Q1. Bubble Sort

📝 Question:

A warehouse system stores package IDs in the order they arrive.
To prepare for dispatch, the IDs must be sorted in ascending order.
Write a program using **Bubble Sort** to arrange the following IDs:
`[5, 4, 3, 2, 1]`

💻 Program:

```
#include <iostream>     // for input-output operations
using namespace std;

int main() {
    int arr[] = {5, 4, 3, 2, 1};   // initial package IDs
    int n = 5;                     // number of elements

    // Outer loop runs (n-1) times - number of passes
    for (int i = 0; i < n - 1; i++) {
        // Inner loop compares adjacent elements
        for (int j = 0; j < n - i - 1; j++) {
            // If current element is greater than next, swap them
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    cout << "Sorted Package IDs: ";
    // Display the sorted array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
```

```
}
```

Q2. Insertion Sort

📃 Question:

A warehouse system stores package IDs in the order they arrive.
To prepare for dispatch, the IDs must be sorted in ascending order.
Write a program using **Insertion Sort** to arrange the following IDs:
`[5, 4, 3, 2, 1]`

💻 Program:

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

int main() {
    int arr[] = {5, 4, 3, 2, 1};
    int n = 5;

    // Start from second element (as first is already "sorted")
    for (int i = 1; i < n; i++) {
        int key = arr[i];        // current element to insert
        int j = i - 1;           // index of previous element

        // Shift all greater elements one step ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // Insert key in correct position
        arr[j + 1] = key;
    }

    cout << "Sorted Package IDs: ";
```

```
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

Q3. Selection Sort

📃 Question:

A warehouse system stores package IDs in the order they arrive.
To prepare for dispatch, the IDs must be sorted in ascending order.
Write a program using **Selection Sort** to arrange the following IDs:
`[5, 4, 3, 2, 1]`

💻 Program:

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

int main() {
    int arr[] = {5, 4, 3, 2, 1};
    int n = 5;

    // Outer loop selects position to place smallest element
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;  // assume current index has smallest value

        // Find index of minimum element in remaining array
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }

        // Swap smallest element with current position
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
```

```
    }

    cout << "Sorted Package IDs: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

Q4. Linked List Creation and Display

📓 Question:

A hospital management system stores patient IDs in a linked list to maintain their admission order.
You are given the following sequence:
`111 → 123 → 124 → NULL`
Write a program to create and display this linked list.

💻 Program:

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

// Define structure for a node
struct Node {
    int data;      // stores patient ID
    Node* next;    // pointer to next node
};

int main() {
    // Create 3 nodes dynamically using new
    Node* head = new Node{111, nullptr};
    Node* second = new Node{123, nullptr};
    Node* third = new Node{124, nullptr};

    // Connect the nodes
    head->next = second;
```

```
    second->next = third;

    // Display linked list
    Node* temp = head;
    cout << "Patient IDs: ";
    while (temp != nullptr) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL";
    return 0;
}
```

Q5. Graph Representation (Adjacency List & Matrix)

📄 Question:

A social networking app wants to represent user connections as a graph.
Each user is a node and friendships are edges between them.
Create the **Adjacency List** and **Adjacency Matrix** representation.

💻 Program:

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

int main() {
    int V = 4; // Number of users/nodes

    // Adjacency List using vector of vectors
    vector<int> adj[4];
    adj[0] = {1, 2}; // user 0 connected to 1,2
    adj[1] = {0, 3}; // user 1 connected to 0,3
    adj[2] = {0};    // user 2 connected to 0
    adj[3] = {1};    // user 3 connected to 1
```

```
    cout << "Adjacency List:\n";
    for (int i = 0; i < V; i++) {
        cout << i << " -> ";
        for (int j : adj[i])
            cout << j << " ";
        cout << endl;
    }

    // Adjacency Matrix
    int matrix[4][4] = {0};
    matrix[0][1] = matrix[1][0] = 1;
    matrix[0][2] = matrix[2][0] = 1;
    matrix[1][3] = matrix[3][1] = 1;

    cout << "\nAdjacency Matrix:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++)
            cout << matrix[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

Q6. Binary Tree Implementation

📄 Question:

A university's examination system stores student roll numbers in a binary tree.
Given the structure:

```
      50
     /    \
   30      70
  / \     /
20  40  60
```

Write a program to create and display the binary tree.

💻 Program:

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

// Node structure for binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* create(int val) {
    Node* n = new Node;
    n->data = val;
    n->left = n->right = nullptr;
    return n;
}

// Inorder traversal (Left → Root → Right)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    // Construct binary tree as per question
    Node* root = create(50);
    root->left = create(30);
    root->right = create(70);
    root->left->left = create(20);
    root->left->right = create(40);
    root->right->left = create(60);
```

```
    cout << "Inorder Traversal of Tree: ";
    inorder(root);
    return 0;
}
```

Q7. Hashing

📓 Question:

An online library stores book IDs using the hash function:
`h(key) = key % table_size`.
Given keys `[1,2,3,4]` and table size 3, insert and display the hash table.

💻 Program:

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

int main() {
    int keys[] = {1, 2, 3, 4};
    int table_size = 3;
    int hash_table[3] = {-1, -1, -1}; // initialize with -1 (empty)

    // Insert keys using linear probing
    for (int i = 0; i < 4; i++) {
        int index = keys[i] % table_size; // hash function
        while (hash_table[index] != -1)   // if slot filled, probe
next
            index = (index + 1) % table_size;
        hash_table[index] = keys[i];
    }

    cout << "Final Hash Table:\n";
    for (int i = 0; i < table_size; i++)
        cout << i << " -> " << hash_table[i] << endl;
```

```
    return 0;
}
```

Q8. Graph Representation (City Roads)

📓 Question:

A city traffic control system represents roads between intersections as a graph. Create the **Adjacency Matrix** representation for the graph.

💻 Program:

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

int main() {
    // 4 intersections (nodes)
    int V = 4;

    // 1 means road exists between intersections
    int graph[4][4] = {
        {0, 1, 0, 1},
        {1, 0, 1, 1},
        {0, 1, 0, 0},
        {1, 1, 0, 0}
    };

    cout << "Adjacency Matrix for City Roads:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Q1 — Insert node in linked list *after* a particular element

```
// Question:
// Write a C++ program to insert the node in a linked list after a
particular element.

#include <iostream>
using namespace std;

// Node structure for singly linked list
struct Node {
    int data;       // stores value
    Node* next;     // pointer to next node
    Node(int v) : data(v), next(nullptr) {} // constructor
};

// Insert a node with value 'value' after the first node containing
'key'
void insertAfter(Node* head, int key, int value) {
    Node* curr = head;                  // start traversal from head
    while (curr != nullptr && curr->data != key) {
        curr = curr->next;              // move to next node until key
found or list ends
    }
    if (curr == nullptr) {
        cout << "Key " << key << " not found. Insertion failed.\n";
        return;                         // key not present
    }
    Node* newNode = new Node(value);    // allocate new node
    newNode->next = curr->next;         // new node points to node
after curr
    curr->next = newNode;               // curr now points to new node
-> insertion done
}

// Utility to print list
void printList(Node* head) {
    Node* t = head;
    while (t) {
```

```
        cout << t->data;
        if (t->next) cout << " -> ";
        t = t->next;
    }
    cout << " -> NULL\n";
}

int main() {
    // create sample list: 10 -> 20 -> 30
    Node* head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);

    cout << "Original list: ";
    printList(head);

    // insert 25 after 20
    insertAfter(head, 20, 25);

    cout << "After inserting 25 after 20: ";
    printList(head);

    return 0;
}
```

Q2 — Effective address calculation for 2D array (row-major & column-major)

```
// Question:
// Array X with rows indexed from -5 to 10 and columns from 5 to 20.
// Element size = 1 byte, base address = 1500.
// Find effective address of X[8][12] using row-major and column-major
layout.

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

```
int main() {
    int LR = -5;        // lower row bound
    int UR = 10;        // upper row bound
    int LC = 5;         // lower column bound
    int UC = 20;        // upper column bound
    int B = 1500;       // base address
    int w = 1;          // element size in bytes

    int i = 8;          // target row index
    int j = 12;         // target column index

    int M = UR - LR + 1; // number of rows
    int N = UC - LC + 1; // number of columns

    // Row-major formula: EA = B + [(i - LR) * N + (j - LC)] * w
    int EA_row = B + ((i - LR) * N + (j - LC)) * w;

    // Column-major formula: EA = B + [(j - LC) * M + (i - LR)] * w
    int EA_col = B + ((j - LC) * M + (i - LR)) * w;

    cout << "Row-major EA: " << EA_row << "\n"; // expected 1715
    cout << "Column-major EA: " << EA_col << "\n"; // expected 1625

    return 0;
}
```

Q3 — Convert upper triangular matrix elements into a linear array (C++)

```
// Question:
// Write a C++ program to convert the upper triangular elements of an
n x n matrix
// into a linear array (store row-wise: traverse row i, columns j >=
i).
```

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

int main() {
    // Example 4x4 matrix (only upper triangle stored)
    int n = 4;
    int mat[4][4] = {
        {1, 2, 3, 4},
        {0, 5, 6, 7},
        {0, 0, 8, 9},
        {0, 0, 0, 10}
    };

    // linear vector to store upper triangular elements
    vector<int> linear;

    // traverse rows i = 0..n-1, and columns j = i..n-1
    for (int i = 0; i < n; ++i) {
        for (int j = i; j < n; ++j) {
            linear.push_back(mat[i][j]); // append element
        }
    }

    // print linear array
    cout << "Upper triangular elements stored linearly: ";
    for (int val : linear) cout << val << " ";
    cout << "\n";
    return 0;
}
```

Q4 — Construct binary tree from inorder & preorder traversals (C++)

```cpp
// Question:
// Construct the binary tree using given traversals:
```

```
// Inorder:  [40, 20, 50, 10, 60, 30]
// Preorder: [10, 20, 40, 50, 30, 60]

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

// helper: find index of value in inorder array between l..r
int findIndex(int inorder[], int l, int r, int value) {
    for (int i = l; i <= r; ++i)
        if (inorder[i] == value) return i;
    return -1; // should not happen if inputs valid
}

// recursive builder using current preorder index passed by reference
Node* buildTree(int inorder[], int preorder[], int inStart, int inEnd,
int &preIndex) {
    if (inStart > inEnd) return nullptr;        // no nodes in this
subtree

    int rootVal = preorder[preIndex++];         // take next preorder
element as root
    Node* root = new Node(rootVal);             // create node

    if (inStart == inEnd) return root;          // leaf node

    int inIndex = findIndex(inorder, inStart, inEnd, rootVal); //
locate root in inorder

    // recursively build left and right subtrees
    root->left = buildTree(inorder, preorder, inStart, inIndex - 1,
preIndex);
    root->right = buildTree(inorder, preorder, inIndex + 1, inEnd,
```

```
preIndex);
    return root;
}

// inorder traversal to verify constructed tree
void inorderPrint(Node* root) {
    if (!root) return;
    inorderPrint(root->left);
    cout << root->data << " ";
    inorderPrint(root->right);
}

int main() {
    int inorder[] = {40, 20, 50, 10, 60, 30};
    int preorder[] = {10, 20, 40, 50, 30, 60};
    int n = 6;
    int preIndex = 0;

    Node* root = buildTree(inorder, preorder, 0, n - 1, preIndex);

    cout << "Inorder traversal of constructed tree: ";
    inorderPrint(root); // should print original inorder
    cout << "\n";
    return 0;
}
```

Q5 — Insert given keys into BST and print Preorder, Inorder, Postorder

```
// Question:
// Insert keys in order: 20, 30, 35, 27, 10, 15, 5, 19, 12, 14
// Print Preorder, Inorder, Postorder traversals.

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

```
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

// Insert value into BST (recursive)
Node* bstInsert(Node* root, int val) {
    if (root == nullptr) return new Node(val);       // new node if
spot found
    if (val < root->data) root->left = bstInsert(root->left, val);
    else root->right = bstInsert(root->right, val);
    return root;
}

// Traversals
void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    cout << r->data << " ";
    inorder(r->right);
}

void preorder(Node* r) {
    if (!r) return;
    cout << r->data << " ";
    preorder(r->left);
    preorder(r->right);
}

void postorder(Node* r) {
    if (!r) return;
    postorder(r->left);
    postorder(r->right);
    cout << r->data << " ";
}

int main() {
```

```
    int keys[] = {20, 30, 35, 27, 10, 15, 5, 19, 12, 14};
    Node* root = nullptr;

    // Insert keys in given order
    for (int k : keys) root = bstInsert(root, k);

    // Print traversals
    cout << "Inorder:  "; inorder(root); cout << "\n";
    cout << "Preorder: "; preorder(root); cout << "\n";
    cout << "Postorder:"; postorder(root); cout << "\n";
    return 0;
}
```

Q6 — Explain deletion in BST (code: delete node with usual 3 cases)

(You asked for code-only answers; below is full C++ delete implementation with comments explaining the 3 cases.)

```
// Question:
// Explain with help of code how deletion of a node in BST happens.
// (Cases: leaf node, node with one child, node with two children)

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

// find minimum value node in subtree (used to find inorder successor)
Node* findMin(Node* root) {
    while (root && root->left) root = root->left;
```

```
    return root;
}

// delete key from BST and return new root
Node* deleteNode(Node* root, int key) {
    if (!root) return nullptr;               // empty tree

    if (key < root->data) {
        // key is in left subtree
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        // key is in right subtree
        root->right = deleteNode(root->right, key);
    } else {
        // found node to delete
        if (!root->left && !root->right) {
            // Case 1: node is a leaf -> simply delete
            delete root;
            return nullptr;
        } else if (!root->left) {
            // Case 2: node has only right child -> replace node with
right child
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            // Case 2: node has only left child -> replace node with
left child
            Node* temp = root->left;
            delete root;
            return temp;
        } else {
            // Case 3: node has two children
            // Option: find inorder successor (smallest in right
subtree)
            Node* succ = findMin(root->right);
            root->data = succ->data;                // copy successor
value to current node
            root->right = deleteNode(root->right, succ->data); //
```

```
delete successor
        }
    }
    return root;
}

// utility: insert
Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// inorder print
void inorder(Node* r) { if (!r) return; inorder(r->left); cout << r-
>data << " "; inorder(r->right); }

int main() {
    // build sample BST
    int keys[] = {50, 30, 20, 40, 70, 60, 80};
    Node* root = nullptr;
    for (int k : keys) root = insert(root, k);

    cout << "Original Inorder: "; inorder(root); cout << "\n";

    // delete a leaf (20)
    root = deleteNode(root, 20);
    cout << "After deleting 20: "; inorder(root); cout << "\n";

    // delete node with one child (30 has left deleted earlier,
depends on structure)
    root = deleteNode(root, 30);
    cout << "After deleting 30: "; inorder(root); cout << "\n";

    // delete node with two children (50)
    root = deleteNode(root, 50);
    cout << "After deleting 50: "; inorder(root); cout << "\n";
```

```
    return 0;
}
```

Q7 — Construct an AVL tree with keys <29, 13, 20, 30, 50, 80> (AVL insertion with rotations)

```
// Question:
// Construct an AVL tree inserting keys: 29, 13, 20, 30, 50, 80
// (Code performs AVL insertions with balancing rotations)

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

struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
    Node(int k) : key(k), left(nullptr), right(nullptr), height(1) {}
};

// get node height
int height(Node* n) { return n ? n->height : 0; }

// update height
void updateHeight(Node* n) { n->height = 1 + max(height(n->left),
height(n->right)); }

// get balance factor (left height - right height)
int getBalance(Node* n) { return n ? height(n->left) - height(n-
>right) : 0; }

// right rotate (LL case)
Node* rightRotate(Node* y) {
    Node* x = y->left;
```

```
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    updateHeight(y);
    updateHeight(x);
    return x;
}

// left rotate (RR case)
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    updateHeight(x);
    updateHeight(y);
    return y;
}

// AVL insert
Node* avlInsert(Node* node, int key) {
    if (!node) return new Node(key);           // normal BST insert
    if (key < node->key) node->left = avlInsert(node->left, key);
    else if (key > node->key) node->right = avlInsert(node->right,
key);
    else return node; // no duplicates

    updateHeight(node);                        // update height after
insertion
    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && key < node->left->key) return
rightRotate(node);
    // Right Right Case
    if (balance < -1 && key > node->right->key) return
leftRotate(node);
    // Left Right Case
    if (balance > 1 && key > node->left->key) {
```

```
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

// preorder print to observe structure
void pre(Node* r) { if (!r) return; cout << r->key << " "; pre(r-
>left); pre(r->right); }

int main() {
    int keys[] = {29, 13, 20, 30, 50, 80};
    Node* root = nullptr;
    for (int k : keys) root = avlInsert(root, k);

    cout << "Preorder of constructed AVL tree: ";
    pre(root); cout << "\n";
    return 0;
}
```

Q8 — Bellman-Ford algorithm (pseudo-coded as C++ implementation)

```
// Question:
// Write algorithm (C++ implementation) for Bellman-Ford to find
single-source shortest path.
// (Detects negative-weight cycles)

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

```
using namespace std;

struct Edge { int u, v, w; }; // directed edge u->v with weight w

// Bellman-Ford: returns false if negative cycle reachable from src
bool bellmanFord(int V, vector<Edge>& edges, int src, vector<int>&
dist) {
    const int INF = numeric_limits<int>::max() / 4;
    dist.assign(V, INF);
    dist[src] = 0;

    // Relax edges V-1 times
    for (int i = 0; i < V - 1; ++i) {
        for (auto &e : edges) {
            if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.w;
            }
        }
    }

    // Check for negative-weight cycles
    for (auto &e : edges) {
        if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
            return false; // negative cycle detected
        }
    }
    return true;
}

int main() {
    int V = 5; // number of vertices
    vector<Edge> edges;
    // sample graph edges: u, v, weight
    edges.push_back({0, 1, 6});
    edges.push_back({0, 2, 7});
    edges.push_back({1, 2, 8});
    edges.push_back({1, 3, 5});
    edges.push_back({1, 4, -4});
    edges.push_back({2, 3, -3});
```

```
    edges.push_back({2, 4, 9});
    edges.push_back({3, 1, -2});
    edges.push_back({4, 0, 2});
    edges.push_back({4, 3, 7});

    vector<int> dist;
    bool ok = bellmanFord(V, edges, 0, dist);

    if (!ok) cout << "Graph contains negative weight cycle reachable
from source\n";
    else {
        cout << "Shortest distances from source 0:\n";
        for (int i = 0; i < V; ++i) {
            cout << "0 -> " << i << " = " << dist[i] << "\n";
        }
    }
    return 0;
}
```

Q9 — Dijkstra's algorithm (pseudocode implemented in C++ using priority_queue)

```
// Question:
// Provide pseudocode/implementation of Dijkstra's algorithm.
// State whether it is applicable to directed graphs (yes).

#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

using pii = pair<int,int>; // (distance, node)
const int INF = numeric_limits<int>::max();

void dijkstra(int V, vector<vector<pair<int,int>>>& adj, int src,
```

```
vector<int>& dist) {
    dist.assign(V, INF);
    priority_queue<pii, vector<pii>, greater<pii>> pq; // min-heap

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

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue; // stale entry

        for (auto &edge : adj[u]) {
            int v = edge.first;
            int w = edge.second;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}

int main() {
    int V = 5;
    vector<vector<pair<int,int>>> adj(V);
    // build sample directed graph: add edge u->v with weight w
    adj[0].push_back({1, 10});
    adj[0].push_back({4, 5});
    adj[1].push_back({2, 1});
    adj[1].push_back({4, 2});
    adj[2].push_back({3, 4});
    adj[3].push_back({0, 7});
    adj[4].push_back({1, 3});
    adj[4].push_back({2, 9});
    adj[4].push_back({3, 2});

    vector<int> dist;
    dijkstra(V, adj, 0, dist);
```

```
    cout << "Dijkstra distances from 0:\n";
    for (int i = 0; i < V; ++i) {
        cout << "0 -> " << i << " = " << dist[i] << "\n";
    }
    // Note: Dijkstra works on directed graphs and undirected graphs
but requires non-negative edge weights.
    return 0;
}
```

Q10 — Prim's Algorithm (MST) implementation (simple using adjacency matrix / priority queue)

```
// Question:
// Apply Prim's Algorithm to find MST for a weighted undirected graph.
// Implementation using adjacency list + min-heap.

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

using pii = pair<int,int>; // (weight, vertex)

int primMST(int V, vector<vector<pair<int,int>>>& adj) {
    vector<int> key(V, INT_MAX);      // track min weight to connect
vertex
    vector<bool> inMST(V, false);
    priority_queue<pii, vector<pii>, greater<pii>> pq; // (weight,
vertex)

    int src = 0;
    key[src] = 0;
    pq.push({0, src});
    int mst_weight = 0;

    while (!pq.empty()) {
```

```
        auto [w, u] = pq.top(); pq.pop();
        if (inMST[u]) continue;        // skip processed vertex
        inMST[u] = true;
        mst_weight += w;               // include edge weight

        for (auto &edge : adj[u]) {
            int v = edge.first;
            int wt = edge.second;
            if (!inMST[v] && wt < key[v]) {
                key[v] = wt;
                pq.push({key[v], v});
            }
        }
    }
    return mst_weight;
}

int main() {
    int V = 5;
    vector<vector<pair<int,int>>> adj(V);
    // undirected edges (u<->v, w)
    auto addEdge = [&](int u,int v,int w){ adj[u].push_back({v,w});
adj[v].push_back({u,w}); };
    addEdge(0,1,2);
    addEdge(0,3,6);
    addEdge(1,2,3);
    addEdge(1,3,8);
    addEdge(1,4,5);
    addEdge(2,4,7);
    addEdge(3,4,9);

    int total = primMST(V, adj);
    cout << "Total weight of MST (Prim): " << total << "\n";
    return 0;
}
```

Q11 — What happens to QuickSort complexity if all elements identical? (code + demonstration)

```
// Question:
// What will happen to complexity of Quick sort if all elements of the
array are identical?
// (Demonstration: naive quicksort with last-element pivot will
degenerate to O(n^2).)

#include <iostream>
using namespace std;

int partition(int arr[], int l, int r) {
    int pivot = arr[r];               // choose last element as pivot
    int i = l - 1;
    for (int j = l; j < r; ++j) {
        if (arr[j] <= pivot) {        // with identical elements, many
swaps occur
            ++i;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i+1], arr[r]);
    return i+1;
}

void quicksort(int arr[], int l, int r) {
    if (l < r) {
        int p = partition(arr, l, r);
        quicksort(arr, l, p - 1);
        quicksort(arr, p + 1, r);
    }
}

int main() {
    int n = 5;
    int arr[] = {1,1,1,1,1}; // all identical
    quicksort(arr, 0, n-1);
```

```
    cout << "After quicksort: ";
    for (int i=0;i<n;++i) cout<<arr[i]<<" ";
    cout << "\n";
    // Explanation (in comments): With identical elements and naive
partition,
    // pivot places narrow partitioning and recursion depth becomes
O(n), causing O(n^2) comparisons.
    return 0;
}
```

Q12 — Linear probing vs Quadratic probing (code demonstration + which is better)

```
// Question:
// Compare linear probing and quadratic probing.
// Provide small demonstration showing both insertion strategies.

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

// Linear probing insert into hash table with -1 = empty
void linearInsert(vector<int>& table, int key) {
    int m = table.size();
    int h = key % m;
    int idx = h;
    while (table[idx] != -1) { // probe linearly
        idx = (idx + 1) % m;
        if (idx == h) { cout << "Table full\n"; return; }
    }
    table[idx] = key;
}

// Quadratic probing insert: (h + i^2) mod m
void quadraticInsert(vector<int>& table, int key) {
    int m = table.size();
```

```
    int h = key % m;
    for (int i = 0; i < m; ++i) {
        int idx = (h + i * i) % m;
        if (table[idx] == -1) { table[idx] = key; return; }
    }
    cout << "Quadratic: Table full or probing failed\n";
}

int main() {
    vector<int> t1(7, -1), t2(7, -1);
    int keys[] = {10, 20, 30, 40, 50};
    for (int k : keys) { linearInsert(t1, k); quadraticInsert(t2,
k); }

    cout << "Linear probing table: ";
    for (int v : t1) cout << v << " ";
    cout << "\nQuadratic probing table: ";
    for (int v : t2) cout << v << " ";
    cout << "\n";
    // Which is better? Quadratic probing reduces primary clustering
vs linear,
    // but may fail to find slot depending on table size and probing
sequence.
    return 0;
}
```

Q13 — Quadratic probing failure example (m=4)

```
// Question:
// For table size m=4, T[0] and T[2] occupied, T[1], T[3] empty.
// Show that inserting key with h(key)=0 using (h + i^2) mod 4 fails
even though free slots exist.

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

```
int main() {
    int m = 4;
    vector<int> T(m, -1);
    // occupy T[0] and T[2]
    T[0] = 100; T[2] = 200;
    cout << "Initial table: ";
    for (int i = 0; i < m; ++i) {
        if (T[i] == -1) cout << "_ ";
        else cout << T[i] << " ";
    }
    cout << "\n";

    int key = 8; // h(key) = 8 % 4 = 0
    int h = key % m;
    bool inserted = false;
    // quadratic probing sequence: (h + i^2) mod 4 for i = 0..m-1
    for (int i = 0; i < m; ++i) {
        int idx = (h + i * i) % m;
        cout << "i=" << i << " probes index " << idx << "\n";
        if (T[idx] == -1) { T[idx] = key; inserted = true; break; }
    }
    if (!inserted) cout << "Insertion failed with quadratic
probing\n";
    else {
        cout << "After insertion: ";
        for (int v : T) cout << v << " ";
        cout << "\n";
    }
    // In this scenario the probe sequence visits indices: 0,1,0,1,...
or 0,1,0,1 depending on modulo,
    // actually for m=4 and i=0..3: indices: 0,1,0,1 -> it never
checks index 3 so insertion may fail.
    return 0;
}
```

Q14 — Illustrate hashing techniques: Division, Multiplication, Mid-square, Folding (demo code)

```
// Question:
// Demonstrate hashing techniques:
// (i) Division Method (ii) Multiplication Method (iii) Mid-square
(iv) Folding

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

// Division method: h(k) = k % m
int hashDivision(int k, int m) { return k % m; }

// Multiplication method: h(k) = floor(m * frac(k*A)), where A in
(0,1), typical A = (sqrt(5)-1)/2
int hashMultiplication(int k, int m) {
    double A = (sqrt(5.0) - 1) / 2.0; // Knuth's suggestion
    double frac = fmod(k * A, 1.0);   // fractional part
    return int(floor(m * frac));
}

// Mid-square: square the key, extract middle digits then mod m if
needed
int hashMidSquare(int k, int m) {
    long long sq = 1LL * k * k;
    // get middle 2 digits (example), convert to positive
    string s = to_string(llabs(sq));
    int len = s.size();
    int mid = len / 2;
    // extract two digits around middle (safe-guard)
    int start = max(0, mid - 1);
    string midDigits = s.substr(start, min(2, (int)s.size()-start));
    int val = stoi(midDigits);
    return val % m;
}
```

```
// Folding: break key into parts and add them
int hashFolding(int k, int m) {
    string s = to_string(abs(k));
    int partSize = 2; // group digits by 2
    int sum = 0;
    for (int i = s.size(); i > 0; i -= partSize) {
        int start = max(0, i - partSize);
        string part = s.substr(start, i - start);
        sum += stoi(part);
    }
    return sum % m;
}

int main() {
    int keys[] = {1234, 5678, 9012, 3456};
    int m = 10;
    cout << "Division:"; for (int k : keys) cout << " " <<
hashDivision(k,m); cout << "\n";
    cout << "Multiplication:"; for (int k : keys) cout << " " <<
hashMultiplication(k,m); cout << "\n";
    cout << "Mid-square:"; for (int k : keys) cout << " " <<
hashMidSquare(k,m); cout << "\n";
    cout << "Folding:"; for (int k : keys) cout << " " <<
hashFolding(k,m); cout << "\n";
    return 0;
}
```