## ➢ Time complexity

```cpp
for (i = 1; i <= n; i = i * 2) {
    for (j = 1; j <= i; ++j) {
        cout << j << endl;
    }
}
```

```
Outer loop analysis:
The outer loop runs with i = 1, 2, 4, 8, ..., n.
i doubles in each iteration.
The number of iterations is approximately log_2(n) (as i increases exponentially by powers of 2, up to n).


Inner loop analysis:
For each iteration of the outer loop, the inner loop runs i times (i.e., from j = 1 to j = i).
Hence, the number of iterations for the inner loop depends on the value of i.


Total work:
To calculate the total number of iterations, sum the iterations of the inner loop for each value of i:
When i = 1, inner loop runs 1 time.
When i = 2, inner loop runs 2 times.
When i = 4, inner loop runs 4 times.
...
When i = n, inner loop runs n times.

Thus, the total number of iterations is:
1+2+4+8+⋯+n1 + 2 + 4 + 8 + ... + n

This is a geometric progression with the sum:
Sum=20+21+22+⋯+2k where k=log2(n)\text{Sum} = 2^0 + 2^1 + 2^2 + \dots + 2^k \quad \text{where } k = \log_2(n)
The sum of this series is:
Sum= = 2^{k+1} - 1
Since k = \log_2(n), we have:
Sum= 2^{\log_2(n) + 1} - 1 = 2n - 1

Time Complexity:
The total work done is proportional to 2n−12n - 1, which simplifies to O(n)O(n).
Final Answer:
The time complexity of the given code snippet is O(n).
```

---

## ➢ Space complexity

```
int root(double n) {
    if (n > 1.0)
        return (1 + root(n / 2));
    else
        return 0;
}

```

```
Understanding the Function:
Input: A positive number n (assume n is a multiple of 2).
Recursive Calls:
If n > 1.0, the function makes a recursive call with n / 2 as the input.
If n <= 1.0, the recursion terminates.
Recursion Depth:
The recursion keeps halving n until it becomes less than or equal to 1.
For example, if n = 16, the recursive calls are made with n = 8, 4, 2, 1.
This implies the number of recursive calls is proportional to log₂(n).
Space Complexity Analysis:
The space complexity of a recursive function depends on the memory required to store:
Local Variables: Each function call needs space for its local variables and parameters.
In this case, the function uses one parameter (n) and a small constant amount of additional memory.
Call Stack:
Each recursive call adds a new frame to the call stack.
The maximum depth of recursion determines the maximum memory used by the call stack.
Since the recursion depth is log₂(n) (as explained earlier), the call stack uses O(log₂(n)) space.
Total Space Complexity:
The space required for each individual call is constant (O(1)).
The total space used by the call stack is proportional to the recursion depth, which is O(log₂(n)).
Final Answer:
The space complexity of the given code is O(logn).


```

--------------------------------------------------------------------------------------------------------------------------

Generate the prefix expression against the following postfix expression: XYZ/+EJ%-KF*+

```
To convert the given postfix expression XYZ/+EJ%-KF*+ into its prefix expression, we need to follow these steps:
Steps to Convert Postfix to Prefix:
Start from left to right in the postfix expression.
```

```
Use a stack to build expressions.
Whenever encountering an operand, push it onto the stack.
Whenever encountering an operator, pop the required number of operands from the stack, combine them with the operator in prefix order, and push the result back onto the stack.
At the end, the stack will contain the complete prefix expression.



Given Postfix Expression: XYZ/+EJ%-KF*+
Step-by-Step Conversion:
Encounter X (operand):
Push onto stack: Stack = [X].
Encounter Y (operand):
Push onto stack: Stack = [X, Y].
Encounter Z (operand):
Push onto stack: Stack = [X, Y, Z].
Encounter / (operator):
Pop Z and Y.
Combine: / Y Z → Push back onto stack.
Stack: [X, /YZ].
Encounter + (operator):
Pop /YZ and X.
Combine: + X /YZ → Push back onto stack.
Stack: [+X/YZ].
Encounter E (operand):
Push onto stack: Stack = [+X/YZ, E].
Encounter J (operand):
Push onto stack: Stack = [+X/YZ, E, J].
Encounter % (operator):
Pop J and E.
Combine: % E J → Push back onto stack.
Stack: [+X/YZ, %EJ].
Encounter - (operator):
Pop %EJ and +X/YZ.
Combine: - +X/YZ %EJ → Push back onto stack.
Stack: [-+X/YZ%EJ].
Encounter K (operand):
Push onto stack: Stack = [-+X/YZ%EJ, K].
Encounter F (operand):
Push onto stack: Stack = [-+X/YZ%EJ, K, F].
Encounter * (operator):
Pop F and K.
Combine: * K F → Push back onto stack.
Stack: [-+X/YZ%EJ, *KF].
Encounter + (operator):
Pop *KF and -+X/YZ%EJ.
Combine: + -+X/YZ%EJ *KF → Push back onto stack.
Stack: [+-+X/YZ%EJ*KF].

Final Prefix Expression:
The prefix expression is:
{+-+X/YZ%EJ*KF}

```

---

## ➢ LINKED LIST

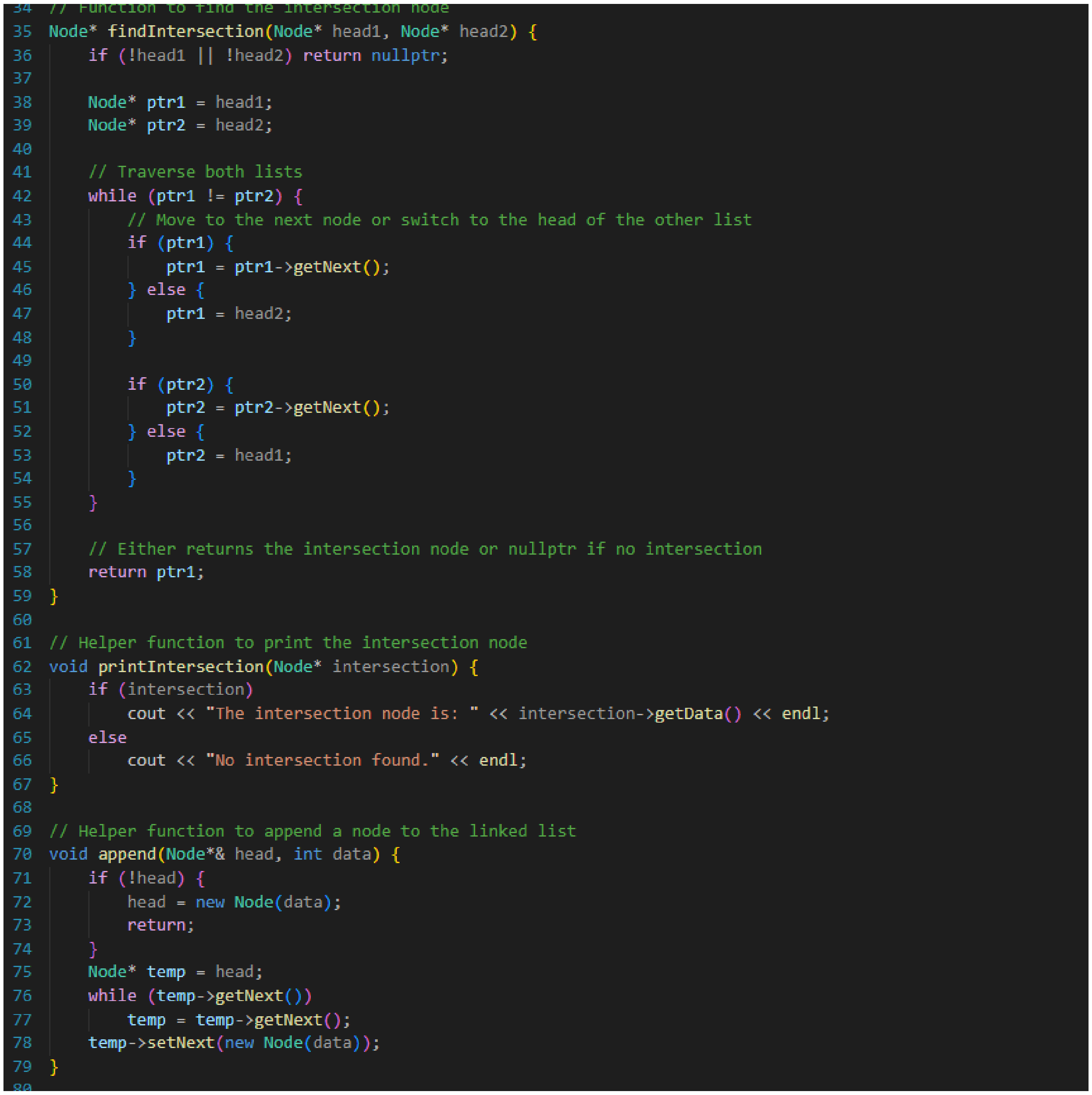
Suppose there are two singly linked lists both of which intersect at some node (as shown below) and become the same list. The number of nodes in each list before they intersect is not known and both may have it different. List! may have m nodes before it reaches intersection node and list2 may have n nodes where m and n may be m=n, m<n or m>n.

Construct an algorithm to find the merging node is as given below but its time complexity is O(mn).

Please give an algorithm that improves the performance of the above problem to 0(m + n)

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

class Node {
private:
    int data;
    Node* next;

public:
    // Constructor
    Node(int val) : data(val), next(nullptr) {}

    // Getter for data
    int getData() {
        return data;
    }

    // Setter for data
    void setData(int val) {
        data = val;
    }

    // Getter for next
    Node* getNext() {
        return next;
    }

    // Setter for next
    void setNext(Node* nextNode) {
        next = nextNode;
    }
};

```

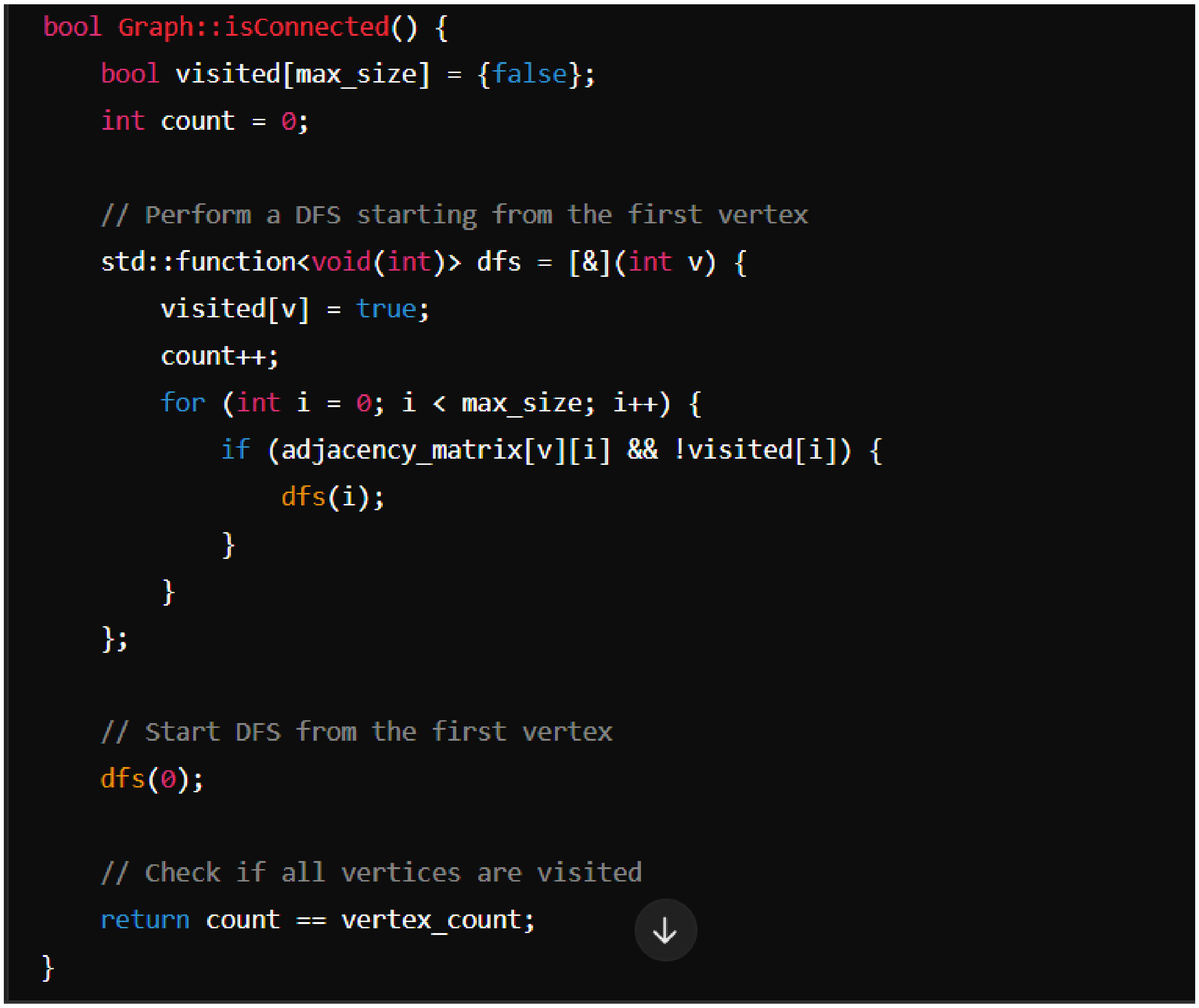
```cpp
// Function to find the intersection node

Node* findIntersection(Node* head1, Node* head2) {
    if (!head1 || !head2) return nullptr;

    Node* ptr1 = head1;
    Node* ptr2 = head2;

    // Traverse both lists

    while (ptr1 != ptr2) {
        // Move to the next node or switch to the head of the other list

        if (ptr1) {
            ptr1 = ptr1->getNext();
        } else {
            ptr1 = head2;
        }

        if (ptr2) {
            ptr2 = ptr2->getNext();
        } else {
            ptr2 = head1;
        }
    }

    // Either returns the intersection node or nullptr if no intersection
    return ptr1;
}


// Helper function to print the intersection node

void printIntersection(Node* intersection) {
    if (intersection)
        cout << "The intersection node is: " << intersection->getData() << endl;
    else
        cout << "No intersection found." << endl;
}

// Helper function to append a node to the linked list

void append(Node*& head, int data) {
    if (!head) {
        head = new Node(data);
        return;
    }
    Node* temp = head;
    while (temp->getNext())
        temp = temp->getNext();
    temp->setNext(new Node(data));
}

int main() {
    // Create two linked lists
    Node* list1 = nullptr;
    Node* list2 = nullptr;

    // Populate the first list
    append(list1, 1);
    append(list1, 2);
    append(list1, 3);

    // Populate the second list
    append(list2, 6);
    append(list2, 7);

    // Create the intersection point
    Node* common = new Node(8);
    list1->getNext()->getNext()->setNext(common); // Link common node to list1
    list2->getNext()->setNext(common);           // Link common node to list2

    // Continue the common list
    append(common, 9);
    append(common, 10);

    // Find the intersection
    Node* intersection = findIntersection(list1, list2);

    // Print the result
    printIntersection(intersection);

    return 0;
}

```



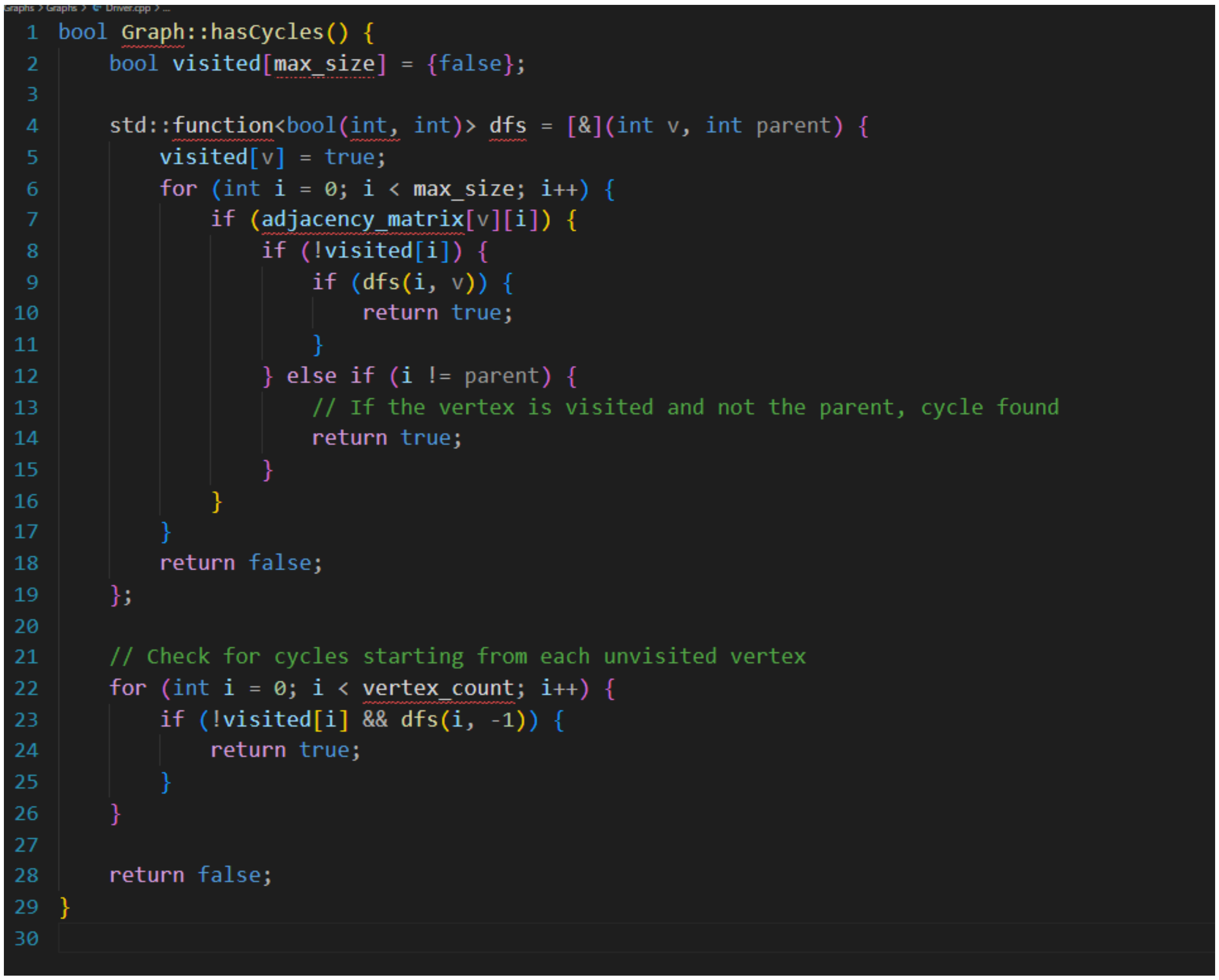
```cpp
// Function to find the intersection node
Node* findIntersection(Node* head1, Node* head2) {
    if (!head1 || !head2) return nullptr;

    Node* ptr1 = head1;
    Node* ptr2 = head2;

    // Traverse both lists
    while (ptr1 != ptr2) {
        // Move to the next node or switch to the head of the other list
        if (ptr1) {
            ptr1 = ptr1->getNext();
        } else {
            ptr1 = head2;
        }

        if (ptr2) {
            ptr2 = ptr2->getNext();
        } else {
            ptr2 = head1;
        }
    }

    // Either returns the intersection node or nullptr if no intersection
    return ptr1;
}

// Helper function to print the intersection node
void printIntersection(Node* intersection) {
    if (intersection)
        cout << "The intersection node is: " << intersection->getData() << endl;
    else
        cout << "No intersection found." << endl;
}

// Helper function to append a node to the linked list
void append(Node*& head, int data) {
    if (!head) {
        head = new Node(data);
        return;
    }
    Node* temp = head;
    while (temp->getNext())
        temp = temp->getNext();
    temp->setNext(new Node(data));
}
```

---

## ➢ GRAPH



```cpp
bool Graph::isConnected() {
    bool visited[max_size] = {false};
    int count = 0;

    // Perform a DFS starting from the first vertex
    std::function<void(int)> dfs = [&](int v) {
        visited[v] = true;
        count++;
        for (int i = 0; i < max_size; i++) {
            if (adjacency_matrix[v][i] && !visited[i]) {
                dfs(i);
            }
        }
    };

    // Start DFS from the first vertex
    dfs(0);

    // Check if all vertices are visited
    return count == vertex_count;
}
```



```cpp
bool Graph::hasCycles() {
    bool visited[max_size] = {false};

    std::function<bool(int, int)> dfs = [&](int v, int parent) {
        visited[v] = true;
        for (int i = 0; i < max_size; i++) {
            if (adjacency_matrix[v][i]) {
                if (!visited[i]) {
                    if (dfs(i, v)) {
                        return true;
                    }
                } else if (i != parent) {
                    // If the vertex is visited and not the parent, cycle found
                    return true;
                }
            }
        }
        return false;
    };

    // Check for cycles starting from each unvisited vertex
    for (int i = 0; i < vertex_count; i++) {
        if (!visited[i] && dfs(i, -1)) {
            return true;
        }
    }

    return false;
}
```

```
1. Explanation

2. isConnected:
3. A DFS is used to traverse the graph starting from vertex 0.
4. An array visited keeps track of which vertices are visited.
5. After the traversal, if the number of visited vertices equals vertex_count, the graph is
connected.


6. hasCycles:
7. A DFS is used to detect cycles by checking back edges.
8. The parent parameter ensures that a node isn't revisited from its immediate parent during the
DFS, which helps identify true cycles.
```

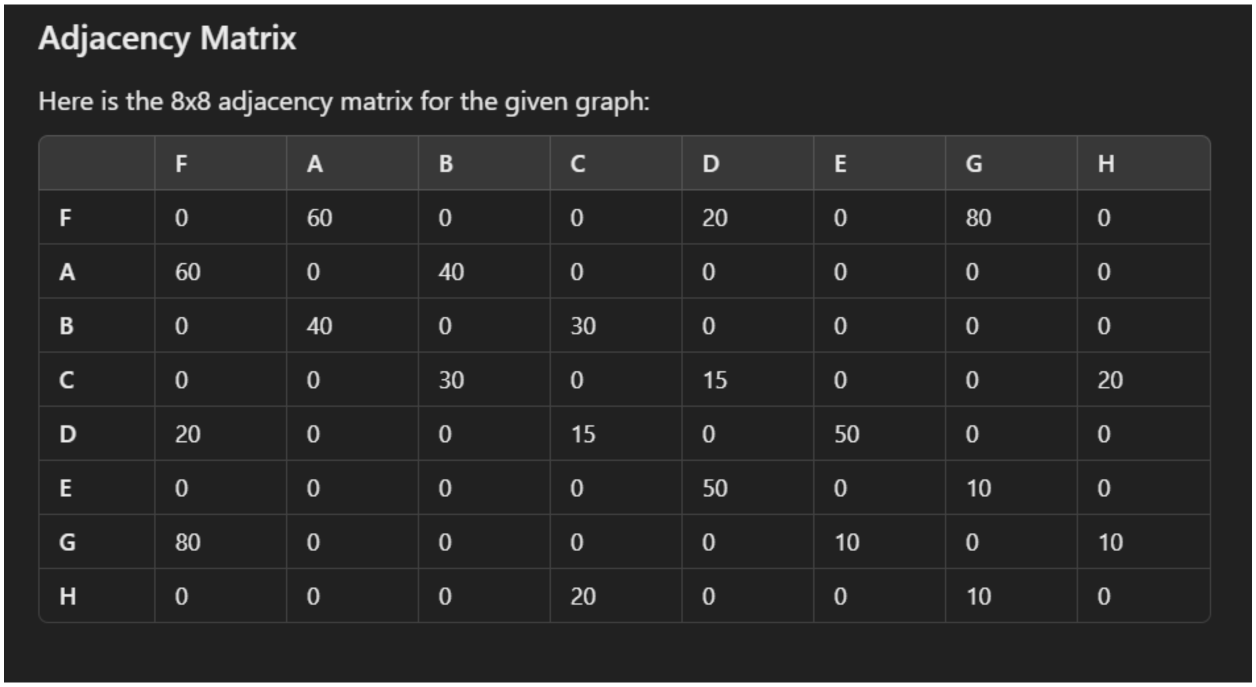
---

## ➢ FARMER ...

To generate the adjacency matrix for the given graph, follow these steps:

1. **Understand the Problem:**
   - The nodes are labeled as **F, A, B, C, H, G, D, E**.
   - Each edge between two nodes is weighted by the number of steps between them.
   - The graph is undirected, so the adjacency matrix will be symmetric.
2. **Nodes in Order:** Let's label the nodes in order: **F, A, B, C, D, E, G, H**.
3. **Edge List:** Based on the information:
   - F to A: 60
   - A to B: 40
   - B to C: 30
   - C to H: 20
   - H to G: 10
   - G to F: 80
   - F to D: 20
   - D to E: 50
   - E to G: 10
   - C to D: 15
4. **Adjacency Matrix:** An adjacency matrix is a 2D array where:
   - Rows and columns correspond to the nodes.
   - The value at matrix[i][j] represents the weight of the edge between node i and node j (or 0 if no edge exists).

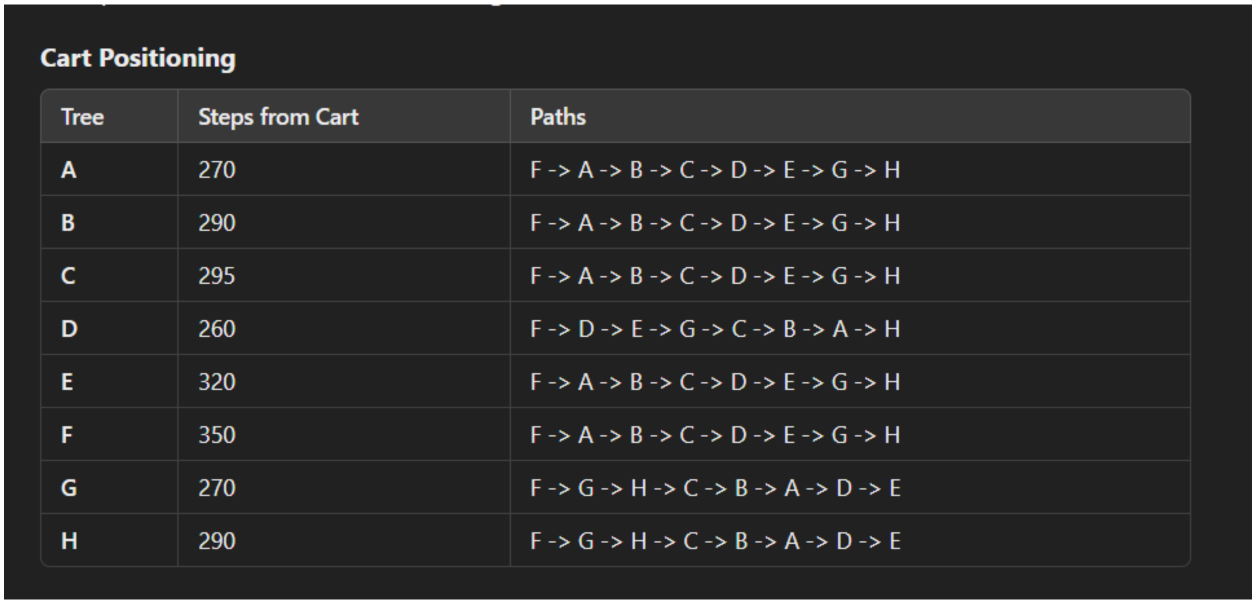
```
Explanation of the Matrix:
The value at matrix[F][A] is 60 because the distance from F to A is 60 steps.
The matrix is symmetric because the graph is undirected. For example, matrix[F][A] = matrix[A][F] = 60.
For nodes with no direct edges (e.g., F and B), the value is 0.
This adjacency matrix fully represents the graph.

```



## Adjacency Matrix

Here is the 8x8 adjacency matrix for the given graph:

| | F | A | B | C | D | E | G | H |
|---|---|---|---|---|---|---|---|---|
| **F** | 0 | 60 | 0 | 0 | 20 | 0 | 80 | 0 |
| **A** | 60 | 0 | 40 | 0 | 0 | 0 | 0 | 0 |
| **B** | 0 | 40 | 0 | 30 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 30 | 0 | 15 | 0 | 0 | 20 |
| **D** | 20 | 0 | 0 | 15 | 0 | 50 | 0 | 0 |
| **E** | 0 | 0 | 0 | 0 | 50 | 0 | 10 | 0 |
| **G** | 80 | 0 | 0 | 0 | 0 | 10 | 0 | 10 |
| **H** | 0 | 0 | 0 | 20 | 0 | 0 | 10 | 0 |



**Cart Positioning**

| Tree | Steps from Cart | Paths |
|---|---|---|
| **A** | 270 | F -> A -> B -> C -> D -> E -> G -> H |
| **B** | 290 | F -> A -> B -> C -> D -> E -> G -> H |
| **C** | 295 | F -> A -> B -> C -> D -> E -> G -> H |
| **D** | 260 | F -> D -> E -> G -> C -> B -> A -> H |
| **E** | 320 | F -> A -> B -> C -> D -> E -> G -> H |
| **F** | 350 | F -> A -> B -> C -> D -> E -> G -> H |
| **G** | 270 | F -> G -> H -> C -> B -> A -> D -> E |
| **H** | 290 | F -> G -> H -> C -> B -> A -> D -> E |

**Explanation:**

```
Tree D gives the lowest total steps (260), which makes it the best tree to park the cart at.
The optimal path from Tree D to collect all mangoes is:
Start at F, walk to D.
From D, walk to E.
From E, walk to G.
From G, walk to C.
From C, walk to B.
From B, walk to A.
Finally, walk to H.
This is the optimal path with the least walking distance for the farmer to collect all the
mangoes.
```

------------------------------------------------------------------------------------------------------------------------

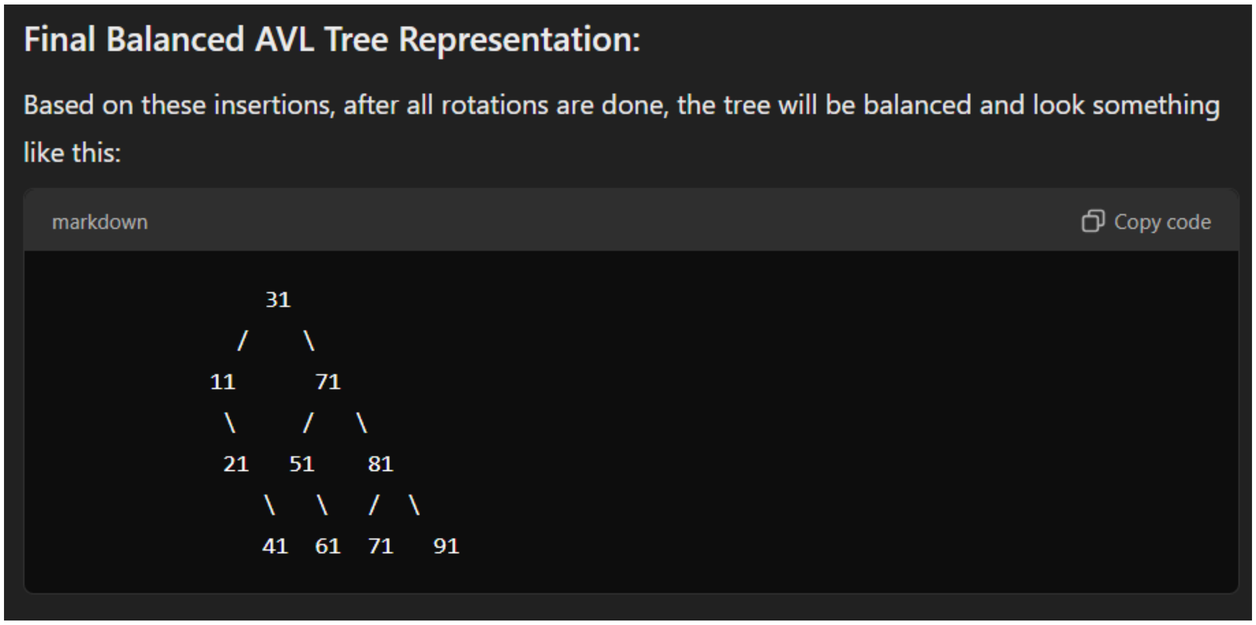
Following code runs on an AVL Tree template class. Construct draw the ima representation if the tree.

```
AVLTree<int> A;
for(int i=1;i<100;i=i+10)
 A.insert(i)

```

AVL tree are: 1, 11, 21, 31, 41, 51, 61, 71, 81, and 91

**Insertions:**

```
Insert 1: The tree is empty, so the root becomes 1.
Insert 11: Inserted as the right child of 1, no need for balancing.
Insert 21: Inserted as the right child of 11, and a right rotation on 1 is required to balance.
Insert 31: Inserted as the right child of 21, no need for balancing.
Insert 41: Inserted as the right child of 31, and a rotation is performed to maintain balance.
Insert 51: Inserted as the right child of 41, and further rotations are performed to keep the
tree balanced.
Insert 61: Inserted as the right child of 51, followed by necessary rotations.
Insert 71: Inserted as the right child of 61, requiring balancing.
Insert 81: Inserted as the right child of 71, requiring further rotations.
Insert 91: Inserted as the right child of 81, requiring balancing to ensure the tree remains
balanced.


```


**Final Balanced AVL Tree Representation:**

Based on these insertions, after all rotations are done, the tree will be balanced and look something like this:

```
              31
            /    \
          11      71
           \     /   \
           21   51     81
              \   \   /  \
              41  61  71   91
```

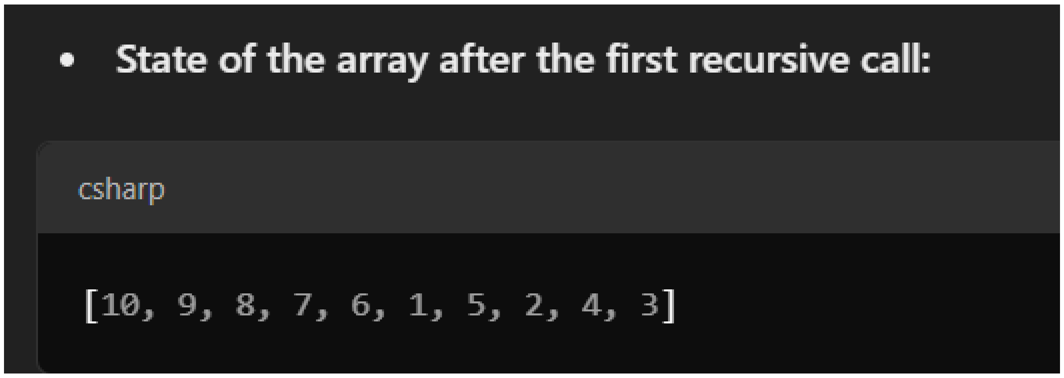
**Explanation:**

```
1. 1, 11, 21: Inserted in order, requiring rotations to balance.

2. 31 becomes the root after rotations to balance the left and right subtrees.

3. As we continue inserting, balancing occurs after each insertion, with the tree rotating as needed
to maintain its AVL properties.

4. The final tree is height-balanced with the values distributed evenly across both left and right
subtrees.

5. This diagram represents the tree after all insertions, with the correct balance maintained
through the AVL rotations.
```

---

**Apply quick sort algorithm to sort the array given below in descending order, using the first element of the array and subarrays as pivot. 6 1 5 7 9 10 8 2 4 3**

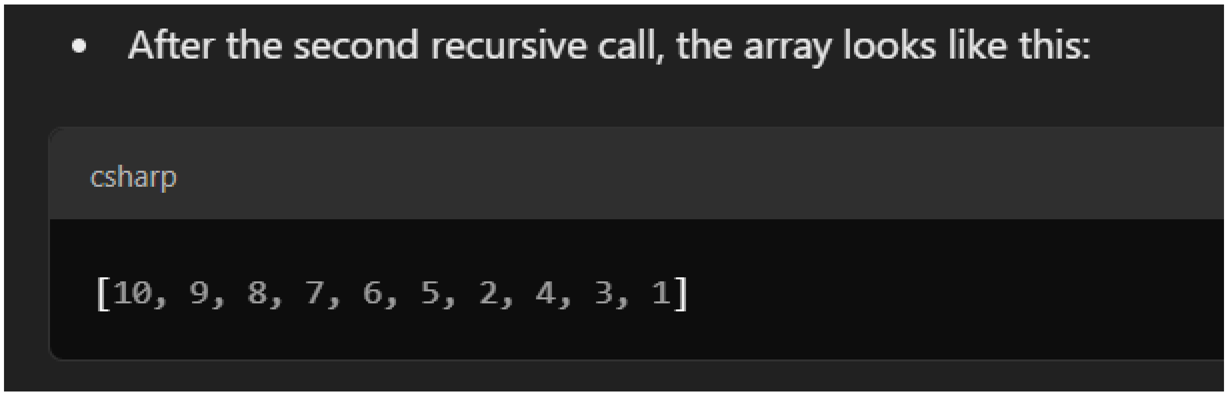
**Show the state of the array after the first recursive call of the quicksort function.**



- State of the array after the first recursive call:

```csharp
[10, 9, 8, 7, 6, 1, 5, 2, 4, 3]
```

**Show the state of the array after the second call of the quicksort function.**

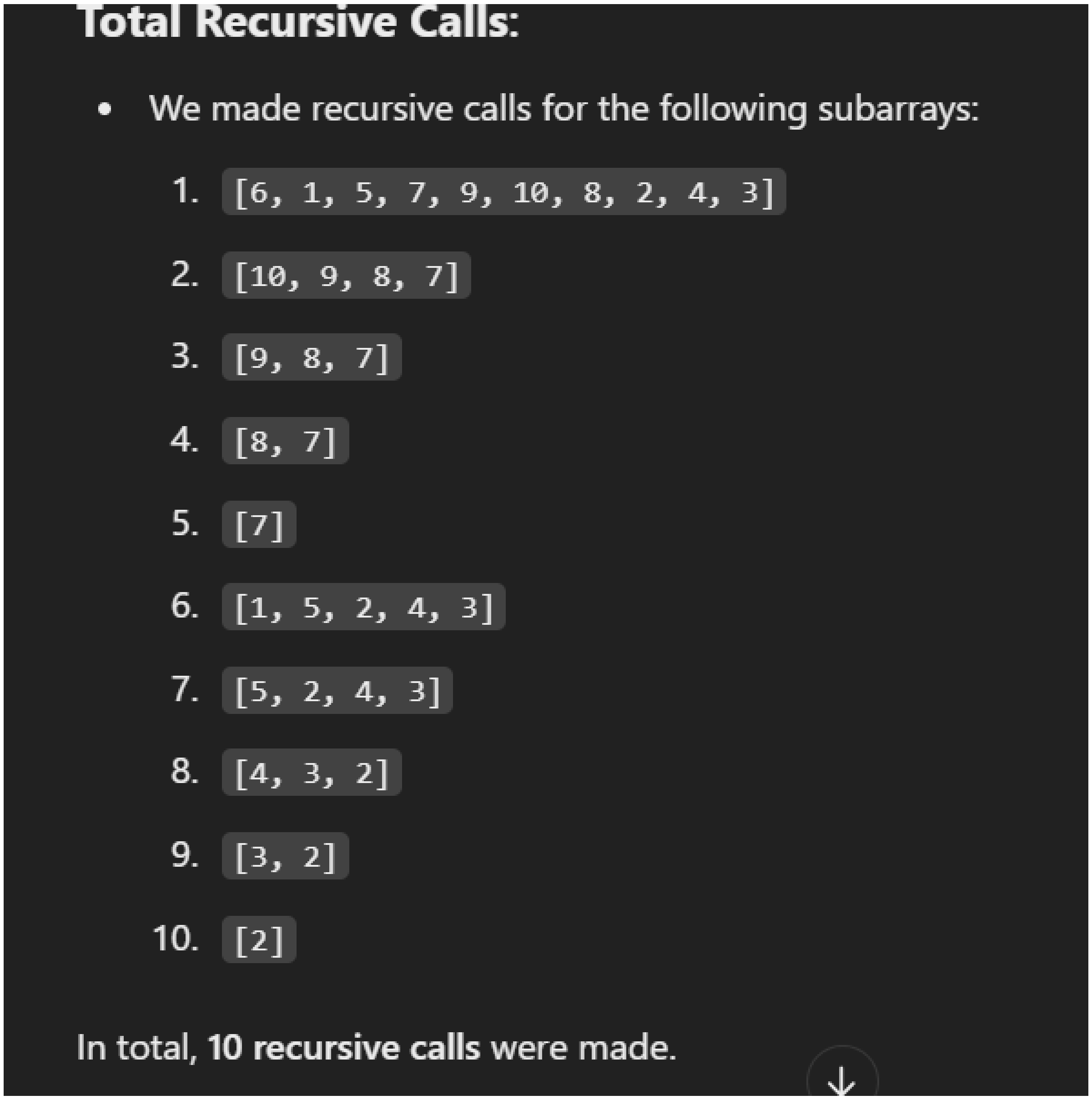


- After the second recursive call, the array looks like this:

```csharp
[10, 9, 8, 7, 6, 5, 2, 4, 3, 1]
```

**The quicksort function would be called how many times?**

```
Total Number of Recursive Calls:

First call: To partition the array around 6.
Second call: For the left subarray [10, 9, 8, 7] with pivot 10 and no further partitioning needed.
Third call: For the right subarray [1, 5, 2, 4, 3] with pivot 1 and no further partitioning needed.

Additional recursive calls will be made for the subarrays left of 10 (which is an empty subarray) and right of 1 (which is also an empty subarray).

So, the total number of recursive calls will be 3.

Conclusion:
State after first recursive call: [10, 9, 8, 7, 6, 1, 5, 2, 4, 3]
State after second recursive call: [10, 9, 8, 7, 6, 5, 2, 4, 3, 1]
Total number of recursive calls: 3

```



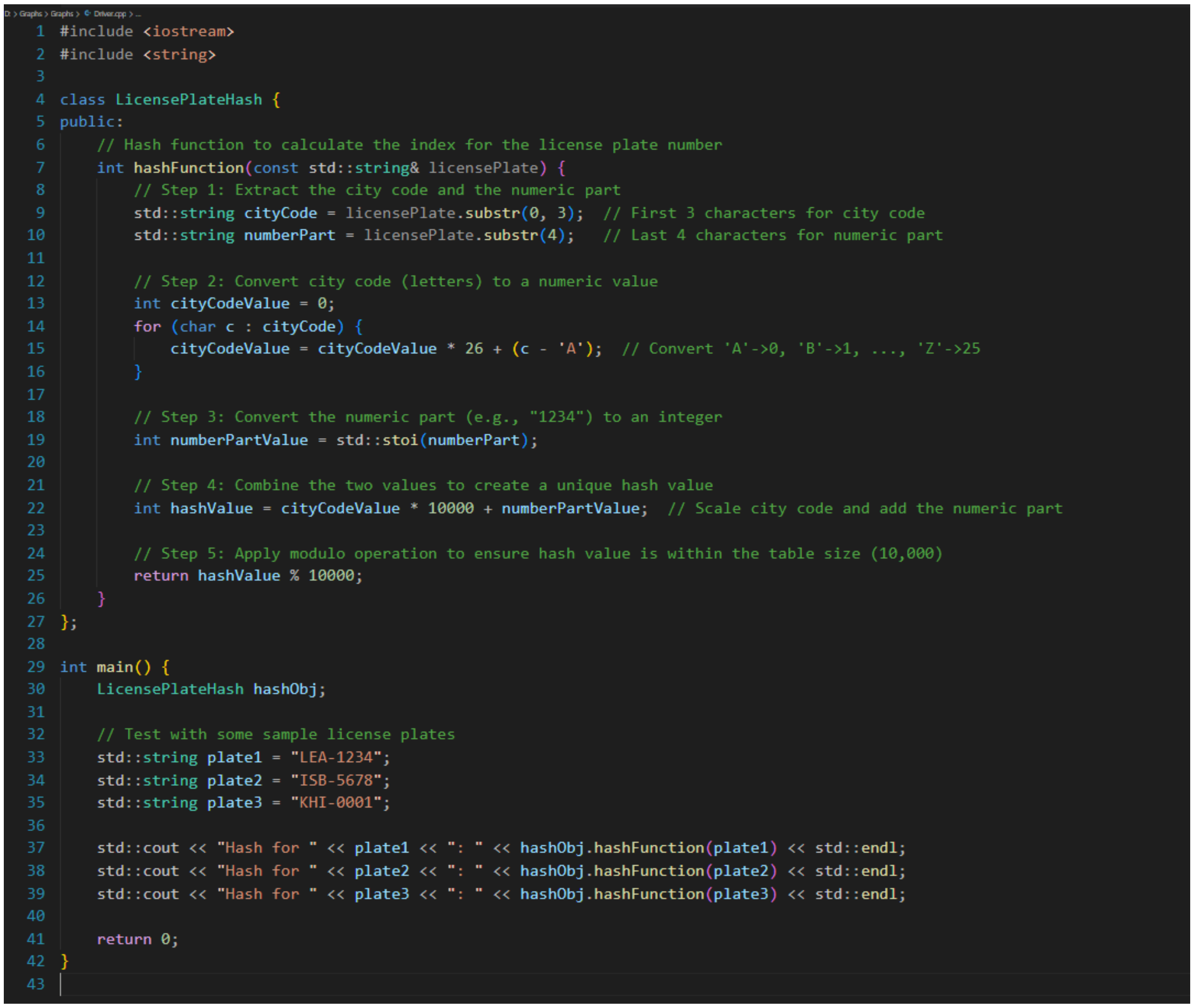
**Total Recursive Calls:**

- We made recursive calls for the following subarrays:
  1. `[6, 1, 5, 7, 9, 10, 8, 2, 4, 3]`
  2. `[10, 9, 8, 7]`
  3. `[9, 8, 7]`
  4. `[8, 7]`
  5. `[7]`
  6. `[1, 5, 2, 4, 3]`
  7. `[5, 2, 4, 3]`
  8. `[4, 3, 2]`
  9. `[3, 2]`
  10. `[2]`

In total, **10 recursive calls** were made.

--------------------------------------------------------------------------------------------------

➢ License plate number of vehicles in Pakistan are alphanumeric of the form of LEA-1234, The first The letters represent the city of registration, and the remaining are incremented from A-0001 f4 The first Vehicle registered. The traffic police want to implement the record in a hash table. The maximum size of hash table is 10 thousand records. Write (construct) the code for a hash function, which takes the number plate value as key of datatype string and returns an appropriate integer for data storage and retrieval.



D: > Graphs > Graphs > Driver.cpp > ...

```cpp
#include <iostream>
#include <string>

class LicensePlateHash {
public:
    // Hash function to calculate the index for the license plate number
    int hashFunction(const std::string& licensePlate) {
        // Step 1: Extract the city code and the numeric part
        std::string cityCode = licensePlate.substr(0, 3);  // First 3 characters for city code
        std::string numberPart = licensePlate.substr(4);   // Last 4 characters for numeric part

        // Step 2: Convert city code (letters) to a numeric value
        int cityCodeValue = 0;
        for (char c : cityCode) {
            cityCodeValue = cityCodeValue * 26 + (c - 'A');  // Convert 'A'->0, 'B'->1, ..., 'Z'->25
        }

        // Step 3: Convert the numeric part (e.g., "1234") to an integer
        int numberPartValue = std::stoi(numberPart);

        // Step 4: Combine the two values to create a unique hash value
        int hashValue = cityCodeValue * 10000 + numberPartValue;  // Scale city code and add the numeric part

        // Step 5: Apply modulo operation to ensure hash value is within the table size (10,000)
        return hashValue % 10000;
    }
};

int main() {
    LicensePlateHash hashObj;

    // Test with some sample license plates
    std::string plate1 = "LEA-1234";
    std::string plate2 = "ISB-5678";
    std::string plate3 = "KHI-0001";

    std::cout << "Hash for " << plate1 << ": " << hashObj.hashFunction(plate1) << std::endl;
    std::cout << "Hash for " << plate2 << ": " << hashObj.hashFunction(plate2) << std::endl;
    std::cout << "Hash for " << plate3 << ": " << hashObj.hashFunction(plate3) << std::endl;

    return 0;
}
```