Lab 1

Write a program to perform operation count for a given pseudo code.

Theory:
Pseudocode is a simplified way of describing an algorithm using plain language rather than a specific programming language.

Algorithm to Perform Operation Count:
1. Identify Primitive Operations: Break down pseudocode into smallest operations like arithmetic, comparisons, assignments, etc.
2. Analyze Loop Structures: For loops and while loops execute based on conditions, count iterations and operations in each.
3. Account for Conditional Statements: Consider worst-case, best-case, and average-case scenarios.
4. Sum Up Operation Counts: Add counts for each line, considering loop and condition frequencies.
5. Express Result: The operation count is expressed as a function of input size (n), typically using Big O notation.

Pseudo Code:

```
for i = 1 to n
   for j = 1 to i
      sum = sum + j
```

Source Code:

```
#include <stdio.h>

void operation_count(int n) {
   int assignment_count = 0, comparison_count = 0, addition_count = 0, sum = 0;
   for (int i = 1; i <= n; i++) {
      comparison_count++;
      assignment_count++;
      for (int j = 1; j <= i; j++) {
         comparison_count++;
         addition_count++;
         sum += j;
         assignment_count++;
      }
   }
   comparison_count++;
   printf("Assignments: %d\n", assignment_count);
   printf("Comparisons: %d\n", comparison_count);
   printf("Additions: %d\n", addition_count);
}

int main() {
   int n;
   printf("Enter the value of n: ");
   scanf("%d", &n);
   operation_count(n);
   return 0;
}
```

Output:

```
Enter the value of n: 3
Assignments: 7
Comparisons: 7
Additions: 6
```

Lab 2

Write a program to perform bubble sort for any given list of numbers.

Theory:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the list is sorted. Its time complexity is $O(n^2)$ for an average and worst case, making it inefficient for large lists.

Algorithm:

1. Start
2.For the first iteration, compare all the elements (n). For the subsequent runs, compare (n-1) (n-2) and so on.
3. Compare each element with its right-side neighbor.
4. Swap the smallest element to the left.
5. Keep repeating steps 1-3 until the whole list is covered.
6. Stop

Code:

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
   int temp;
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

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

int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Original array:\n");
   printArray(arr, n);
   bubbleSort(arr, n);
   printf("Sorted array:\n");
   printArray(arr, n);
   return 0;
}
```

Output:

```
Original array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

Lab 3

Write a program to perform insertion sort for any given list of numbers.

Theory:

Insertion Sort is a simple, comparison-based sorting algorithm that builds a sorted portion of the list one element at a time. Each element is compared to those before it and inserted in its correct position. It has an average and worst-case time complexity of $O(n^2)$, making it suitable for small datasets.

Algorithm:

1. Start
2. Consider the first element to be sorted and the rest to be unsorted
3. Compare with the second element:
    a. If the second element< the first element, insert the element in the correct position of the sorted portion
    b. Else, leave it as it is
4. Repeat 1 and 2 until all elements are sorted
5. Stop

Code:

```
#include <stdio.h>

void insertionSort(int A[], int n) {
   int i, j, temp;
   for (i = 1; i < n; i++) {
      temp = A[i];
      j = i - 1;
      while (j >= 0 && A[j] > temp) {
         A[j + 1] = A[j];
         j = j - 1;
      }
      A[j + 1] = temp;
   }
}
void printArray(int A[], int size) {
   int i;
   for (i = 0; i < size; i++) {
      printf("%d ", A[i]);
   }
   printf("\n");
}

int main() {
   int A[] = {12, 11, 13, 5, 6};
   int n = sizeof(A) / sizeof(A[0]);
   printf("Original array:\n");
   printArray(A, n);
   insertionSort(A, n);
   printf("Sorted array:\n");
   printArray(A, n);
   return 0;
}
```

Output:

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

Lab 4

Write a program to perform Quicksort for the given list of numbers.

Theory:
Quick Sort is a divide-and-conquer sorting algorithm that selects a "pivot" element and partitions the array into two halves — elements less than the pivot on one side and elements greater on the other. It then recursively sorts each partition. Quick Sort is efficient with an average time complexity of O(n log n), but its worst-case time complexity is $O(n^2)$ if the pivot choices are poor.

Algorithm:
1.Choose a pivot element from the array.
2.Partition the array so that elements less than the pivot are on the left and those greater are on the right.
3.Recursively apply Quicksort to the left and right partitions.
4.Repeat until the entire array is sorted.

Code:

```
#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 p = arr[low];
   int i = low + 1;
   int j = high;
   while (1) {
      while (i <= j && arr[i] <= p) {
         i++;
      }
      while (i <= j && arr[j] >= p) {
         j--;
      }
      if (i <= j) {
         swap(&arr[i], &arr[j]);
      } else {
         break;
      }
   }
   swap(&arr[low], &arr[j]);
   return j;
}

void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pivot = partition(arr, low, high);
      quickSort(arr, low, pivot - 1);
      quickSort(arr, pivot + 1, high);
   }
}
```

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

int main() {
    int arr[] = {5, 8, 1, 2, 6, 3, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:

```
Original array:
5 8 1 2 6 3 9
Sorted array:
1 2 3 5 6 8 9
```

Lab 5

Write a program to find maximum and minimum for a given list of numbers.

Theory:

Finding the maximum and minimum values in a list involves scanning each element to identify the highest and lowest numbers. This process is linear, requiring O(n) time complexity, as each element is examined once. This is efficient and effective for small to medium-sized lists.

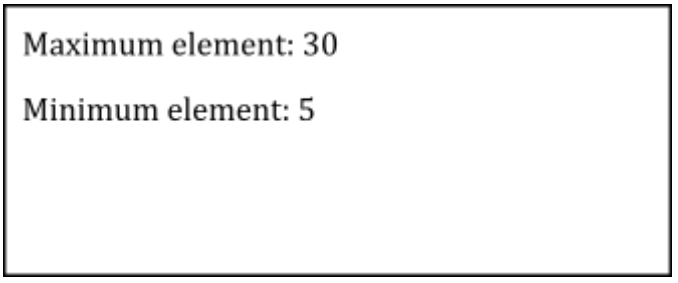
Algorithm:

1.Initialize max and min variables to the first element in the list.
2.For each element in the list:
    a.If the element is greater than max, update max.
    b.If the element is smaller than min, update min.
3.At the end, max holds the largest value and min holds the smallest.

Code:

```
#include <stdio.h>

 int main() {
   int arr[] = {10, 25, 5, 30, 15};
   int n = sizeof(arr) / sizeof(arr[0]);
   int max = arr[0], min = arr[0];


   for (int i = 1; i < n; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }

     else if (arr[i] < min) {
         min = arr[i];
      }

   }


   printf("Maximum element: %d\n", max);
   printf("Minimum element: %d\n", min);
   return 0;

}
```

Output:



Lab 6

Write a program to perform merge sort for the given list of numbers.

Theory:
Merge Sort is a divide-and-conquer sorting algorithm that splits the array into halves, recursively sorts each half, and then merges the sorted halves back together. It has a time complexity of O(n log n) in all cases, making it efficient for large datasets. The main drawback is that it requires additional space for the merged arrays.

Algorithm:
1. Start
2. Split the unsorted list into two groups recursively until there is one element per group
3. Compare each of the elements and then sort and group them
4. Repeat step 2 until the whole list is merged and sorted in the process
5. Stop

Code:

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1;
   int 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) {
      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++;
   }
}
```

```
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[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    mergeSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:

```
Original array:
38 27 43 3 9 82 10
Sorted array:
3 9 10 27 38 43 82
```

Lab 7

Write a program to perform binary search for a given set of integer values recursively and non-recursively.

Theory:
Binary Search is an efficient algorithm used to find a target value within a sorted array or list. It operates by dividing the search interval in half repeatedly. If the target value is less than the middle element of the array, the search continues in the lower half; if the target value is greater, the search continues in the upper half. The recursive approach to binary search involves the function calling itself with updated parameters, while the non-recursive approach utilizes a loop to achieve the same goal.

Recursive Binary Search Algorithm:
1.Check if the current range is valid (low ≤ high).
2.Calculate the mid index.
3.If the mid element equals the target, return the mid index.
4.If the mid element is greater than the target, search in the left half by calling the function with updated parameters.
5.If the mid element is less than the target, search in the right half by calling the function with updated parameters.
6.If the range is invalid, return -1 (target not found).

Non-Recursive Binary Search Algorithm:
1.Set low to 0 and high to the last index of the list.
2.While low ≤ high:
a.Calculate the mid index.
b.If the mid element equals the target, return the mid index.
c.If the mid element is greater than the target, set high to mid - 1.
d.If the mid element is less than the target, set low to mid + 1.
3.If the target is not found, return -1.

Code:

```
#include <stdio.h>

int binarySearchRecursive(int arr[], int low, int high, int target);
int binarySearchIterative(int arr[], int size, int target);

int main() {
   int arr[] = {2, 4, 5, 7, 9, 11, 14, 18, 20};
   int size = sizeof(arr) / sizeof(arr[0]);
   int target = 9;
   int result;

   result = binarySearchRecursive(arr, 0, size - 1, target);
   if (result != -1)
      printf("Recursive: Element found at index %d\n", result);
   else
      printf("Recursive: Element not found\n");

   result = binarySearchIterative(arr, size, target);
   if (result != -1)
      printf("Non-Recursive: Element found at index %d\n", result);
```

```
    else
        printf("Non-Recursive: Element not found\n");

    return 0;
}

int binarySearchRecursive(int arr[], int low, int high, int target) {
    if (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] > target)
            return binarySearchRecursive(arr, low, mid - 1, target);
        else
            return binarySearchRecursive(arr, mid + 1, high, target);
    }
    return -1;
}

int binarySearchIterative(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] > target)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

Output:

```
Recursive: Element found at index 4
Non-Recursive: Element found at index 4
```

Lab 8

Write a program to find the solution of a knapsack problem with a greedy method.

Theory:

The Knapsack Problem is a classic optimization problem that can be solved using different methods. The Greedy Method is often used for the Fractional Knapsack Problem, where items can be broken down into smaller pieces. Here's how to solve the Fractional Knapsack Problem using the Greedy approach.

Algorithm:

1.Calculate Ratios: For each item, compute the value-to-weight ratio.
2.Sort Items: Sort items by their value-to-weight ratio in descending order.
3.Initialize Value: Set totalValue to 0.
4.Fill Knapsack:
    a.For each item:
        i.If it can fit in the knapsack, add its value to totalValue and reduce the remaining capacity.
        ii.If it cannot fit, add the fractional value of the item that fits and stop.

Source Code:

```
#include <stdio.h>

struct Item {
  int value, weight;
};

double fractionalKnapsack(struct Item items[], int n, int capacity) {
  double totalValue = 0.0;
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if ((double)items[j].value / items[j].weight < (double)items[j + 1].value / items[j + 1].weight) {
        struct Item temp = items[j];
        items[j] = items[j + 1];
        items[j + 1] = temp;
      }
    }  }
  for (int i = 0; i < n; i++) {
    if (items[i].weight <= capacity) {
      totalValue += items[i].value;
      capacity -= items[i].weight;
    } else {
      totalValue += items[i].value * ((double)capacity / items[i].weight);
      break;
    }
  }
  return totalValue;
}
int main() {
  struct Item items[] = {{60, 10}, {100, 20}, {120, 30}};
  int n = sizeof(items) / sizeof(items[0]);
  int capacity = 50;
  printf("Maximum value in the knapsack = %.2f\n", fractionalKnapsack(items, n, capacity));
  return 0;
}
```

Output:

```
Maximum value in the knapsack = 240.00
```

Lab 9

Write a program to find the minimum cost spanning tree using Prim's algorithm.

Theory:

Prim's algorithm builds the MST by starting with a single vertex and adding edges to connect it to the remaining vertices with the smallest weight. It continues this process until all vertices are included in the MST.

Algorithm:

1.Initialize: Start with a selected vertex, set its cost to 0, and all other vertices' costs to infinity.
2.Add Vertex: Select the vertex with the smallest cost and add it to the MST.
3.Update Costs: For each adjacent vertex of the selected vertex:
    a.If the edge weight is less than the current cost, update the cost.
4.Repeat: Continue until all vertices are included in the MST.

Code:

```
#include <stdio.h>
#include <limits.h>
#define V 5

int findMinKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++) {
      if (mstSet[v] == 0 && key[v] < min) {
         min = key[v];
         min_index = v;
      }   }
   return min_index;
}
void primMST(int graph[V][V]) {
   int parent[V], key[V], mstSet[V];
   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }
   key[0] = 0,  parent[0] = -1;
   for (int count = 0; count < V - 1; count++) {
      int u = findMinKey(key, mstSet);
      mstSet[u] = 1;
      for (int v = 0; v < V; v++) {
         if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
         } } }
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
}
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} };
   primMST(graph);
   return 0;
}
```

Output:

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

Lab 10

Write a program to find the minimum cost spanning tree using Kruskal's algorithm.

Theory:

Kruskal's algorithm sorts all edges in the graph by weight in ascending order, then adds edges one by one to the MST, ensuring no cycles are formed. It uses the Union-Find data structure to keep track of connected components and prevent cycles.

Algorithm:

1.Sort edges in ascending order of weight.
2.Initialize sets for each vertex using Union-Find.
3.Add Edge:
    a.For each edge, check if it forms a cycle:
        i.If not, add it to the MST.
4.Repeat until the MST contains V-1 edges, where V is the number of vertices.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define V 4
#define E 5

int parent[V];
int find(int i) {
   while (parent[i] != i)
      i = parent[i];
   return i;
}
void unionSets(int u, int v) {
   int rootU = find(u), rootV = find(v);
   parent[rootU] = rootV;
}
int compare(const void* a, const void* b) {
   return ((int*)a)[2] - ((int*)b)[2];
}
void kruskalMST(int edges[E][3]) {
   for (int i = 0; i < V; i++) parent[i] = i;
   qsort(edges, E, sizeof(edges[0]), compare);
   printf("Edge \tWeight\n");

   for (int i = 0, count = 0; count < V - 1 && i < E; i++) {
      int u = edges[i][0], v = edges[i][1], weight = edges[i][2];
      if (find(u) != find(v)) {
         printf("%d - %d \t%d\n", u, v, weight);
         unionSets(u, v);
         count++;
      } }}
int main() {
   int edges[E][3] = { {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4} };
   kruskalMST(edges);
   return 0;
}
```

Output:

```
Edge  Weight
2 - 3   4
0 - 3   5
0 - 1   10
```

Lab 11

Write a program to perform a single source shortest path problem for a given graph.

Theory:

Dijkstra's algorithm is a popular algorithm for finding the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights. It is based on the greedy approach, where the algorithm selects the shortest possible path at each step to ensure the minimum path cost to each vertex. Starting from the source, Dijkstra's algorithm iteratively explores the nearest unvisited vertices and updates the shortest distances for each until all vertices have been processed. This method ensures optimal solutions for graphs without negative weights.

Algorithm:

1.Initialize distances from the source vertex to all vertices as infinity, except for the source itself, which is set to 0.
2.Mark all vertices as unvisited.
3.Repeat until all vertices are visited:
a.Select the unvisited vertex with the smallest distance.
b.Mark the selected vertex as visited.
cUpdate the distance for each unvisited neighbor of the selected vertex by adding the weight of the edge between them, if this new path is shorter than the previously recorded distance.
4.The algorithm terminates once the shortest distances to all vertices from the source are finalized.

Code:

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

#define V 5

int minDistance(int dist[], bool visited[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   return min_index;
}

void printSolution(int dist[]) {
   printf("Vertex \tDistance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src) {
   int dist[V];
   bool visited[V] = {0};

   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX;
   dist[src] = 0;
```

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

    for (int v = 0; v < V; v++)
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
        dist[u] + graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
  }
  printSolution(dist);
}

int main() {
  int graph[V][V] = { {0, 10, 0, 0, 5},
                      {0, 0, 1, 0, 2},
                      {0, 0, 0, 4, 0},
                      {7, 0, 6, 0, 0},
                      {0, 3, 9, 2, 0} };

  dijkstra(graph, 0);
  return 0;
}
```

Output:

Edge	Weight
2 - 3	4
0 - 3	5
0 - 1	10

Lab 12

Write a program to find solutions for job Sequencing with deadline program.

Theory:

The Job Sequencing Problem is a scheduling problem where a set of jobs, each with a profit and a deadline, needs to be scheduled on a single machine to maximize the total profit.

Algorithm:

1.Sort Jobs:
- a.Sort jobs in decreasing order of profit.

2.Create a Slot Array:
- a.Create a boolean array slot to represent available time slots.

3.Fill Slots:
- a.For each job:
  - i.Iterate backward from the job's deadline to find the first available slot.
  - ii..If a slot is found, assign the job to that slot.

4.Calculate Total Profit:
- a.Iterate through the slot array and sum the profits of assigned jobs.

Code:

```
#include <stdio.h>
#include <stdbool.h>
struct Job {
   int id, deadline, profit;
};
int main() {
   struct Job arr[] = {{1, 2, 100}, {2, 1, 19}, {3, 2, 27}, {4, 1, 25}, {5, 3, 15}};
   int n = sizeof(arr) / sizeof(arr[0]);
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j].profit < arr[j + 1].profit) {
            struct Job temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }      }   }
   int result[n];
   bool slot[n];
   for (int i = 0; i < n; i++) {
      slot[i] = false;   }
   for (int i = 0; i < n; i++) {
      for (int j = (n - 1 < arr[i].deadline - 1 ? n - 1 : arr[i].deadline - 1); j >= 0; j--) {
         if (slot[j] == false) {
            result[j] = i;
            slot[j] = true;
            break;
         }      }   }
   int total_profit = 0;
   for (int i = 0; i < n; i++) {
      if (slot[i]) {
         total_profit += arr[result[i]].profit; }    }
   printf("Total Profit is %d\n", total_profit);
   return 0;
}
```

Output:

```
Total Profit is 142
```

Lab 13

Write a program for all pairs shortest path problem.

Theory:

The Floyd-Warshall algorithm is a dynamic programming algorithm that calculates the shortest paths between all pairs of vertices in a weighted graph. It works by iteratively considering each vertex as an intermediate node and updating the shortest path between all pairs of vertices.

Algorithm:

1.Initialize a 2D distance matrix that records the direct path costs between vertices, using INT_MAX for no direct path.

2.For each vertex k, consider it as an intermediate point between all pairs of vertices (i,j)(i, j)(i,j).

a.If the path from i to j through k is shorter than the direct path from i to j, update the distance.

3.Repeat for all pairs (i,j)(i, j)(i,j) and for each intermediate vertex k.

4.The final distance matrix holds the shortest path between every pair of vertices.

Code:

```
#include <stdio.h>
#define V 4
#define INF 99999

void floydWarshall(int graph[V][V]) {
   int dist[V][V];
   for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
         dist[i][j] = graph[i][j];

   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] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];

   printf("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, 3, INF, 7},
                       {8, 0, 2, INF},
                       {5, INF, 0, 1},
                       {2, INF, INF, 0} };
   floydWarshall(graph);
   return 0;
}
```

Output:

```
Shortest distances between every pair of vertices:
      0      3      5      6
      8      0      2      3
      5      8      0      1
      2      5      7      0
```

Lab 14

Write a program to solve N-QUEENS problem.

Theory:

The N-Queens problem is a classic backtracking problem where the goal is to place N chess queens on an N×N chessboard such that no two queens threaten each other. A queen can attack horizontally, vertically, and diagonally.

Algorithm:

1.Start from the leftmost column.
2.For each row in the current column, do the following:
    a.Check if placing a queen at this position is safe (no other queens in the same row, upper diagonal, or lower diagonal).
    b.If safe, place the queen at this position.
3.Recursively attempt to place the remaining queens in the next columns.
4.If placing queens in all columns is successful, a solution is found.
5.If no position is safe, backtrack to the previous column and try the next row.
6.Continue until all solutions are found.

Code:

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

#define N 4

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

bool isSafe(int board[N][N], int row, int col) {
  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; j >= 0 && i < N; i++, j--)
    if (board[i][j])
      return false;

  return true;
}

bool solveNQUtil(int board[N][N], int col) {
  if (col >= N)
```

```
      return true;

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

         if (solveNQUtil(board, col + 1))
            return true;

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

bool solveNQ() {
   int board[N][N] = {0};

   if (!solveNQUtil(board, 0)) {
      printf("Solution does not exist\n");
      return false;
   }

   printSolution(board);
   return true;
}

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

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Lab 15

Write a program to solve subset sum problem for given set of distinct numbers.

Theory:

The subset sum problem is a decision problem in computer science. Given a set of integers, the task is to determine whether there exists a subset of the given set with a sum equal to a given target sum.

Algorithm:

1.Start with the target sum and an empty subset.
2.For each element in the set, do the following:
    a.Include the element in the subset and reduce the target sum by the element's value.
    b.Recursively check if this inclusion can lead to the target sum.
    c.If the subset achieves the target, return success.
3.If including the element doesn't yield the target, exclude it and backtrack.
4.Continue until the target sum is achieved or all elements are considered.

Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10

bool subsetSum(int set[], int n, int sum) {
   if (sum == 0)
      return true;
   if (n == 0)
      return false;
   if (set[n - 1] > sum)
      return subsetSum(set, n - 1, sum);
   return subsetSum(set, n - 1, sum) || subsetSum(set, n - 1, sum - set[n - 1]);
}

int main() {
   int set[MAX], n, sum;
   printf("Enter the number of elements in the set: ");
   scanf("%d", &n);
   printf("Enter the elements of the set:\n");
   for (int i = 0; i < n; i++)
      scanf("%d", &set[i]);
   printf("Enter the target sum: ");
   scanf("%d", &sum);

   if (subsetSum(set, n, sum))
      printf("A subset with the given sum exists.\n");
   else
      printf("No subset with the given sum exists.\n");
   return 0;
}
```

Output:

```
Enter the number of elements in the set: 3
Enter the elements of the set:
3
4
6
Enter the target sum: 9
A subset with the given sum exists.
```

```
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;
}
```