Prog 1:

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

#define V 5 // Number of vertices

// Structure to represent an edge
typedef struct {
   int u, v, weight;
} Edge;

// Function to find the parent of a vertex in the disjoint set
int find(int parent[], int i) {
   if (parent[i] == i)
      return i;
   return find(parent, parent[i]);
}

// Function to perform union of two sets in the disjoint set
void unionSet(int parent[], int rank[], int x, int y) {
   int xroot = find(parent, x);
   int yroot = find(parent, y);

   if (rank[xroot] < rank[yroot])
      parent[xroot] = yroot;
   else if (rank[xroot] > rank[yroot])
      parent[yroot] = xroot;
   else {
      parent[yroot] = xroot;
      rank[xroot]++;
   }
}

// Function to implement Kruskal's algorithm
void kruskal(Edge edges[], int numVertices) {
   int parent[numVertices], rank[numVertices];
   int i, e = 0;
   int minimumCost = 0;

   // Initialize the disjoint set
   for (i = 0; i < numVertices; i++) {
      parent[i] = i;
      rank[i] = 0;
   }

   // Sort the edges in non-decreasing order of their weights
   for (i = 0; i < numVertices - 1; i++) {
      int min = i;
      for (int j = i + 1; j < numVertices; j++) {
         if (edges[j].weight < edges[min].weight)
            min = j;
      }
      Edge temp = edges[i];
      edges[i] = edges[min];
      edges[min] = temp;
   }

   // Find the MST
```

```
    for (i = 0; i < numVertices - 1; i++) {
        int x = find(parent, edges[i].u);
        int y = find(parent, edges[i].v);

        if (x != y) {
            minimumCost += edges[i].weight;
            printf("%d - %d\n", edges[i].u, edges[i].v);
            unionSet(parent, rank, x, y);
        }
    }

    printf("Minimum Cost: %d\n", minimumCost);
}

int main() {
    Edge edges[] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}};

    kruskal(edges, V);

    return 0;
}
```

Prog 2:

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

#define V 5

int graph[V][V] = {
    {0, 2, 3, 1},
    {2, 0, 4,0},
    {3,4, 0,5},
    {1, 0, 5, 0}
};

int visited[V] = {0};
int parent[V];
int totalCost = 0; // To store the total cost of the MST

int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void primMST() {
    int key[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        parent[i] = -1;
    }
```

```
    key[0] = 0;

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

        printf("%d - %d\n", parent[u], u);
        totalCost += key[u]; // Add the edge weight to the total cost

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

int main() {
    primMST();
    printf("Total cost of MST: %d\n", totalCost);
    return 0;
}
```

Prog 3

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

#define V 4 // Number of vertices

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

    // Create a copy of the graph
    int dist[V][V];
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Apply Floyd's algorithm
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the shortest paths
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            if (dist[i][j] == 999) {
```

```
                printf("INF ");
            } else {
                printf("%d ", dist[i][j]);
            }
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = {
        {999,8,4,999},
        {999, 999, 1, 999},
        {4, 999, 999, 999},
        {999, 2, 9, 999}
    };

    floyd(graph);

    return 0;
}
```

Prog 3b

```
#include <stdio.h>

#define V 4 // Number of vertices

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

    // Create a copy of the graph
    int transitiveClosure[V][V];
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            transitiveClosure[i][j] = graph[i][j];
        }
    }

    // Apply Warshal's algorithm
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                transitiveClosure[i][j] = transitiveClosure[i][j] || (transitiveClosure[i][k] &&
transitiveClosure[k][j]);
            }
        }
    }
```

```
    // Print the transitive closure
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            if (transitiveClosure[i][j]) {
                printf("1 ");
            } else {
                printf("0 ");
            }
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = {
        {1, 1, 0, 0},
        {0, 1, 1, 0},
        {0, 0, 1, 1},
        {0, 0, 0, 1}
    };

    warshal(graph);

    return 0;
}
```

Prog 4

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

#define V 5 // Number of vertices

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

    // Initialize distances and visited array
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }

    dist[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        int min = INT_MAX;
        int minIndex;

        // Find the vertex with minimum distance
        for (int j = 0; j < V; j++) {
```

```
            if (!visited[j] && dist[j] < min) {
                min = dist[j];
                minIndex = j;
            }
        }

        visited[minIndex] = 1;

        // Update distances of adjacent vertices
        for (int j = 0; j < V; j++) {
            if (!visited[j] && graph[minIndex][j] && dist[minIndex] + graph[minIndex][j] < dist[j]) {
                dist[j] = dist[minIndex] + graph[minIndex][j];
            }
        }
    }

    // Print the shortest distances
    printf("Vertex\tDistance\n");
    for (int i = 0; i < V; i++) {
        printf("%d\t%d\n", i, dist[i]);
    }
}

int main() {
    int graph[V][V] = {
        {0, 4, 0, 0, 0},
        {4, 0, 8, 0, 0},
        {0, 8, 0, 7, 0},
        {0, 0, 7, 0, 9},
        {0, 0, 0, 9, 0}
    };

    int src = 0;

    dijkstra(graph, src);

    return 0;
}
```

Prog 5

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

#define V 5 // Number of vertices

// Structure to represent a graph
typedef struct {
   int V;
   int *adj;
} Graph;

// Function to create a graph
Graph* createGraph(int V) {
   Graph* graph = (Graph*)malloc(sizeof(Graph));
   graph->V = V;
   graph->adj = (int*)malloc(V * sizeof(int));
   for (int i = 0; i < V; i++) {
      graph->adj[i] = 0;
   }
   return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int u, int v) {
   graph->adj[u] = v;
}

// Function to perform DFS
void DFS(Graph* graph, int v, int* visited, int* stack) {
   visited[v] = 1;
   for (int i = 0; i < graph->V; i++) {
      if (graph->adj[v] == i && !visited[i]) {
         DFS(graph, i, visited, stack);
      }
   }
   stack[graph->V - 1] = v;
   graph->V--;
}

// Function to perform Topological Sort
void TopologicalSort(Graph* graph) {
   int* visited = (int*)malloc(graph->V * sizeof(int));
   int* stack = (int*)malloc(graph->V * sizeof(int));
   for (int i = 0; i < graph->V; i++) {
      visited[i] = 0;
   }
   for (int i = 0; i < graph->V; i++) {
      if (!visited[i]) {
         DFS(graph, i, visited, stack);
      }
   }
   printf("Topological Order: ");
   for (int i = 0; i < graph->V; i++) {
      printf("%d ", stack[i]);
   }
   printf("\n");
}
```

```
int main() {
   Graph* graph = createGraph(5);
   addEdge(graph, 0, 1);
   addEdge(graph, 1, 2);
   addEdge(graph, 2, 3);
   addEdge(graph, 3, 4);
   TopologicalSort(graph);
   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() {
```

```c
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() {
```

```c
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;
}
```