp_16\HamiltonianCycles.c**

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

#define MAX 20

int graph[MAX][MAX];
int path[MAX];
int n;
int cycleCount = 0;

void printPath() {
    printf("Hamiltonian Cycle %d: ", ++cycleCount);
    for (int i = 0; i < n; i++) {
        printf("%d ", path[i]);
    }
    printf("%d\n", path[0]); // Return to starting vertex
}

bool isSafe(int v, int pos) {
    // Check if vertex v is adjacent to the previous vertex
    if (graph[path[pos - 1]][v] == 0)
        return false;

    // Check if vertex v is already in path
    for (int i = 0; i < pos; i++) {
        if (path[i] == v)
            return false;
    }

    return true;
}

bool hamiltonianCycleUtil(int pos) {
    // Base case: if all vertices are included in path
    if (pos == n) {
        // Check if there is an edge from the last vertex to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1) {
            printPath();
            return true; // Continue to find more cycles
        }
        return false;
    }

    // Try different vertices as the next candidate
    for (int v = 1; v < n; v++) {
        if (isSafe(v, pos)) {
            path[pos] = v;

            // Recur to construct rest of the path
            hamiltonianCycleUtil(pos + 1);

```

```
            // Backtrack
            path[pos] = -1;
        }
    }

    return false;
}

void findHamiltonianCycles() {
    // Initialize path array
    for (int i = 0; i < n; i++)
        path[i] = -1;

    // Start from vertex 0
    path[0] = 0;
    cycleCount = 0;

    printf("\nFinding all Hamiltonian Cycles:\n");
    hamiltonianCycleUtil(1);

    if (cycleCount == 0) {
        printf("No Hamiltonian Cycle exists in the given graph.\n");
    } else {
        printf("\nTotal number of Hamiltonian Cycles: %d\n", cycleCount);
    }
}

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

    printf("\nEnter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("\nAdjacency Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }

    findHamiltonianCycles();

    return 0;
}
```