i)

WhatsApp is a mobile app that allow its users to send and receive text, audio and video messages. The WhatsApp team wish to find whether a messages reach back to its sender or not? For example, a person P sends a message to his friends and they forward the message to their friends and so on. Given the users of WhatsApp and the information of all the messages sent by its users to other users, we need to determine that a message is sent back to its user or not for a particular user.

a) What data structure(s) and algorithm(s) is most suitable to represent this data and why?

```
1. •  Data Structure:

2. Use a directed graph to represent the relationships between users.
3. Each user is represented as a node (vertex).
4. Each message sent is represented as a directed edge from one node to another.

5. •  Why a Directed Graph?
6. The problem involves determining whether a directed path (message forwarding) forms a cycle that
returns to the sender.
7. A directed graph is well-suited to represent this type of relationship.

8. •  Algorithm:
9. Use Depth-First Search (DFS) to traverse the graph and detect cycles.
10. During DFS, if a node is revisited while still being in the current recursion stack, a cycle is
detected (indicating the message returns to its sender).
```

b) How would you solve this problem with your suggested data structure? Briefly explain your idea in the form of pseudocode.

b) Solution with Suggested Data Structure

Pseudocode:

1. Represent the directed graph using an **adjacency list**.
2. Use a **DFS traversal** to check for cycles:
    - Use a visited set to keep track of visited nodes.
    - Use a recStack (recursion stack) to detect back edges (indicative of cycles).

```
Function isMessageReturned(graph, start):

    # Initialize visited and recStack

    visited ← empty set      # Set to track visited nodes
    recStack ← empty set     # Set to track nodes in the current recursion stack

    # Helper function for Depth-First Search (DFS)

    Function dfs(node):
        Add node to visited    # Mark the node as visited
        Add node to recStack   # Add the node to the recursion stack

        # Traverse all neighbors of the current node

        For each neighbor in graph[node]:
            If neighbor is not in visited:        # If the neighbor has not been visited
                If dfs(neighbor) is True:         # Recur for the neighbor
                    Return True                   # Cycle detected
            Else if neighbor is in recStack:      # If the neighbor is in the recursion stack
                Return True                       # Cycle detected (back edge found)

        Remove node from recStack                 # Backtracking: remove node from recursion
stack
        Return False                              # No cycle found from this node

    # Call dfs for the starting node

    Return dfs(start)

# Example usage
Function main():
    # Input: Number of users (nodes) and connections (edges)
    numUsers ← Input("Enter the number of users:")
    numConnections ← Input("Enter the number of connections:")

    # Represent the graph as an adjacency list
    graph ← empty dictionary
    For i from 0 to numUsers - 1:
        graph[i] ← empty list

    # Input the connections (directed edges)
    For i from 0 to numConnections - 1:
        u, v ← Input("Enter connection (u → v):")
        Append v to graph[u]

    # Input the starting user
    start ← Input("Enter the starting user:")

    # Check if the message returns to its sender
    If isMessageReturned(graph, start) is True:
        Print("The message reaches back to its sender.")
    Else:
        Print("The message does not reach back to its sender.")

```

2nd Way

```
Function isMessageReturned(graph, start):

    Initialize visited as an empty set
    Initialize recStack as an empty set

    Function dfs(node):

        Add node to visited
        Add node to recStack

        For each neighbor in graph[node]:

            If neighbor is not in visited:
                If dfs(neighbor) is True:
                    Return True  # Cycle detected
            Else if neighbor is in recStack:
                Return True  # Back edge found, cycle detected

        Remove node from recStack
        Return False

    Return dfs(start)  # Start DFS from the given node

```

Explanation of the Pseudocode:

1. **Graph Representation**:
    - The graph is represented as an adjacency list, where each node maps to a list of its neighbors. This allows efficient traversal of the graph.
2. **Cycle Detection**:
    - Start DFS from the given user (start).
    - During DFS:
        - Mark the current node as visited.
        - Add the current node to the recursion stack.
        - Traverse all neighbors of the current node.
            - If a neighbor is unvisited, recursively call DFS on it.
            - If a neighbor is in the recursion stack, a **cycle** is detected.
    - Remove the node from the recursion stack after all its neighbors are processed.
3. **Return Value**:
    - If a cycle is found during DFS, the message returns to the sender.
    - Otherwise, it does not.

Example Workflow:

1. **Input**: A directed graph with edges:

- 0 → 1, 1 → 2, 2 → 0 (cycle exists).

2. **Start**: User 0.
3. **Output**: True (the message reaches back to the sender).

---

ii)

You are required to develop a cab hailing app. Once a client requests a cab via your app, it shou select the cabs which are free in the vicinity of the client (the one who is closest to the client shou be allocated the request first, if the cab driver declines the request then it should be forwarded to the second nearest and so on) till a cab driver accepts the ride request. Once the cab has reached the client, your app should calculate the shortest distance to the destination and guide the driver accordingly.

a) Which data structures and algorithms would you like to use in your app so that it functions

efficiently?

```
   1. Data Structures:
 2. Min-Heap (Priority Queue):

To efficiently select the nearest cabs, store cabs in a min-heap where the key is the distance of
the cab from the client. This allows efficient retrieval of the nearest cab.

 3. Graph Representation (Adjacency List or Matrix):
For guiding the driver to the destination, represent the road network as a graph. Nodes represent
locations (e.g., intersections), and edges represent roads with weights (distances or travel times).

 4. Hash Table:
To maintain the availability status of cabs and their locations (e.g., cab_id -> {location,
availability}).

 5. Spatial Data Structure (e.g., KD-Tree):
For efficient nearest-neighbor searches, especially in dense urban areas.

 6. •  Algorithms:
 7. Dijkstra's Algorithm:
To calculate the shortest distance from the cab's current location to the destination.

 8. Haversine Formula (for Latitude-Longitude):
To calculate the great-circle distance between two points on Earth for real-world applications.

 9. Breadth-First Search (BFS):
For simpler shortest path calculations if edge weights are uniform.
```

b) Give an algorithm (in terms of pseudo-code) that should run on your data structures specified above and selects cabs located in the vicinity of the client (the driver present nearest should be selected first, then the 2nd nearest and so on).

➔ **Key Idea:**

1. Use a **min-heap** to store available cabs based on their distances from the client.
2. Pop the nearest cab from the heap, and if the driver declines, move to the next nearest cab.
3. Once a cab accepts, calculate the shortest path from the client to the destination.

```
// Function to calculate the distance between two points (latitude and longitude)
double calculateDistance(lat1, lon1, lat2, lon2) {
    // Use the Haversine formula to calculate distance
    // (assuming input is in degrees and Earth is a sphere)
    const double R = 6371.0; // Radius of Earth in kilometers
    double dLat = (lat2 - lat1) * PI / 180.0;
    double dLon = (lon2 - lon1) * PI / 180.0;
    double a = sin(dLat / 2) * sin(dLat / 2) +
               cos(lat1 * PI / 180.0) * cos(lat2 * PI / 180.0) *
               sin(dLon / 2) * sin(dLon / 2);
    double c = 2 * atan2(sqrt(a), sqrt(1 - a));
    return R * c; // Distance in kilometers
}

// Function to select the nearest cab

void selectNearestCab(clientLocation, destination, cabs) {

    // Min-Heap for storing cabs based on their distance to the client

    PriorityQueue<Cab> minHeap;

    // Populate the heap with distances of available cabs

    for (Cab cab : cabs) {
        if (cab.isAvailable) {
            double distance = calculateDistance(clientLocation.lat, clientLocation.lon,
cab.location.lat, cab.location.lon);
            minHeap.push({distance, cab}); // Min-Heap orders by distance
        }
    }

    // Iterate over the cabs in the order of proximity

    while (!minHeap.empty()) {
        Cab nearestCab = minHeap.top().cab;
        minHeap.pop();

        // Send ride request to the nearest cab

        if (nearestCab.acceptRequest()) {
            cout << "Cab " << nearestCab.id << " has accepted the ride request." << endl;

```

```
            // Calculate the shortest path to the destination
            vector<int> shortestPath = calculateShortestPath(nearestCab.location, destination);

            // Guide the driver using the shortest path
            guideDriver(shortestPath);
            return;
        } else {
            cout << "Cab " << nearestCab.id << " has declined the ride request." << endl;
        }
    }

    // If no cabs accept the ride request
    cout << "No cabs available at the moment." << endl;
}

// Function to calculate the shortest path using Dijkstra's algorithm
vector<int> calculateShortestPath(start, end) {
    // Use a graph representation (adjacency list or matrix)
    // Implement Dijkstra's algorithm to compute the shortest path
    return dijkstraShortestPath(graph, start, end);
}
```

Explanation of Pseudocode:

1. **Distance Calculation**:
    - The calculateDistance function computes the distance between the client and a cab using the **Haversine formula** for geographical coordinates.
2. **Min-Heap**:
    - All available cabs are inserted into the min-heap with their distances to the client as the key. This ensures that the nearest cab is always on top of the heap.
3. **Request Handling**:
    - The nearest cab is queried first. If the driver declines, the app moves on to the next cab until one accepts or no cabs are left.
4. **Shortest Path**:
    - Once a cab accepts the ride, the shortest path from the pickup location to the destination is computed using **Dijkstra's algorithm** or a similar shortest path algorithm.
5. **Driver Guidance**:

- The guideDriver function provides step-by-step directions to the driver based on the shortest path.

STACKS

Write a function called copyStack to copy the elements of an integer type stack, sl, into another stack, s2, where both sl and s2 are reference parameters, and s2 is initially empty, copyStack is a global function which is only allowed to use the standard stack ADT functions: push, pop, top (or peek), and empty. Furthermore, copyStack cannot create an extra array or any other data structure, but it can use one or two extra variables. When the function ends, s2 contains exactly the same elements as sl, and sl is exactly as it was initially.

Following picture shows an example, before and after copyStack:

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

// Global function to copy one stack into another
void copyStack(stack<int> &sl, stack<int> &s2) {
    // Step 1: Transfer elements from sl to s2 (reversing sl)
    while (!sl.empty()) {
        s2.push(sl.top());
        sl.pop();
    }

    // Step 2: Transfer elements back from s2 to sl (restoring sl's order)
    stack<int> tempStack; // Temporary stack to preserve s2 during this process
    while (!s2.empty()) {
        int temp = s2.top();
        s2.pop();
        sl.push(temp);
        tempStack.push(temp); // Store the element temporarily
    }

    // Step 3: Transfer elements from tempStack back to s2 to finalize its copy
    while (!tempStack.empty()) {
        s2.push(tempStack.top());
        tempStack.pop();
    }
}
```

HASH

Add the numbers 120, 55, 22, 13, 1, 5, 25, 15 to a Hash table with 7 buckets using the simple modulo function as the hash function. Show the table if Chaining is chosen for the hash table for overflow handling.

To create a hash table with **7 buckets** and use the **modulo function** as the hash function (key%7k), we compute the bucket index for each number and handle collisions using **chaining** (linked lists in each bucket).

Step 1: Compute Bucket Indexes

```
1. We calculate the bucket for each number using key%7:

2. 120%7=1120 \% 7 = 1
3. 55%7=655 \% 7 = 6
4. 22%7=122 \% 7 = 1
5. 13%7=613 \% 7 = 6
6. 1%7=11 \% 7 = 1
7. 5%7=55 \% 7 = 5
8. 25%7=425 \% 7 = 4
9. 15%7=115 \% 7 = 1
10.
```

Step 2: Build the Hash Table with Chaining

```
1. Each bucket contains a linked list for numbers mapping to the same index.
2. Bucket Linked List (Chained Entries)
3. 0      Empty
4. 1      120 → 22 → 1 → 15
5. 2      Empty
6. 3      Empty
7. 4      25
8. 5      5
9. 6      55 → 13
```

Explanation:

1. **Bucket 0**: No numbers map here (key%7≠0key \% 7 \neq 0).
2. **Bucket 1**: Numbers 120, 22, 1, and 15 all map here, forming a chain 120→22→1→15120 → 22 → 1 → 15.
3. **Bucket 2**: No numbers map here (key%7≠2key \% 7 \neq 2).
4. **Bucket 3**: No numbers map here (key%7≠3key \% 7 \neq 3).
5. **Bucket 4**: Number 25 maps here (25%7=425 \% 7 = 4).

6. **Bucket 5**: Number 5 maps here (5%7=55 \% 7 = 5).
7. **Bucket 6**: Numbers 55 and 13 map here, forming a chain 55→1355 → 13.

```
Final Hash Table with Chaining:
Bucket 0:
Bucket 1: 120 → 22 → 1 → 15
Bucket 2:
Bucket 3:
Bucket 4: 25
Bucket 5: 5
Bucket 6: 55 → 13
This structure ensures collisions are handled efficiently using chaining.

```

➢ Time Complexity

Calculate the time complexity expression for the following code snippet with respect to n and express

it in terms of Big-O notation. Show all working for full credit.

```
for (int i = 1; i < n; i = i * 2) {
    for (int j = 0; j < i; ++j) {
        sum++;
    }
}

```

ii. While doing worst case analysis of an algorithm, I found it takes 2 steps to compiste the job, where n is the input size. Can I say it is 0(2")? Why or why not?

```
ii. Analysis of the statement:
"Can I say it is O(2n)? Why or why not?"
If the algorithm takes a constant number of steps (e.g., 2 steps) irrespective of the input size nnn, then its time complexity is O(1) because the runtime does not grow with n.

Reasoning:
O(2^n) implies the runtime grows exponentially with n, which is not the case here.
The algorithm's runtime being fixed at 2 steps is independent of n, so O(2^n) is incorrect.
The correct time complexity is O(1).

Summary:
The time complexity of the code snippet is O(n).
The algorithm with 2 constant steps is O(1), not O(2n)O(2^n)O(2n).

```

SPACE COMPLEXITY + TIME

```
1. int Enigma(int N) {
2.     if (N <= 1)
3.         return 11;
4.     else {
5.         Enigma(N - N / 2);
6.         Enigma(N - N / 2);
7.     }
8. }
9.
```

Since the number of levels is O(log2(N)), the total time complexity is **exponential**.

T(N) = O(2^N)

Step 2: Space Complexity

Recursive Space:

The space complexity is determined by the depth of the recursion tree. Since each recursive call reduces N geometrically, the depth of the recursion tree is:

O(log2(N))

For example, if the array is {3, 7, 5, 12, 6) and k = 3, then the function should return 25 (12+7+6) i.e.
• A number k≤n and calculates the sum of the k largest numbers in the array.

```
1. Function SumOfLargestKValues(array, n, k):

2.     1. Sort the array in descending order.  // Time complexity: O(n log n)
3.     2. Initialize sum ← 0

4.     3. For i ← 0 to k-1:  // Iterate over the first k elements
5.            sum ← sum + array[i]
6.     4. Return sum
7.
```

Explanation

```
1. Sorting:
```

```
2. Sort the array in descending order so that the k largest elements are at the beginning of the
array.
3. Sorting takes O(nlogn) time.
4. Summing the k largest values:
5. Iterate through the first k elements of the sorted array and compute their sum.
6. This loop runs O(k), which is O(n) in the worst case if k=n, but does not exceed   O(nlogn)
overall due to the sorting step.

7. Returning the Result:
8. The computed sum is returned as the output.
```

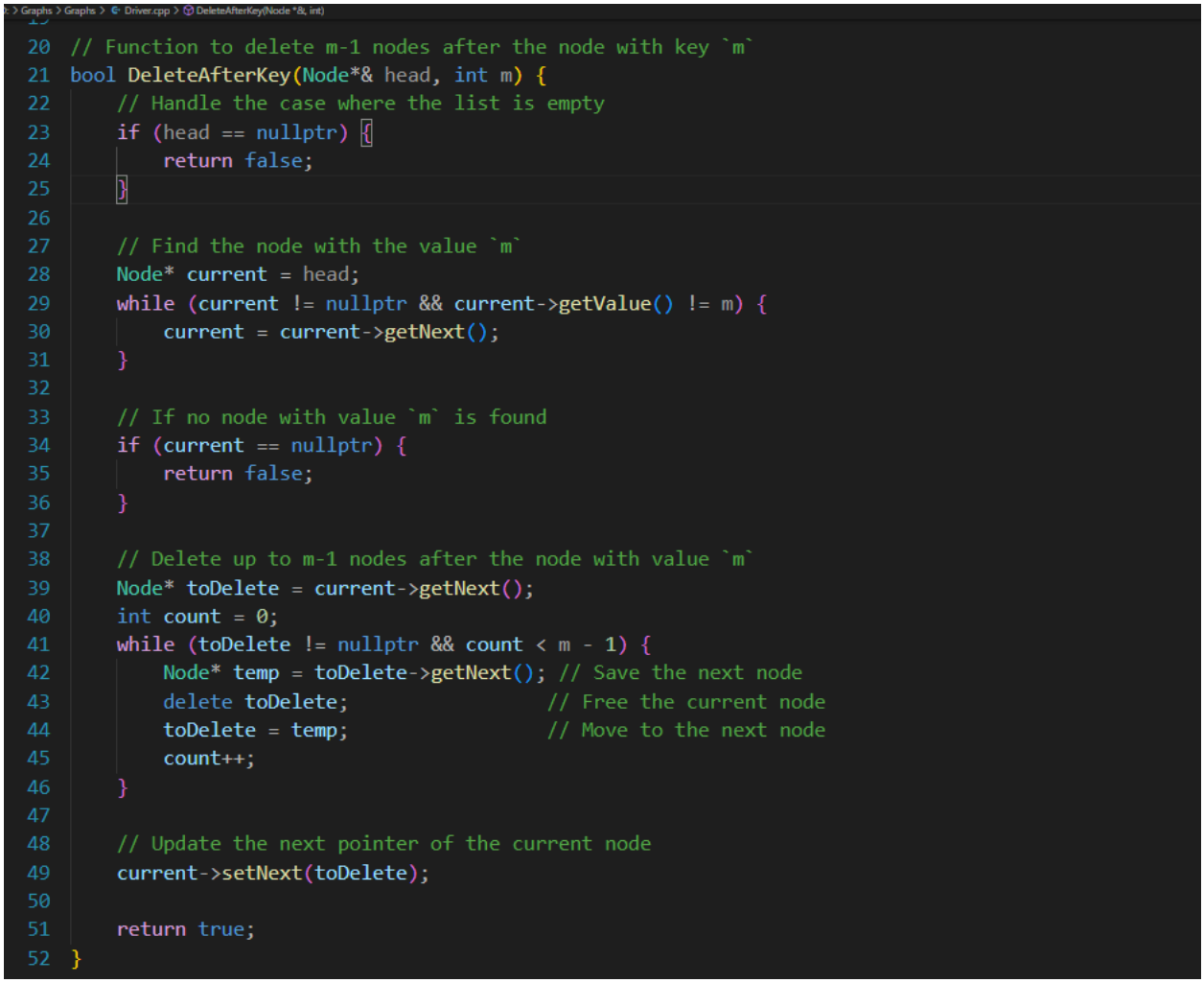
Example Walkthrough

```
1. Input:
2. Array: [3,7,5,12,6]
3. n=5 k=3
4. Execution:
5. Sort the array in descending order:
6. [12,7,6,5,3]
7. Sum the first k=3 largest elements:
8. 12+7+6=25
9. Return 25
```

Consider the list to end at the last node). So if m=5, it will find the node containing 5, then delete this node as ell as the 4 nodes that come after it. If there are less than m-1 nodes after the node with key=m, all these would be deleted. And if the list is: 2->9->4->6->1->2->5- then delete AfterKey(6) will result in 2->9->4

For example if the list is 2->9->4->3->1->2->5-then delete AfterKey(3) would result in 2->9->4->5 If there is no node in the list with key=m, the function simply returns false. If you use any helper function then implement it as well. Make sure your code runs in linear time and creates no memory leaks. It should also handle all special cases.

```
54  // Helper function to add a node at the end of the list
55  void Append(Node*& head, int value) {
56      if (head == nullptr) {
57          head = new Node(value);
58          return;
59      }
60      Node* current = head;
61      while (current->getNext() != nullptr) {
62          current = current->getNext();
63      }
64      current->setNext(new Node(value));
65  }
```



CIRCULAR SINGLY

```cpp
bool deleteAfterKey(int m) {
    if (last == nullptr) {
        return false; // Empty list
    }

    // Find the node with key `m`
    Node* current = last->next;
    do {
        if (current->key == m) {
            break;
        }
        current = current->next;
    } while (current != last->next);

    // If key `m` is not found
    if (current->key != m) {
        return false;
    }

    // Delete the next m-1 nodes
    Node* toDelete = current->next;
    for (int count = 0; count < m - 1 && toDelete != nullptr; ++count) {
        if (toDelete == last) {
            last = (last->next == last) ? nullptr : current; // Adjust last pointer if needed
        }
        Node* temp = toDelete->next;
        delete toDelete;
        toDelete = temp;
    }

    // Update the current node's next pointer
    current->next = toDelete;

    // If all nodes are deleted, adjust `last` pointer
    if (toDelete == nullptr) {
        last = nullptr;
    }

    return true;
}
```

```cpp
    // Function to add a node to the end of the list
    void append(int key) {
        Node* newNode = new Node(key);
        if (last == nullptr) {
            last = newNode;
            last->next = last; // Circular link
        } else {
            newNode->next = last->next;
            last->next = newNode;
            last = newNode;
        }
    }

```

To convert the postfix expression XYZ/+EJ%-KF*+ to prefix, follow these steps systematically:

Steps:

1. **Understand Postfix**: In postfix notation, operators follow their operands.
2. **Conversion Process**:
    - Use a stack to process the postfix expression.
    - Push operands (letters) onto the stack.
    - When an operator is encountered, pop the necessary number of operands from the stack, prepend them with the operator, and push the resulting expression back onto the stack.
3. **Result**: Once the expression is fully processed, the stack will contain the prefix expression.

Conversion:

```
1. Given postfix: XYZ/+EJ%-KF*+

2. Step-by-step Process:

3. Process X, Y, and Z: Push them onto the stack.
4. Stack: X, Y, Z
5. Encounter /: Pop Z and Y, apply /, and push (/Y Z).
6. Stack: X, /YZ
7. Encounter +: Pop /YZ and X, apply +, and push (+X /YZ).
8. Stack: +X /YZ
9. Process E and J: Push them onto the stack.
10. Stack: +X /YZ, E, J
11. Encounter %: Pop J and E, apply %, and push (%E J).
12. Stack: +X /YZ, %E J
13. Encounter -: Pop %E J and +X /YZ, apply -, and push (-+X /YZ %E J).
14. Stack: -+X /YZ %E J
15. Process K and F: Push them onto the stack.
16. Stack: -+X /YZ %E J, K, F
17. Encounter *: Pop F and K, apply *, and push (*K F).
18. Stack: -+X /YZ %E J, *K F
19. Encounter +: Pop *K F and -+X /YZ %E J, apply +, and push (+ -+X /YZ %E J *K F).
20. Stack: + -+X /YZ %E J *K F


21. Final Prefix Expression:
22. + - + X / Y Z % E J * K F
```

```
// Data structure to represent a Location
STRUCT Location:
    latitude
    longitude

// Class to represent a Cab
CLASS Cab:
    id
    location
    available (Boolean)  // If the cab is available for a ride

    // Function to calculate Euclidean distance (for simplicity)
    FUNCTION distanceTo(clientLocation):
        RETURN sqrt((clientLocation.latitude - location.latitude)^2 + (clientLocation.longitude - location.longitude)^2)

    // Function to check if the cab can accept the ride
    FUNCTION acceptsRequest():
        // Logic to check if the cab accepts the ride
        RETURN available


// Class to represent a Graph for Dijkstra's algorithm
CLASS Graph:
    // A map of nodes (intersections) and edges (roads)
    nodes
    edges

    // Add nodes and edges to the graph (representing road network)
    FUNCTION addEdge(nodeA, nodeB, distance):
        // Add an edge between nodeA and nodeB with a given distance

    // Function to run Dijkstra's Algorithm
    FUNCTION dijkstra(startNode, endNode):

        // Initialize distances for all nodes

        distances[startNode] = 0
        FOR each node in nodes:
            IF node != startNode:
                distances[node] = infinity

        // Set of unvisited nodes
        unvisitedNodes = nodes

        WHILE unvisitedNodes is NOT empty:

            currentNode = node with the smallest distance from startNode

            REMOVE currentNode from unvisitedNodes

            FOR each neighbor of currentNode:

                newDistance = distances[currentNode] + distance from currentNode to neighbor

                IF newDistance < distances[neighbor]:

                    distances[neighbor] = newDistance

```

```
        // Return the shortest path distance to endNode
        RETURN distances[endNode]


// Class to represent a Priority Queue for selecting the nearest available cab
CLASS CabQueue:

    // A priority queue (min-heap) to manage cabs based on proximity

    FUNCTION addCab(cab, distance):

        // Add cab to priority queue with distance as priority

    FUNCTION getClosestCab():

        IF queue is NOT empty:

            closestCab = REMOVE top cab from queue


            RETURN closestCab
        ELSE:
            RETURN INVALID CAB (No cabs available)


// Main function to select the nearest available cab and calculate the shortest path

FUNCTION selectCabForRide(clientLocation, destinationLocation, cabs, graph, cabQueue):

    // Step 1: Search for nearby cabs from the list

    FOR each cab in cabs:
        IF cab is available:
            distance = cab.distanceTo(clientLocation)
            cabQueue.addCab(cab, distance)

    // Step 2: Select the closest available cab

    selectedCab = cabQueue.getClosestCab()

    // Step 3: Check if the selected cab accepts the ride

    WHILE selectedCab is NOT INVALID CAB:
        IF selectedCab.acceptsRequest():

            // Calculate the shortest path from cab to client

            cabToClientDistance = graph.dijkstra(selectedCab.location, clientLocation)

            // Calculate the shortest path from client to destination

            clientToDestinationDistance = graph.dijkstra(clientLocation, destinationLocation)

            // Total distance for the ride

            totalRideDistance = cabToClientDistance + clientToDestinationDistance

            PRINT "Cab assigned: " + selectedCab.id
            PRINT "Total ride distance: " + totalRideDistance
            BREAK

        ELSE:
```

```
            // If the selected cab declines, continue with the next closest cab

            selectedCab = cabQueue.getClosestCab()

    // Step 4: Handle case if no cab accepts the request

    IF selectedCab is INVALID CAB:

        PRINT "No available cab found."

        // Notify the client that no cabs are available


// Main Program Execution:
BEGIN
    // Create an instance of the Graph, CabQueue, and list of cabs
    graph = NEW Graph
    cabQueue = NEW CabQueue
    cabs = [Cab(1, Location(40.730000, -73.935000)),
            Cab(2, Location(40.731000, -73.934000)),
            Cab(3, Location(40.732000, -73.936000))]

    // Add roads (edges) to the graph (example data)
    graph.addEdge(Node(1), Node(2), 5)  // Road between node 1 and node 2 with distance 5
    graph.addEdge(Node(2), Node(3), 3)  // Road between node 2 and node 3 with distance 3

    // Example locations for the client and the destination
    clientLocation = Location(40.730610, -73.935242)
    destinationLocation = Location(40.735610, -73.940242)

    // Select a cab for the client ride
    selectCabForRide(clientLocation, destinationLocation, cabs, graph, cabQueue)
END
```

Explanation of the Pseudo-code:

1. Cab and Location:
2. Each Cab has an ID, a location (latitude and longitude), and an availability status. The distanceTo function calculates the Euclidean distance between the cab's location and the client's location.
3. The acceptsRequest function checks if the cab is available and can accept the ride request.

4. Graph (Road Network):
5. A Graph class is used to represent the road network, where nodes represent intersections (locations) and edges represent roads (with distances).
6. The dijkstra function implements Dijkstra's algorithm to find the shortest path from the starting node to the destination node. It returns the shortest distance between the start and end nodes.

7. Priority Queue:
8. A CabQueue (min-heap) stores cabs sorted by their proximity to the client. The addCab function adds a cab to the queue based on its distance to the client, and the getClosestCab function retrieves the closest available cab.

```
 9. Selecting the Closest Available Cab:
10. The selectCabForRide function starts by searching for nearby cabs from the list and adding them
to the priority queue.
11. It then selects the closest available cab and checks if it accepts the ride.
12. If the selected cab accepts the ride, Dijkstra's algorithm is used to calculate the shortest
distance from the cab to the client (cabToClientDistance), and from the client to the destination
(clientToDestinationDistance).
13. The total ride distance is the sum of these two distances.
14. If the cab declines the ride, the algorithm tries the next closest cab.


15. Handling No Cab Acceptance:
16. If no cab accepts the ride request, the system notifies the client that no cabs are available.
17.
```

Key Points:

- **Dijkstra's Algorithm** helps find the shortest path between the cab and the client, and between the client and the destination.
- **Priority Queue** ensures that the closest cab is selected first.
- The algorithm is efficient for finding the optimal route and handling multiple cabs.