Prog 1:

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

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

// Structure to represent a graph
struct Graph {
    int V, E;
    struct Edge* edge;
};

// Create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// A structure to represent a subset for union-find
struct subset {
    int parent;
    int rank;
};
```

```
// A utility function to find set of an element i (uses path compression technique)
int find(struct subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// A function that does union of two sets of x and y (uses union by rank)
void Union(struct subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    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++;
    }
}

// Compare two edges according to their weights. Used in qsort() for sorting edges
int compare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
```

```
int V = graph->V;
struct Edge result[V];
int e = 0;
int i = 0;
int totalCost = 0;

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare);

struct subset* subsets = (struct subset*)malloc(V * sizeof(struct subset));

for (int v = 0; v < V; ++v) {
   subsets[v].parent = v;
   subsets[v].rank = 0;
}

while (e < V - 1 && i < graph->E) {
   struct Edge next_edge = graph->edge[i++];

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

   if (x != y) {
      result[e++] = next_edge;
      Union(subsets, x, y);
      totalCost += next_edge.weight;
   }
}

printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
   printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
```

```
    printf("Total cost of MST: %d\n", totalCost);
    free(subsets);
}

// Driver program to test above functions
int main() {
    int V;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

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

    // Count the number of edges
    int E = 0;
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (costMatrix[i][j] != 0) {
                E++;
            }
        }
    }

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

```
    int edgeIndex = 0;
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (costMatrix[i][j]  != 0) {
                graph->edge[edgeIndex].src  = i;
                graph->edge[edgeIndex].dest  = j;
                graph->edge[edgeIndex].weight  = costMatrix[i][j];
                edgeIndex++;
            }
        }
    }

    KruskalMST(graph);

    free(graph->edge);
    free(graph);

    return 0;
}
```

Prog 2:

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

#define MAX 100

int findMinVertex(int  cost[], int visited[], int V) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++) {
```

```
        if (visited[v] == 0 && cost[v] < min) {
            min = cost[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void primMST(int graph[MAX][MAX], int V) {
    int parent[V];
    int cost[V];
    int visited[V];

    for (int i = 0; i < V; i++) {
        cost[i] = INT_MAX;
        visited[i] = 0;
    }

    cost[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = findMinVertex(cost, visited, V);
        visited[u] = 1;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && visited[v] == 0 && graph[u][v] < cost[v]) {
                parent[v] = u;
                cost[v] = graph[u][v];
            }
        }
```

```
    }

    int totalCost = 0;
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
        totalCost += graph[i][parent[i]];
    }
    printf("Total cost of MST: %d\n", totalCost);
}

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

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

    primMST(graph, V);

    return 0;
}
```

Prog 3

```
#include <stdio.h>

#define MAX 100
#define INF 99999

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

void floydWarshall(int graph[MAX][MAX], int V) {
   int dist[MAX][MAX];

   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];
            }
        }
    }

    printSolution(dist, V);
}

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

    int graph[MAX][MAX];
    printf("Enter the cost matrix (use %d to represent infinity):\n", INF);
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    floydWarshall(graph, V);

    return 0;
}
```

Prog 3b

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

```
#define MAX 100

void printSolution(int reach[MAX][MAX], int V) {
   printf("The transitive closure of the given graph is:\n");
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         printf("%d ", reach[i][j]);
      }
      printf("\n");
   }
}

void warshall(int graph[MAX][MAX], int V) {
   int reach[MAX][MAX];

   for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
         reach[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++) {
            reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
         }
      }
   }

   printSolution(reach, V);
}

int main() {
```

```
    int V;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

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

    warshall(graph, V);

    return 0;
}
```

Prog 4

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

#define MAX 100
#define INF INT_MAX

int minDistance(int dist[], int visited[], int V) {
    int min = INF, minIndex;
    for (int v = 0; v < V; v++) {
        if (visited[v] == 0 && dist[v] <= min) {
            min = dist[v];
```

```
            minIndex = v;
        }
    }
    return minIndex;
}

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

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

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

    dist[src] = 0;

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

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

```
            }
        }
    }

    printSolution(dist, V);
}

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

    int graph[MAX][MAX];
    printf("Enter the adjacency matrix (enter %d for no direct edge):\n", INF);
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
            if (graph[i][j] == INF) {
                graph[i][j] = 0;  // For simplicity, treat INF as no edge in the input
            }
        }
    }

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

    dijkstra(graph, V, src);

    return 0;
}
```

Prog 5

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

#define MAX 100

// Stack structure for topological sorting
struct Stack {
    int items[MAX];
    int top;
};

void initStack(struct Stack* stack) {
    stack->top = -1;
}

void push(struct Stack* stack, int value) {
    stack->items[++(stack->top)] = value;
}

int pop(struct Stack* stack) {
    if (stack->top == -1) {
        return -1;
    }
    return stack->items[(stack->top)--];
}

// Function to perform DFS and push vertices into the stack
void dfs(int v, int visited[], struct Stack* stack, int graph[MAX][MAX], int V) {
    visited[v] = 1;
```

```
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && !visited[i]) {
            dfs(i, visited, stack, graph, V);
        }
    }
    push(stack, v);
}

// Function to perform topological sort
void topologicalSort(int graph[MAX][MAX], int V) {
    struct Stack stack;
    initStack(&stack);
    int visited[V];

    for (int i = 0; i < V; i++) {
        visited[i] = 0;
    }

    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(i, visited, &stack, graph, V);
        }
    }

    printf("Topological ordering of vertices: ");
    while (stack.top != -1) {
        printf("%d ", pop(&stack));
    }
    printf("\n");
}
```

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

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

    topologicalSort(graph, V);

    return 0;
}
```

Prog 6

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

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

// Function to solve the 0/1 Knapsack problem using Dynamic Programming
void knapsack(int W, int n, int wt[], int val[]) {
    // Allocate memory for the dp array dynamically
```

```
int **dp = (int **)malloc((n + 1) * sizeof(int *));
for (int i = 0; i <= n; i++) {
    dp[i] = (int *)malloc((W + 1) * sizeof(int));
}

// Initialize the dp array to 0
for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= W; w++) {
        dp[i][w] = 0;
    }
}

// Fill the dp array using the Knapsack logic
for (int i = 1; i <= n; i++) {
    for (int w = 1; w <= W; w++) {
        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];
        }
    }
}

// Debugging: Print the dp table
printf("DP Table:\n");
for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= W; w++) {
        printf("%d ", dp[i][w]);
    }
    printf("\n");
}
```

```
    printf("Maximum value in knapsack: %d\n", dp[n][W]);

    // Free the allocated memory
    for (int i = 0; i <= n; i++) {
        free(dp[i]);
    }
    free(dp);
}

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

    int *val = (int *)malloc(n * sizeof(int));
    int *wt = (int *)malloc(n * sizeof(int));

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

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

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

```
    knapsack(W, n, wt, val);

    // Free the allocated memory
    free(val);
    free(wt);

    return 0;
}
```

Prog 7

```
#include <stdio.h>

#define MAX 100

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

// Function to sort items based on value-to-weight ratio in descending order
void sortItems(Item items[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (items[j].ratio < items[j + 1].ratio) {
                Item temp = items[j];
                items[j] = items[j + 1];
                items[j + 1] = temp;
            }
```

```
        }
    }
}

// Function to solve the discrete knapsack problem using a greedy approximation
void discreteKnapsack(int W, int n, Item items[]) {
    int totalValue = 0;
    int totalWeight = 0;
    int taken[n];  // Array to keep track of taken items

    // Initialize taken items
    for (int i = 0; i < n; i++) {
        taken[i] = 0;
    }

    // Sort items based on their ratio
    sortItems(items, n);

    for (int i = 0; i < n; i++) {
        if (totalWeight + items[i].weight <= W) {
            totalWeight += items[i].weight;
            totalValue += items[i].value;
            taken[i] = 1;
        }
    }

    printf("Maximum value in knapsack (discrete): %d\n", totalValue);
}

// Function to solve the continuous knapsack problem using a greedy approach
void fractionalKnapsack(int W, int n, Item items[]) {
```

```
    double totalValue = 0.0;
    int totalWeight = 0;

    // Sort items based on their ratio
    sortItems(items, n);

    for (int i = 0; i < n; i++) {
        if (totalWeight + items[i].weight <= W) {
            totalWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            int remainingWeight = W - totalWeight;
            totalValue += items[i].value * ((double)remainingWeight / items[i].weight);
            break;
        }
    }

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

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

    Item items[n];
    printf("Enter the values and weights of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &items[i].value, &items[i].weight);
        items[i].ratio = (double)items[i].value / items[i].weight;
    }
```

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

    // Solve discrete knapsack problem
    discreteKnapsack(W, n, items);

    // Solve fractional knapsack problem
    fractionalKnapsack(W, n, items);

    return 0;
}
```

Prog 8

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

// Function to print a subset
void printSubset(int arr[], int subset[], int subsetSize) {
    printf("{ ");
    for (int i = 0; i < subsetSize; i++) {
        printf("%d ", subset[i]);
    }
    printf("}\n");
}

// Recursive function to find and print subsets with the given sum
void findSubsetsWithSum(int arr[], int n, int sum, int subset[], int subsetSize) {
    // Base case
```

```
    if (sum == 0) {
        printSubset(arr, subset, subsetSize);
        return;
    }
    if (n == 0 || sum < 0) return;

    // Exclude the last element and recurse
    findSubsetsWithSum(arr, n-1, sum, subset, subsetSize);

    // Include the last element in the subset and recurse
    subset[subsetSize] = arr[n-1];
    findSubsetsWithSum(arr, n-1, sum-arr[n-1], subset, subsetSize + 1);
}

int main() {
    int n, d;

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

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

    printf("Enter the target sum: ");
    scanf("%d", &d);

    int subset[n]; // Array to hold the current subset
    printf("Subsets with the given sum:\n");
```

```
    findSubsetsWithSum(arr,  n, d, subset, 0);

    return 0;
}
```

Prog 9:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.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 the found minimum  element with the first element
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

// Function to print the array
void printArray(int  arr[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        printf("%d ", arr[i]);
        if ((i + 1) % 20 == 0) // Print 20 elements per line for better readability
            printf("\n");
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of elements (n > 5000): ");
    scanf("%d", &n);

    if (n <= 5000) {
        printf("Number of elements must be greater than 5000.\n");
        return 1;
    }

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Generate random integers and fill the array
    srand(time(NULL)); // Seed for random number generation
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Random integers between 0 and 9999
    }

    // Measure the time taken for sorting
    clock_t start = clock();
```

```
    selectionSort(arr, n);
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    // Print the sorted array
    printf("Sorted array:\n");
    printArray(arr, n);

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

Prog 10

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

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

// Partition function for Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

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

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
        if ((i + 1) % 20 == 0) // Print 20 elements per line for better readability
            printf("\n");
    }
    printf("\n");
}

int main() {
```

```
int n;
printf("Enter the number of elements (n > 5000): ");
scanf("%d", &n);

if (n <= 5000) {
   printf("Number of elements must be greater than 5000.\n");
   return 1;
}

int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {
   printf("Memory allocation failed.\n");
   return 1;
}

// Generate random integers and fill the array
srand(time(NULL)); // Seed for random number generation
for (int i = 0; i < n; i++) {
   arr[i] = rand() % 10000; // Random integers between 0 and 9999
}

// Measure the time taken for sorting
clock_t start = clock();
quickSort(arr, 0, n - 1);
clock_t end = clock();

double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

// Print the sorted array
printf("Sorted array:\n");
```

```
    printArray(arr, n);

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

Prog 11

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

// Function to merge two subarrays of arr[]
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

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

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

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
```

```
        }
    }

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

    free(L);
    free(R);
}

// Function to implement Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
        if ((i + 1) % 20 == 0) // Print 20 elements per line for readability
            printf("\n");
    }
    printf("\n");
}

int main() {
```

```
int n;
printf("Enter the number of elements (n > 5000): ");
scanf("%d", &n);

if (n <= 5000) {
   printf("Number of elements must be greater than 5000.\n");
   return 1;
}

int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {
   printf("Memory allocation failed.\n");
   return 1;
}

srand(time(NULL));
for (int i = 0; i < n; i++) {
   arr[i] = rand() % 10000; // Random integers between 0 and 9999
}

clock_t start = clock();
mergeSort(arr, 0, n - 1);
clock_t end = clock();

double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

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

free(arr);
```

```
    return 0;
}
```

Prog 12

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

#define MAX 20

int board[MAX][MAX];
int solutionCount  = 0;

// Function to print the chessboard
void printBoard(int  n) {
    printf("Solution  %d:\n", ++solutionCount);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == 1)
                printf("  Q ");
            else
                printf("  . ");
        }
        printf("\n");
    }
    printf("\n");
}

// Check if it's safe to place a queen at board[row][col]
bool isSafe(int board[MAX][MAX],  int row, int col, int n) {
    int i, j;
```

```
    // Check the same column
    for (i = 0; i < row; i++)
        if (board[i][col] == 1)
            return false;

    // Check upper left diagonal
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;

    // Check upper right diagonal
    for (i = row, j = col; i >= 0 && j < n; i--, j++)
        if (board[i][j] == 1)
            return false;

    return true;
}

// Recursive function to solve the N-Queens problem
void solveNQueens(int board[MAX][MAX], int row, int n) {
    if (row >= n) {
        printBoard(n);
        return;
    }

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

            solveNQueens(board, row + 1, n);
```

```
            board[row][col] = 0; // Backtrack
        }
    }
}

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

    if (n <= 0 || n > MAX) {
        printf("Number of queens must be between 1 and %d.\n", MAX);
        return 1;
    }

    // Initialize the board with 0
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            board[i][j] = 0;

    solveNQueens(board, 0, n);

    if (solutionCount == 0) {
        printf("No solutions exist.\n");
    }

    return 0;
}
```