Linear Search-

```cpp
#include <iostream>
using namespace std;

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int n, key;

    cout << "Enter the number of elements in the array: ";
    cin >> n;

    int arr[n];

    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "Enter the element to search for: ";
    cin >> key;

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

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found in the array" << endl;
    }

    return 0;
}
```

Binary Search Recursive-

```
#include <iostream>
using namespace std;

int binarySearchRecursive(int arr[], int left, int right, int key) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == key) {
            return mid;
        }

        if (arr[mid] < key) {
            return binarySearchRecursive(arr, mid + 1, right, key);
        }

        else {
            return binarySearchRecursive(arr, left, mid - 1, key);
        }
    }

    return -1;
}

int main() {
    int n, key;

    cout << "Enter the number of elements in the array: ";
    cin >> n;

    int arr[n];

    cout << "Enter the elements of the array (sorted in ascending order):
";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "Enter the element to search for: ";
```

```
    cin >> key;

    int result = binarySearchRecursive(arr, 0, n - 1, key);

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found in the array" << endl;
    }

    return 0;
}
```

Merge Sort-

```
#include <iostream>
using namespace std;

int mergeSortPassCount = 0;

void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    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;
```

```
    int j = 0;
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
        printArray(arr, r + 1);  // Print array after each merge step
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
        printArray(arr, r + 1);  // Print array after adding remaining
elements of L
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
        printArray(arr, r + 1);  // Print array after adding remaining
elements of R
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l >= r) {
        return;
    }
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
    mergeSortPassCount++;
```

```
    cout << "Pass " << mergeSortPassCount << ": ";
    printArray(arr, r + 1);
    cout << "Number of arrays processed: " << mergeSortPassCount << endl;
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
```

Quick sort-

```
#include <iostream>
using namespace std;

int quickSortPassCount = 0;

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

```
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]);
        }
        printArray(arr, high + 1);  // Print array after each swap
    }
    swap(&arr[i + 1], &arr[high]);
    printArray(arr, high + 1);  // Print array after swapping with pivot
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSortPassCount++;
        cout << "Pass " << quickSortPassCount << ": ";
        printArray(arr, high + 1);
        cout << "Number of arrays processed: " << quickSortPassCount <<
endl;

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Initial array: ";
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Kruskals-

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

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

// Comparator to sort edges based on their weight
bool compareEdge(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

// Disjoint Set Union (DSU) for cycle detection
struct DSU {
    vector<int> parent, rank;

    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY])
                parent[rootX] = rootY;
            else if (rank[rootX] > rank[rootY])
                parent[rootY] = rootX;
```

```
            else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

void kruskalMST(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end(), compareEdge);

    DSU dsu(V);
    vector<Edge> mst;

    cout << "Edges in MST (Kruskal's):\n";
    int mstCost = 0;

    for (Edge& edge : edges) {
        if (dsu.find(edge.src) != dsu.find(edge.dest)) {
            mst.push_back(edge);
            mstCost += edge.weight;
            dsu.unite(edge.src, edge.dest);

            cout << edge.src << " -- " << edge.dest << " == " <<
edge.weight << endl;
        }
    }

    cout << "Total MST cost: " << mstCost << endl;
}

int main() {
    int V = 6;  // Number of vertices
    vector<Edge> edges = {
        {0, 1, 4}, {0, 2, 4}, {1, 2, 2}, {1, 3, 5},
        {2, 3, 5}, {3, 4, 7}, {4, 5, 9}, {3, 5, 6}
    };

    kruskalMST(V, edges);
    return 0;
```

```
}
```

Prims-

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
#include <climits>
using namespace std;

typedef pair<int, int> Pair; // (weight, vertex)

void primMST(int V, vector<vector<Pair>>& adj) {
    vector<int> key(V, INT_MAX);  // Minimum weight to include a vertex in
MST
    vector<bool> inMST(V, false); // To track vertices included in MST
    vector<int> parent(V, -1);    // Store MST structure

    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;

    key[0] = 0;  // Start from vertex 0
    pq.push({0, 0});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        inMST[u] = true;

        for (auto& [weight, v] : adj[u]) {
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.push({key[v], v});
                parent[v] = u;
            }
        }
    }

    cout << "Edges in MST (Prim's):\n";
```

```
    int mstCost = 0;
    for (int i = 1; i < V; ++i) {
        cout << parent[i] << " -- " << i << " == " << key[i] << endl;
        mstCost += key[i];
    }

    cout << "Total MST cost: " << mstCost << endl;
}

int main() {
    int V = 6;  // Number of vertices
    vector<vector<Pair>> adj(V);

    adj[0].push_back({4, 1});
    adj[0].push_back({4, 2});
    adj[1].push_back({4, 0});
    adj[1].push_back({2, 2});
    adj[1].push_back({5, 3});
    adj[2].push_back({4, 0});
    adj[2].push_back({2, 1});
    adj[2].push_back({5, 3});
    adj[3].push_back({5, 1});
    adj[3].push_back({5, 2});
    adj[3].push_back({7, 4});
    adj[3].push_back({6, 5});
    adj[4].push_back({7, 3});
    adj[4].push_back({9, 5});
    adj[5].push_back({6, 3});
    adj[5].push_back({9, 4});

    primMST(V, adj);
    return 0;
}
```

Dijkstra-

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>
#include <utility>
#include <climits>

using namespace std;

typedef pair<int, int> Pair; // (distance, vertex)

void dijkstra(int start, int V, vector<vector<Pair>>& adj) {
    vector<int> dist(V, INT_MAX);  // Distance from start to each node
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq; // Min-heap for
(distance, vertex)

    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        int currentDist = pq.top().first;
        pq.pop();

        if (currentDist > dist[u]) continue;  // Skip outdated pairs

        for (auto& [weight, v] : adj[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    // Print the shortest distances from the start node
    cout << "Shortest distances from node " << start << ":\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX)
            cout << "Node " << i << ": Unreachable\n";
        else
            cout << "Node " << i << ": " << dist[i] << "\n";
    }
}
```

```cpp
int main() {
    int V = 6;  // Number of vertices
    vector<vector<Pair>> adj(V);

    // Add edges (u, v, weight)
    adj[0].push_back({4, 1});
    adj[0].push_back({2, 2});
    adj[1].push_back({4, 0});
    adj[1].push_back({1, 2});
    adj[1].push_back({5, 3});
    adj[2].push_back({2, 0});
    adj[2].push_back({1, 1});
    adj[2].push_back({8, 3});
    adj[2].push_back({10, 4});
    adj[3].push_back({5, 1});
    adj[3].push_back({8, 2});
    adj[3].push_back({2, 4});
    adj[3].push_back({6, 5});
    adj[4].push_back({10, 2});
    adj[4].push_back({2, 3});
    adj[4].push_back({3, 5});
    adj[5].push_back({6, 3});
    adj[5].push_back({3, 4});

    int start = 0;  // Starting node
    dijkstra(start, V, adj);

    return 0;
}
```

Knapsack_Dynamic-

```cpp
#include <iostream>
#include <vector>
using namespace std;

int knapsackDP(int W, vector<int>& weights, vector<int>& values, int n) {
```

```
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w -
weights[i - 1]]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][W];
}

int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;
    cout << "Enter capacity of knapsack: ";
    cin >> W;

    vector<int> weights(n), values(n);
    cout << "Enter weight and value of each item:\n";
    for (int i = 0; i < n; ++i) {
        cout << "Item " << i + 1 << ": ";
        cin >> weights[i] >> values[i];
    }

    cout << "Maximum value (0/1 Knapsack): "
         << knapsackDP(W, weights, values, n) << endl;

    return 0;
}
```

Knapsack_Greedy-

```
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>

using namespace std;

struct Item {
    int weight, value;
    double valuePerWeight;
};

bool compare(Item a, Item b) {
    return a.valuePerWeight > b.valuePerWeight;
}

double fractionalKnapsack(int W, vector<Item>& items) {
    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;

    for (const auto& item : items) {
        if (W >= item.weight) {
            W -= item.weight;
            totalValue += item.value;
        } else {
            totalValue += item.valuePerWeight * W;
            break;
        }
    }

    return totalValue;
}

int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;
    cout << "Enter capacity of knapsack: ";
    cin >> W;

    vector<Item> items(n);
```

```
    cout << "Enter weight and value of each item:\n";
    for (int i = 0; i < n; ++i) {
        cout << "Item " << i + 1 << ": ";
        cin >> items[i].weight >> items[i].value;
        items[i].valuePerWeight = (double)items[i].value /
items[i].weight;
    }

    cout << "Maximum value (Fractional Knapsack): "
         << fractionalKnapsack(W, items) << endl;

    return 0;
}
```

Floyds-

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <limits.h>

using namespace std;

#define INF INT_MAX

void printMatrix(const vector<vector<int>> &dist, int V)
{
    cout << "Current distance matrix:\n";
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                cout << setw(5) << "INF";
            else
                cout << setw(5) << dist[i][j];
        }
        cout << "\n";
    }
```

```
    cout << "\n";
}

void floydWarshall(vector<vector<int>> &graph, int V)
{
    vector<vector<int>> dist = graph;

    cout << "Initial distance matrix:\n";
    printMatrix(dist, V);

    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] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }

        cout << "Distance matrix after including vertex " << k + 1 << " as
intermediate:\n";
        printMatrix(dist, V);
    }
}

int main()
{
    int V = 4;

    vector<vector<int>> graph = {
        {0, 3, INF, 7},
        {8, 0, 2, INF},
        {5, INF, 0, 1},
```

```
        {2, INF, INF, 0}};

    floydWarshall(graph, V);

    return 0;
}
```

Merge_Pattern_Backtracking-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <iomanip>

using namespace std;

int totalCost = INT_MAX;  // Store the minimum cost globally
vector<pair<int, int>> optimalSteps;  // Store the optimal merge steps

void displaySteps(const vector<pair<int, int>>& steps, const vector<int>&
initialFiles) {
    vector<int> currentFiles = initialFiles;
    cout << "\nDetailed steps for the optimal merge:\n";
    for (const auto& step : steps) {
        int file1 = currentFiles[step.first];
        int file2 = currentFiles[step.second];
        int mergedSize = file1 + file2;

        cout << "Merge files of size " << file1 << " and " << file2
             << " -> New merged file size: " << mergedSize << endl;

        // Remove the merged files and replace with the new merged file
        currentFiles.erase(currentFiles.begin() + step.second);
        currentFiles.erase(currentFiles.begin() + step.first);
        currentFiles.push_back(mergedSize);

        // Display current state of files
        cout << "Current file sizes: ";
```

```
        for (int size : currentFiles) cout << size << " ";
        cout << "\n";
    }
}

void backtrack(vector<int> files, int currentCost, vector<pair<int, int>>
steps) {
    if (files.size() == 1) {  // Base case: Only one file remains
        if (currentCost < totalCost) {
            totalCost = currentCost;
            optimalSteps = steps;
        }
        return;
    }

    for (int i = 0; i < files.size(); ++i) {
        for (int j = i + 1; j < files.size(); ++j) {
            // Choose files i and j to merge
            int mergedSize = files[i] + files[j];
            int newCost = currentCost + mergedSize;

            vector<int> newFiles = files;
            vector<pair<int, int>> newSteps = steps;

            // Replace files i and j with their merged size
            newFiles.erase(newFiles.begin() + j);
            newFiles.erase(newFiles.begin() + i);
            newFiles.push_back(mergedSize);
            newSteps.push_back({i, j});

            // Recurse with updated files and cost
            backtrack(newFiles, newCost, newSteps);
        }
    }
}

int main() {
    int n;
    cout << "Enter the number of files: ";
    cin >> n;
```

```
    vector<int> files(n);
    cout << "Enter the sizes of the files:\n";
    for (int i = 0; i < n; ++i) {
        cin >> files[i];
    }

    vector<pair<int, int>> steps;  // To store merge steps during
recursion
    backtrack(files, 0, steps);

    cout << "\nOptimal Merge Pattern Found!" << endl;
    cout << "Minimum Total Cost: " << totalCost << endl;

    displaySteps(optimalSteps, files);

    return 0;
}
```

4-queens-

```
#include <iostream>
#include <vector>
using namespace std;

// Function to print the current state of the board
void printBoard(vector<vector<int>>& board) {
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board.size(); j++) {
            if (board[i][j] == 1)
                cout << " Q ";
            else
                cout << " . ";
        }
        cout << endl;
    }
    cout << endl;
}
```

```cpp
// Function to check if placing a queen at board[row][col] is safe
bool isSafe(vector<vector<int>>& board, int row, int col) {
    int N = board.size();

    // Check left side of the row
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1)
            return false;
    }

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

    // Check lower diagonal on the left side
    for (int i = row, j = col; i < N && j >= 0; i++, j--) {
        if (board[i][j] == 1)
            return false;
    }

    return true;
}

// Function to solve the N-Queens problem using backtracking
bool solveNQueens(vector<vector<int>>& board, int col, int& solutionCount)
{
    int N = board.size();

    // Base case: If all queens are placed
    if (col >= N) {
        cout << "Solution " << ++solutionCount << ":\n";
        printBoard(board);
        return true;
    }

    bool foundSolution = false;

    // Try placing a queen in all rows of the current column
```

```
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1; // Place the queen
            solveNQueens(board, col + 1, solutionCount);
            board[i][col] = 0; // Backtrack
        }
    }

    return foundSolution;
}

int main() {
    int N;
    cout << "Enter the number of queens: ";
    cin >> N;

    vector<vector<int>> board(N, vector<int>(N, 0));
    int solutionCount = 0;

    solveNQueens(board, 0, solutionCount);

    if (solutionCount == 0) {
        cout << "No solution exists for " << N << "-Queens problem." <<
endl;
    } else {
        cout << "Total number of solutions: " << solutionCount << endl;
    }

    return 0;
}
```