INSERTION SORT

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j, pass = 1;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;

        // Print array after each pass
        printf("Pass %d: ", pass++);
        for (int k = 0; k < n; k++) {
            printf("%d ", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int arr[100], n;

    // Taking array size input
    printf("Enter number of elements: ");
    scanf("%d", &n);

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

    // Printing original array
    printf("\nOriginal array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n\n");

    // Performing insertion sort
    insertionSort(arr, n);

    // Printing final sorted array
    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

SELECTION SORT

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp, pass = 1;

    for (i = 0; i < n - 1; i++) {
        min_idx = i;

        // Find the index of the minimum element
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element with the current element
        if (min_idx != i) {
            temp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = temp;
        }

        // Print array after each pass
        printf("Pass %d: ", pass++);
        for (int k = 0; k < n; k++) {
            printf("%d ", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int arr[100], n;

    // Taking array size input
    printf("Enter number of elements: ");
    scanf("%d", &n);

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

    // Printing original array
    printf("\nOriginal array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n\n");

    // Performing selection sort
    selectionSort(arr, n);

    // Printing final sorted array
```

```
    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

QUICK SORT

```
#include <stdio.h>

int count = 0; // Global counter for number of quick_sort calls

int divide(int a[], int p, int r)
{
    int x, j, i, temp;

    x = a[r];
    i = p - 1;

    for (j = p; j <= r - 1; j++)
    {
        if (a[j] <= x)
        {
            i++;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    temp = a[i + 1];
    a[i + 1] = a[r];
    a[r] = temp;

    return i + 1;
}

void quick_sort(int a[], int p, int r)
{
    count++; // Increment call counter

    int q;
    if (p < r)
    {
        q = divide(a, p, r);
        quick_sort(a, p, q - 1);
        quick_sort(a, q + 1, r);
    }
}

int main()
{
```

```
    int s, a[50], i;

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

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

    quick_sort(a, 0, s - 1);

    printf("\nArray after Quick Sort: ");
    for (i = 0; i < s; i++)
    {
        printf("%d ", a[i]);
    }

    printf("\n\nNumber of calls to quick_sort(): %d\n", count);

    return 0;
}
```

MERGE SORT

```
#include <stdio.h>

int a[50], b[50];
int mergeSortCalls = 0; // Global counter

void merge(int low, int mid, int high)
{
    int h = low, i = low, j = mid + 1, k;

    while (h <= mid && j <= high)
    {
        if (a[h] <= a[j])
        {
            b[i++] = a[h++];
        }
        else
        {
            b[i++] = a[j++];
        }
    }

    if (h > mid)
    {
        for (k = j; k <= high; k++)
        {
            b[i++] = a[k];
        }
    }
    else
```

```
    {
        for (k = h; k <= mid; k++)
        {
            b[i++] = a[k];
        }
    }

    for (k = low; k <= high; k++)
    {
        a[k] = b[k];
    }
}

void merge_sort(int low, int high)
{
    mergeSortCalls++; // Count each time merge_sort is called

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

        merge_sort(low, mid);
        merge_sort(mid + 1, high);

        merge(low, mid, high);
    }
}

int main()
{
    int s, i;

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

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

    merge_sort(0, s - 1);

    printf("\nArray after Merge Sort: ");
    for (i = 0; i < s; i++)
    {
        printf("%d\t", a[i]);
    }

    printf("\n\nNumber of calls to merge_sort(): %d\n", mergeSortCalls);

    return 0;
}
```

PRIMS ALGORITHM

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

#define MAX 100

int main() {
    int n;
    int cost[MAX][MAX];
    int visited[MAX] = {0};
    int i, j, min, u, v;
    int ne = 1;  // number of edges included in MST
    int min_cost = 0;

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

    printf("Enter the cost adjacency matrix (0 for no edge):\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX; // treat 0 as no edge
        }
    }

    visited[0] = 1; // Start from vertex 0

    printf("\nEdges in the Minimum Spanning Tree:\n");
    while (ne < n) {
        min = INT_MAX;

        for (i = 0; i < n; i++) {
            if (visited[i]) {
                for (j = 0; j < n; j++) {
                    if (!visited[j] && cost[i][j] < min) {
                        min = cost[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }

        printf("Edge %d: (%d -> %d) cost = %d\n", ne, u, v, min);
        visited[v] = 1;
        min_cost += min;
        ne++;
    }

    printf("\nMinimum cost of the spanning tree = %d\n", min_cost);

    return 0;
}
```

KRUSKALS ALGORITHM

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

#define MAX 100

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

// Compare function for qsort
int compare(const void *a, const void *b) {
    return ((struct Edge *)a)->weight - ((struct Edge *)b)->weight;
}

int parent[MAX];

// Find parent (with path compression)
int find(int i) {
    while (parent[i] != i)
        i = parent[i];
    return i;
}

// Union operation
void union_set(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}

int main() {
    int n, e;
    struct Edge edgeList[MAX];

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

    printf("Enter edges (u v weight):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d%d%d", &edgeList[i].u, &edgeList[i].v, &edgeList[i].weight);
    }

    // Initialize parent array
    for (int i = 0; i < n; i++)
        parent[i] = i;

    // Sort edges by weight
    qsort(edgeList, e, sizeof(struct Edge), compare);

    int count = 0;       // number of edges in MST
    int minCost = 0;

    printf("\nEdges in the Minimum Spanning Tree:\n");
```

```
    for (int i = 0; i < e && count < n - 1; i++) {
        int u = edgeList[i].u;
        int v = edgeList[i].v;
        int w = edgeList[i].weight;

        if (find(u) != find(v)) {
            union_set(u, v);
            printf("Edge %d: (%d -> %d) cost = %d\n", count + 1, u, v, w);
            minCost += w;
            count++;
        }
    }

    printf("\nMinimum cost of the spanning tree = %d\n", minCost);
    return 0;
}
```

0/1 KNAPSACK

```
#include <stdio.h>

#define MAX 100

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

// Knapsack function
int knapsack(int n, int W, int weight[], int profit[], int x[]) {
    int dp[MAX][MAX];

    // Fill dp table
    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 (weight[i] <= w)
                dp[i][w] = max(profit[i] + dp[i - 1][w - weight[i]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    // Print DP Matrix
    printf("\nDP Matrix:\n");
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            printf("%3d ", dp[i][w]);
        }
        printf("\n");
    }

    // Backtrack to find solution vector
    int i = n, k = W;
    while (i > 0 && k > 0) {
```

```
        if (dp[i][k] != dp[i - 1][k]) {
            x[i] = 1;  // item i is included
            k = k - weight[i];
        } else {
            x[i] = 0;
        }
        i--;
    }

    return dp[n][W]; // Return total profit
}

int main() {
    int n, W;
    int weight[MAX], profit[MAX], x[MAX] = {0};

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

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

    printf("Enter profits of items:\n");
    for (int i = 1; i <= n; i++)
        scanf("%d", &profit[i]);

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

    int totalProfit = knapsack(n, W, weight, profit, x);

    // Print solution vector
    printf("\nSolution Vector (1 = included):\n");
    for (int i = 1; i <= n; i++) {
        printf("Item %d: %d\n", i, x[i]);
    }

    printf("\nTotal Profit Earned: %d\n", totalProfit);

    return 0;
}
```

LONGEST COMMON SUBSEQUENCE

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

#define MAX 100

// Function to find LCS
void LCS(char X[], char Y[], int m, int n) {
    int dp[MAX][MAX];      // DP matrix
    char arrow[MAX][MAX];  // To store directions: '↖' (diagonal), '↑', or '←'

    // Fill the matrix
```

```
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
                arrow[i][j] = ' ';
            }
            else if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                arrow[i][j] = 'D'; // Diagonal ↖
            }
            else if (dp[i - 1][j] >= dp[i][j - 1]) {
                dp[i][j] = dp[i - 1][j];
                arrow[i][j] = 'U'; // Up ↑
            }
            else {
                dp[i][j] = dp[i][j - 1];
                arrow[i][j] = 'L'; // Left ←
            }
        }
    }

    // Print the DP matrix with arrows
    printf("\nDP Matrix with Arrows:\n");
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (arrow[i][j] == 'D')
                printf("↖%2d ", dp[i][j]);
            else if (arrow[i][j] == 'U')
                printf("↑%2d ", dp[i][j]);
            else if (arrow[i][j] == 'L')
                printf("←%2d ", dp[i][j]);
            else
                printf(" %2d ", dp[i][j]);
        }
        printf("\n");
    }

    // Backtrack to find LCS
    int i = m, j = n;
    char lcs[MAX];
    int index = dp[m][n];
    lcs[index] = '\0'; // End of string

    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[--index] = X[i - 1];
            i--;
            j--;
        }
        else if (dp[i - 1][j] > dp[i][j - 1])
            i--;
        else
            j--;
    }

    printf("\nLength of LCS: %d\n", dp[m][n]);
    printf("LCS: %s\n", lcs);
}

int main() {
```

```
    char X[MAX], Y[MAX];

    printf("Enter first string: ");
    scanf("%s", X);
    printf("Enter second string: ");
    scanf("%s", Y);

    int m = strlen(X);
    int n = strlen(Y);

    LCS(X, Y, m, n);

    return 0;
}
```

DIJKSTRAS ALGORITHM

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

#define MAX 100
#define INF 9999

int D[MAX], Pi[MAX], visited[MAX];
int cost[MAX][MAX];
int n; // number of vertices

// Function to find the vertex with minimum distance
int findMinVertex() {
    int min = INF, min_index = -1;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && D[i] < min) {
            min = D[i];
            min_index = i;
        }
    }
    return min_index;
}

// Dijkstra's Algorithm
void dijkstra(int src) {
    for (int i = 0; i < n; i++) {
        D[i] = INF;
        Pi[i] = -1;
        visited[i] = 0;
    }
    D[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = findMinVertex();
        if (u == -1) break; // all remaining vertices are unreachable
        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            if (!visited[v] && cost[u][v] && D[u] + cost[u][v] < D[v]) {
```

```
                D[v] = D[u] + cost[u][v];
                Pi[v] = u;
            }
        }
    }
}

// Function to print path from source to a vertex
void printPath(int vertex, int src) {
    if (vertex == src) {
        printf("%d", src);
        return;
    }
    if (Pi[vertex] == -1) {
        printf("No path");
        return;
    }
    printPath(Pi[vertex], src);
    printf(" -> %d", vertex);
}

int main() {
    int src;

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

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

    printf("Enter source vertex (0 to %d): ", n - 1);
    scanf("%d", &src);

    dijkstra(src);

    // Print Distance and Predecessor vectors
    printf("\nVertex\tD (Distance)\tPi (Predecessor)\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t", i, D[i]);
        if (Pi[i] == -1)
            printf("NIL\n");
        else
            printf("%d\n", Pi[i]);
    }

    // Print paths from source to all vertices
    printf("\nPaths from Source %d:\n", src);
    for (int i = 0; i < n; i++) {
        printf("Path to %d: ", i);
        if (D[i] == INF)
            printf("No path\n");
        else {
            printPath(i, src);
            printf(" (Cost: %d)\n", D[i]);
```

```
        }
    }

    return 0;
}
```

FLOYD WARSHALL ALGORITHM

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

#define INF 99999
#define MAX 100

int dist[MAX][MAX], pred[MAX][MAX];
int n;

// Function to print the path from i to j
void printPath(int i, int j) {
    if (i == j) {
        printf("%d", i);
        return;
    } else if (pred[i][j] == -1) {
        printf("No path");
        return;
    } else {
        printPath(i, pred[i][j]);
        printf(" -> %d", j);
    }
}

void floydWarshall() {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    pred[i][j] = pred[k][j];
                }
            }
        }
    }
}

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

    printf("Enter the cost adjacency matrix (0 if no edge, and 0 for self-loops):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int val;
            scanf("%d", &val);

            if (i != j && val == 0)
                dist[i][j] = INF;
```

```
            else
                dist[i][j] = val;

            if (i == j || val != 0)
                pred[i][j] = i;
            else
                pred[i][j] = -1;
        }
    }

    floydWarshall();

    // Print Final Distance Matrix
    printf("\nFinal Distance (D) Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][j] == INF)
                printf("INF ");
            else
                printf("%3d ", dist[i][j]);
        }
        printf("\n");
    }

    // Print Predecessor Matrix
    printf("\nPredecessor (Π) Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (pred[i][j] == -1)
                printf(" - ");
            else
                printf("%2d ", pred[i][j]);
        }
        printf("\n");
    }

    // Print Path Between Every Pair
    printf("\nShortest Paths Between All Pairs:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("Path from %d to %d: ", i, j);
            if (dist[i][j] == INF)
                printf("No path\n");
            else {
                printPath(i, j);
                printf(" (Cost = %d)\n", dist[i][j]);
            }
        }
    }

    return 0;
}
```

N QUEEN PROBELM

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

#define MAX 20

int board[MAX], count = 0, calls = 0;

int isSafe(int row, int col) {
    for (int i = 1; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row)) {
            return 0;
        }
    }
    return 1;
}

void printSolution(int n) {
    // 1D Format
    printf("\n1D Solution: ");
    for (int i = 1; i <= n; i++) {
        printf("%d ", board[i]);
    }

    // 2D Format
    printf("\n2D Board:\n");
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (board[i] == j)
                printf(" Q ");
            else
                printf(" . ");
        }
        printf("\n");
    }
    printf("\n");
}

void queen(int row, int n) {
    calls++;
    for (int col = 1; col <= n; col++) {
        if (isSafe(row, col)) {
            board[row] = col;
            if (row == n) {
                count++;
                printSolution(n);
            } else {
                queen(row + 1, n);
            }
        }
    }
}

int main() {
```

```
    int n;
    printf("Enter the value of N: ");
    scanf("%d", &n);

    queen(1, n);

    printf("\nTotal number of solutions: %d", count);
    printf("\nTotal number of recursive calls to queen(): %d\n", calls);

    return 0;
}
```

SUM OF SUBSETS PROBLEM

```
#include <stdio.h>

int totalCalls = 0;

void sumOfSubsets(int set[], int subset[], int n, int subsetSize, int total, int index, int targetSum) {
    totalCalls++;

    if (total == targetSum) {
        // Print the current subset
        printf("Subset: ");
        for (int i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    if (index >= n || total > targetSum) {
        return;
    }

    // Include the current element
    subset[subsetSize] = set[index];
    sumOfSubsets(set, subset, n, subsetSize + 1, total + set[index], index + 1, targetSum);

    // Exclude the current element
    sumOfSubsets(set, subset, n, subsetSize, total, index + 1, targetSum);
}

int main() {
    int set[20], n, targetSum;
    int subset[20];

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

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

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

    printf("\nValid subsets:\n");
    sumOfSubsets(set, subset, n, 0, 0, 0, targetSum);

    printf("\nTotal number of recursive calls: %d\n", totalCalls);

    return 0;
}
```

KMP METHOD

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

// Function to compute LPS array
void computeLPSArray(char* pat, int M, int lps[]) {
    int len = 0;
    lps[0] = 0; // lps[0] is always 0

    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];  // Don't increment i here
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// KMP search function
void KMPSearch(char* pat, char* txt) {
    int M = strlen(pat);
    int N = strlen(txt);
    int matchCount = 0;

    int lps[M];
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]

    printf("\nPattern found at indices: ");
    while (i < N) {
        if (pat[j] == txt[i]) {
            i++;
            j++;
        }
```

```
        if (j == M) {
            printf("%d ", i - j);
            matchCount++;
            j = lps[j - 1];  // Continue searching
        } else if (i < N && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }
    }

    printf("\nTotal matches found: %d\n", matchCount);
}

int main() {
    char txt[100], pat[100];

    printf("Enter the text: ");
    scanf(" %[^\n]", txt);  // To read string with spaces

    printf("Enter the pattern to search: ");
    scanf(" %[^\n]", pat);

    KMPSearch(pat, txt);

    return 0;
}
```

RABIN KARP ALGORITHM

```
#include <stdio.h>
#include <string.h>
#define d 256  // Number of characters in input alphabet

void RabinKarpSearch(char pat[], char txt[], int q) {
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;     // hash value for pattern
    int t = 0;     // hash value for text
    int h = 1;
    int matchCount = 0;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate hash value for pattern and first window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    printf("\nPattern found at indices: ");
    // Slide the pattern over text
    for (i = 0; i <= N - M; i++) {
```

```
        // If hash values match, check for characters one by one
        if (p == t) {
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j])
                    break;
            }

            if (j == M) {
                printf("%d ", i);
                matchCount++;
            }
        }

        // Calculate hash value for next window of text
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // Make sure hash is positive
            if (t < 0)
                t = (t + q);
        }
    }

    printf("\nTotal matches found: %d\n", matchCount);
}

int main() {
    char txt[100], pat[100];
    int q = 101; // A prime number

    printf("Enter the text: ");
    scanf(" %[^\n]", txt);

    printf("Enter the pattern to search: ");
    scanf(" %[^\n]", pat);

    RabinKarpSearch(pat, txt, q);

    return 0;
}
```