Hackerearth
Hub-nmamit

WEEKLY CONTEST 5 SOLUTIONS

Problem 1: No Common Element

Approach:

- Use two hash maps to count the frequency of elements in both arrays (count1 for array 1 and count2 for array 2).
- Iterate through all elements in count1.
- For each element that appears in both maps, find the minimum frequency between the two arrays.
- Add this minimum frequency to a result variable (res), which keeps track of the number of common elements to be removed.
- The value of res represents the minimum number of removals needed to eliminate all common elements between the two arrays.

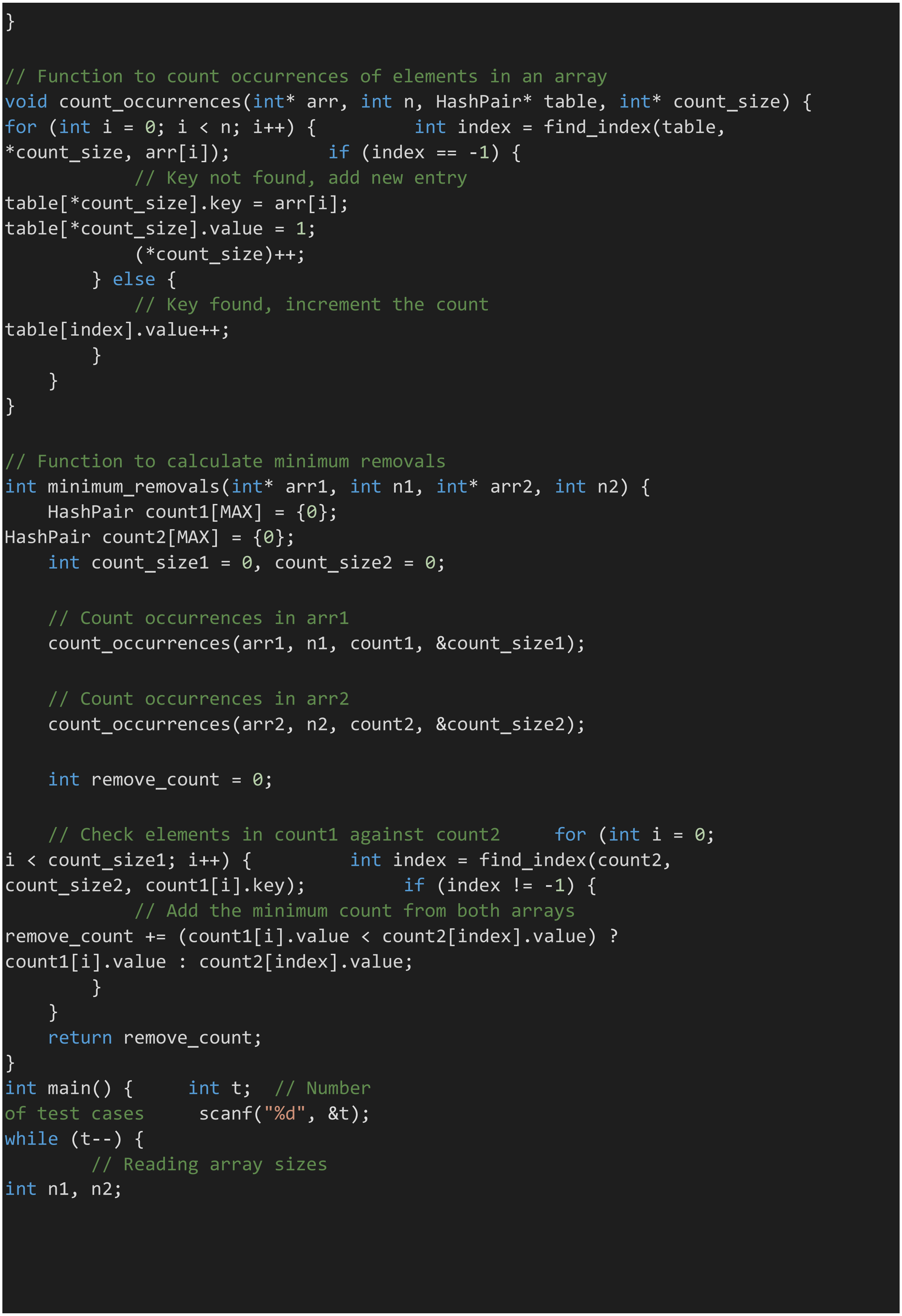
Reference Video : Click Here

PYTHON :

```
def minimum_removals(arr1, arr2):
    # Count occurrences of elements in arr1 and arr2
count1 = {}     count2 = {}
         for num in arr1:
if num in count1:
count1[num] += 1
else:
count1[num] = 1
                 for num in
arr2:         if num in
count2:
count2[num] += 1
else:
count2[num] = 1

    # Find common elements and calculate the total removals needed
    remove_count = 0
         for element in
count1:         if element in
count2:
            # Add the minimum count from both arrays
            remove_count += min(count1[element], count2[element])

    return remove_count

# Input and Output processing
n = int(input())  # Number of test
cases for _ in range(n):     # Reading
array sizes
    n1, n2 = map(int, input().split())


    # Reading the arrays
if n1!=0:
        arr1 = list(map(int, input().split()))
if n2!=0:
        arr2 = list(map(int, input().split()))
         if n1==0 or
n2==0:
        print(0)
continue
    # Call the function and print the
result     print(minimum_removals(arr1,
arr2))
```

C :

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

#define MAX 1000

// Structure to hold a key-value pair for the hash
table typedef struct {     int key;     int value;
} HashPair;

// Function to find the index of a key in the hash
table int find_index(HashPair* table, int size, int
key) {     for (int i = 0; i < size; i++) {         if
(table[i].key == key) {             return i;
        }
}
    return -1; // Key not found
```



```
}

// Function to count occurrences of elements in an array
void count_occurrences(int* arr, int n, HashPair* table, int* count_size) {
for (int i = 0; i < n; i++) {         int index = find_index(table,
*count_size, arr[i]);         if (index == -1) {
            // Key not found, add new entry
table[*count_size].key = arr[i];
table[*count_size].value = 1;
            (*count_size)++;
        } else {
            // Key found, increment the count
table[index].value++;
        }
    }
}

// Function to calculate minimum removals
int minimum_removals(int* arr1, int n1, int* arr2, int n2) {
    HashPair count1[MAX] = {0};
HashPair count2[MAX] = {0};
    int count_size1 = 0, count_size2 = 0;

    // Count occurrences in arr1
    count_occurrences(arr1, n1, count1, &count_size1);

    // Count occurrences in arr2
    count_occurrences(arr2, n2, count2, &count_size2);

    int remove_count = 0;

    // Check elements in count1 against count2     for (int i = 0;
i < count_size1; i++) {         int index = find_index(count2,
count_size2, count1[i].key);         if (index != -1) {
            // Add the minimum count from both arrays
remove_count += (count1[i].value < count2[index].value) ?
count1[i].value : count2[index].value;
        }
    }
    return remove_count;
}
int main() {     int t;  // Number
of test cases     scanf("%d", &t);
while (t--) {
        // Reading array sizes
int n1, n2;
```

```
        scanf("%d %d", &n1, &n2);

        // Reading the arrays
int arr1[MAX], arr2[MAX];         for
(int i = 0; i < n1; i++) {
            scanf("%d", &arr1[i]); // Read arr1
        }
        for (int i = 0; i < n2; i++) {
            scanf("%d", &arr2[i]); // Read arr2
        }

        // If either array is empty, print
0         if (n1 == 0 || n2 == 0) {
printf("0\n");              continue;
        }

        // Call the function and print the result
        printf("%d\n", minimum_removals(arr1, n1, arr2, n2));
    }
    return 0;
}
```

C++ :

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>

using namespace std;

int minimum_removals(const vector<int>& arr1, const vector<int>& arr2) {
    // Count occurrences of elements in arr1
unordered_map<int, int> count1;

    for (int num : arr1) {
count1[num]++;
    }

    // Count occurrences of elements in
arr2     unordered_map<int, int> count2;
for (int num : arr2) {
count2[num]++;
    }

    // Variable to keep track of the total removals needed
int remove_count = 0;
// Check elements in count1 against count2
for(const auto& element : count1) {          auto it =
count2.find(element.first);  // Use find() for O(1) lookup          if (it !=
count2.end()) {
            // Add the minimum count from both arrays
remove_count += min(element.second, it->second);
        }
    }
    return remove_count;
}
int main() {      int n;  // Number
of test cases     cin >> n;
while (n--) {
        // Reading array sizes
int n1, n2;         cin >> n1
>> n2;

        // Reading the arrays         vector<int>
arr1(n1), arr2(n2);         for (int& x : arr1)
cin >> x; // Read arr1          for (int& x : arr2)
cin >> x; // Read arr2

        // If either array is empty, print
0          if (n1 == 0 || n2 == 0) {
cout << 0 << endl;              continue;
        }

        // Call the function and print the result
cout << minimum_removals(arr1, arr2) << endl;
    }
    return 0;
}
```

JAVA :

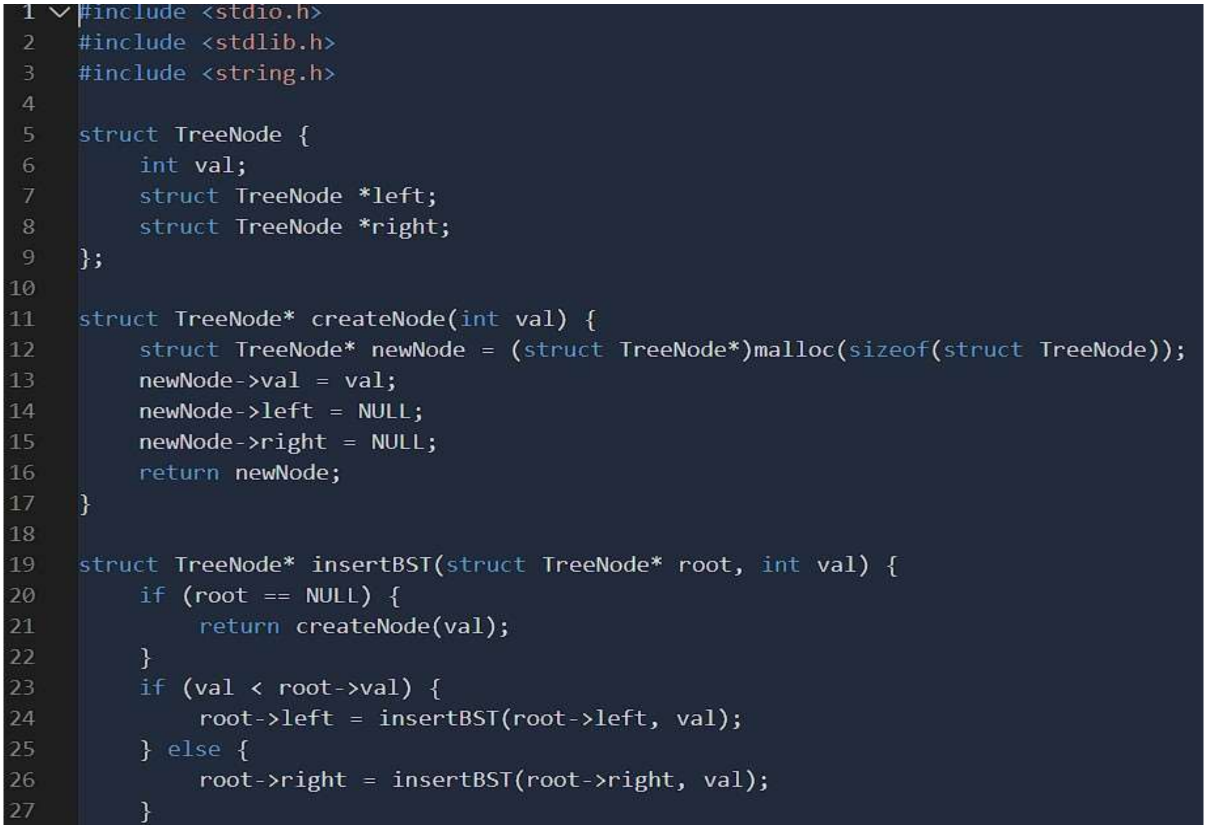
```
import java.util.HashMap;
import java.util.Map; import
java.util.Scanner;

public class Main {      public static int minimumRemovals(int[]
arr1, int[] arr2) {
        // Count occurrences of elements in arr1
        HashMap<Integer, Integer> count1 = new HashMap<>();

        for (int num : arr1) {             count1.put(num,
count1.getOrDefault(num, 0) + 1);
```

```
        }

        // Count occurrences of elements in arr2
        HashMap<Integer, Integer> count2 = new HashMap<>();
for (int num : arr2) {             count2.put(num,
count2.getOrDefault(num, 0) + 1);
        }

        // Variable to keep track of the total removals needed
int removeCount = 0;

        // Check elements in count1 against count2
        for (Map.Entry<Integer, Integer> element : count1.entrySet()) {
Integer countInArr2 = count2.getOrDefault(element.getKey(),0);
removeCount += Math.min(element.getValue(), countInArr2);
        }

        return removeCount;
    }
    public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
int n;  // Number of test cases         n =
scanner.nextInt();          while (n-- > 0) {
            // Reading array sizes
int n1 = scanner.nextInt();
int n2 = scanner.nextInt();

            // Reading the arrays             int[] arr1 = new
int[n1];             int[] arr2 = new int[n2];             for (int i
= 0; i < n1; i++) arr1[i] = scanner.nextInt();              for (int
i = 0; i < n2; i++) arr2[i] = scanner.nextInt();
            // If either array is empty, print 0
            if (n1 == 0 || n2 == 0) {
System.out.println(0);                 continue;
            }

            // Call the function and print the result
            System.out.println(minimumRemovals(arr1, arr2));
        }
        scanner.close();
    }
}
```

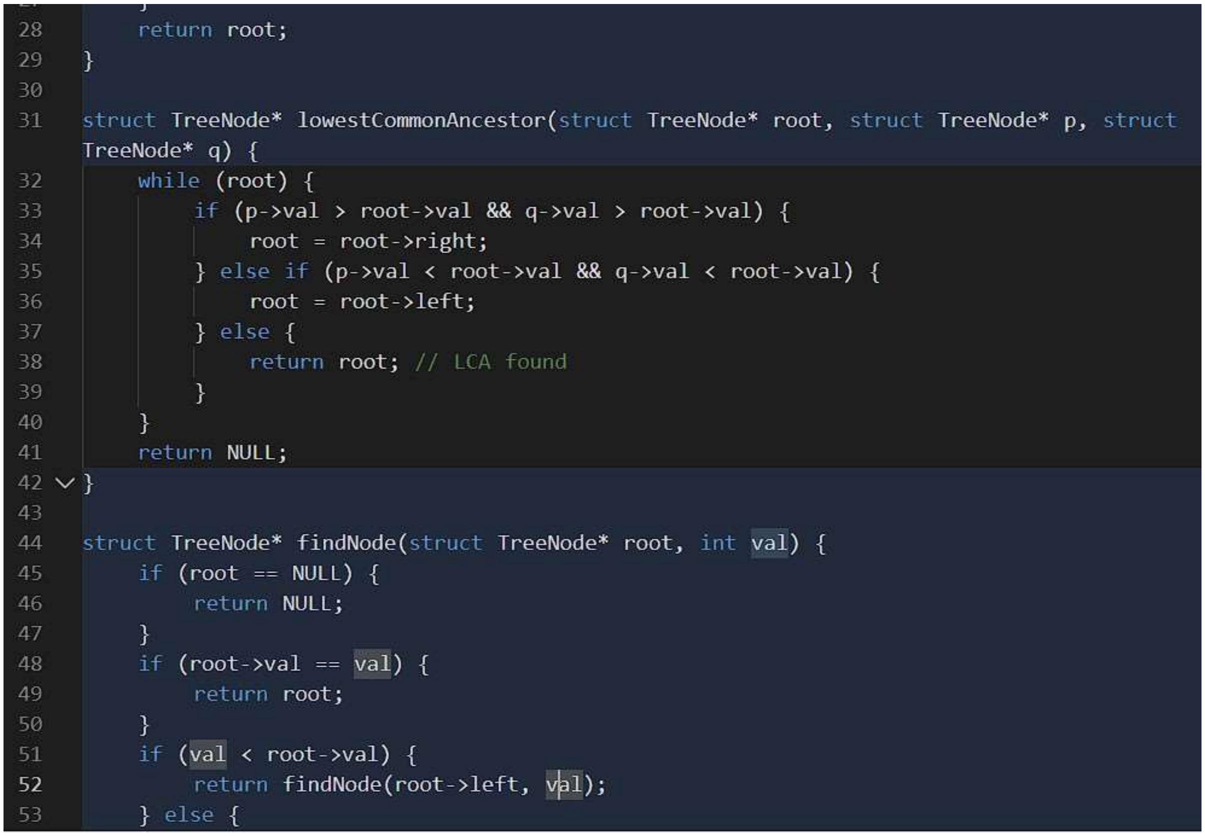
Problem 2 : Lowest Common Ancestor of a Binary Search Tree

Reference Video : Click Here

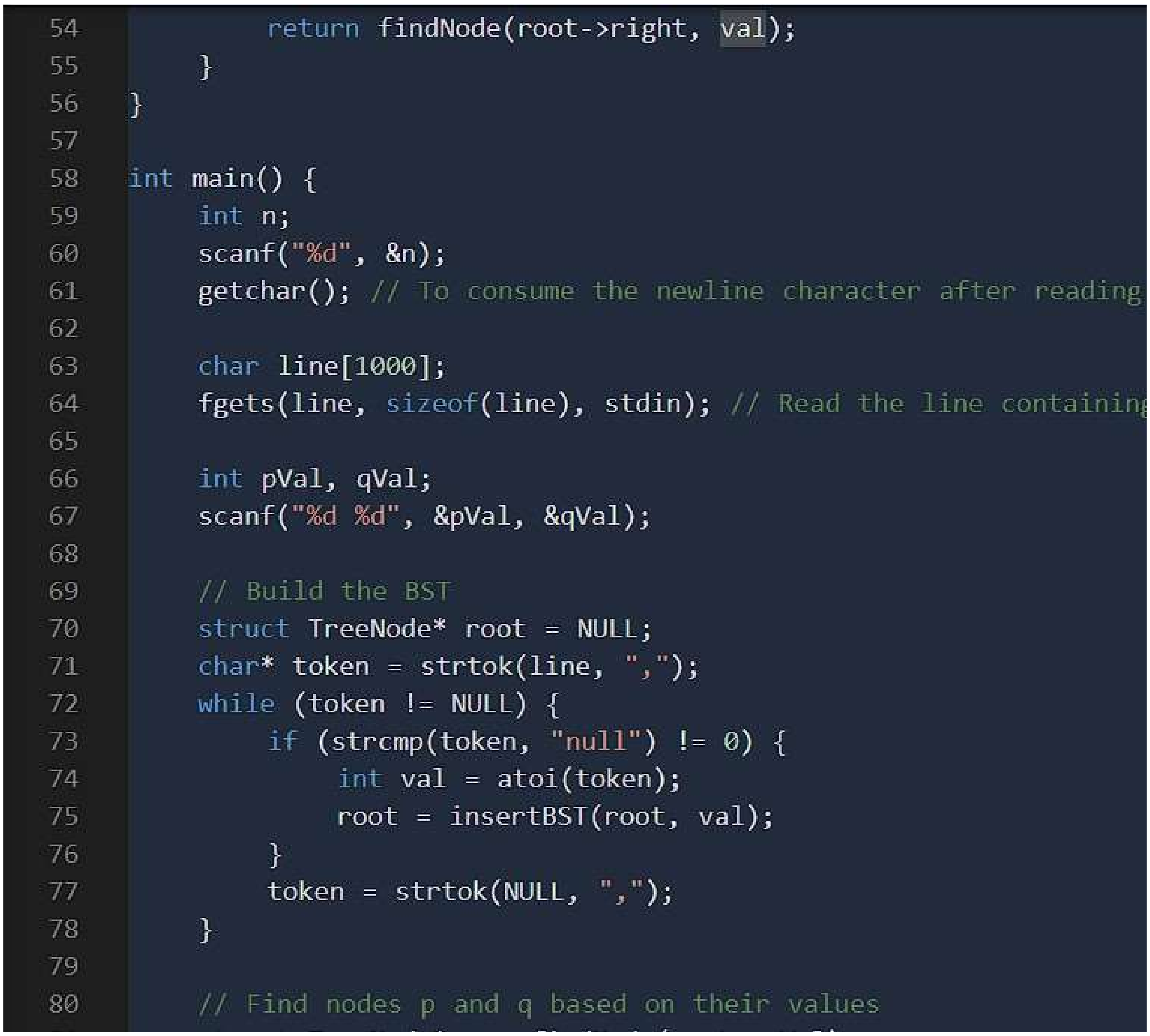
C :



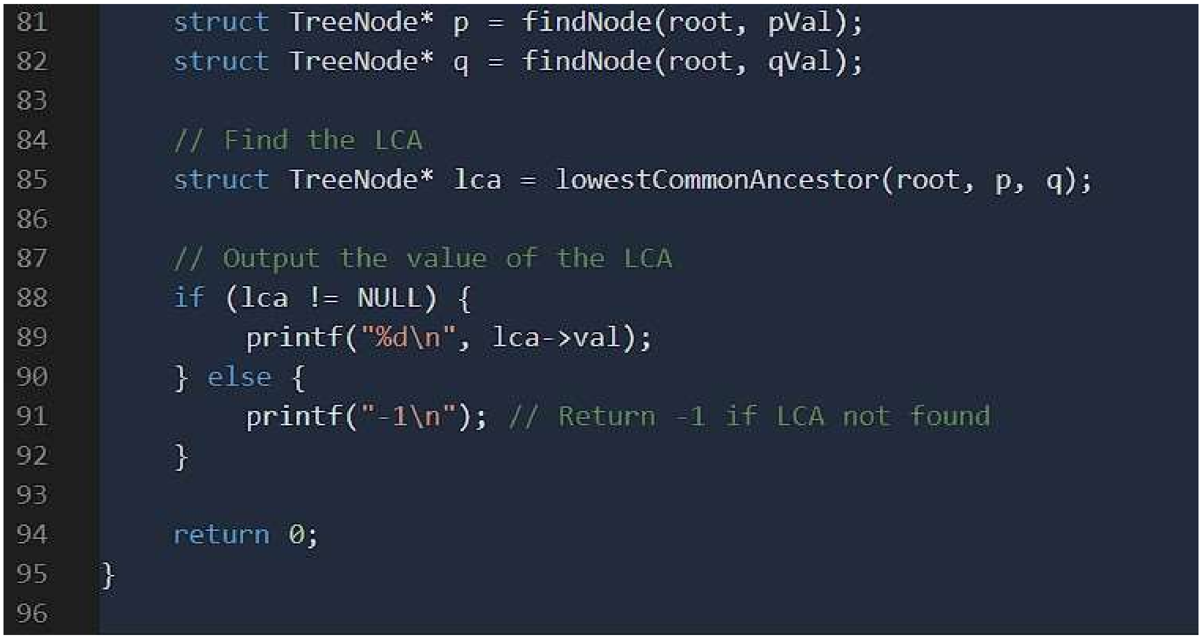
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
};

struct TreeNode* createNode(int val) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->val = val;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct TreeNode* insertBST(struct TreeNode* root, int val) {
    if (root == NULL) {
        return createNode(val);
    }
    if (val < root->val) {
        root->left = insertBST(root->left, val);
    } else {
        root->right = insertBST(root->right, val);
    }
```



```c
    return root;
}

struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct TreeNode* q) {
    while (root) {
        if (p->val > root->val && q->val > root->val) {
            root = root->right;
        } else if (p->val < root->val && q->val < root->val) {
            root = root->left;
        } else {
            return root; // LCA found
        }
    }
    return NULL;
}

struct TreeNode* findNode(struct TreeNode* root, int val) {
    if (root == NULL) {
        return NULL;
    }
    if (root->val == val) {
        return root;
    }
    if (val < root->val) {
        return findNode(root->left, val);
    } else {
```

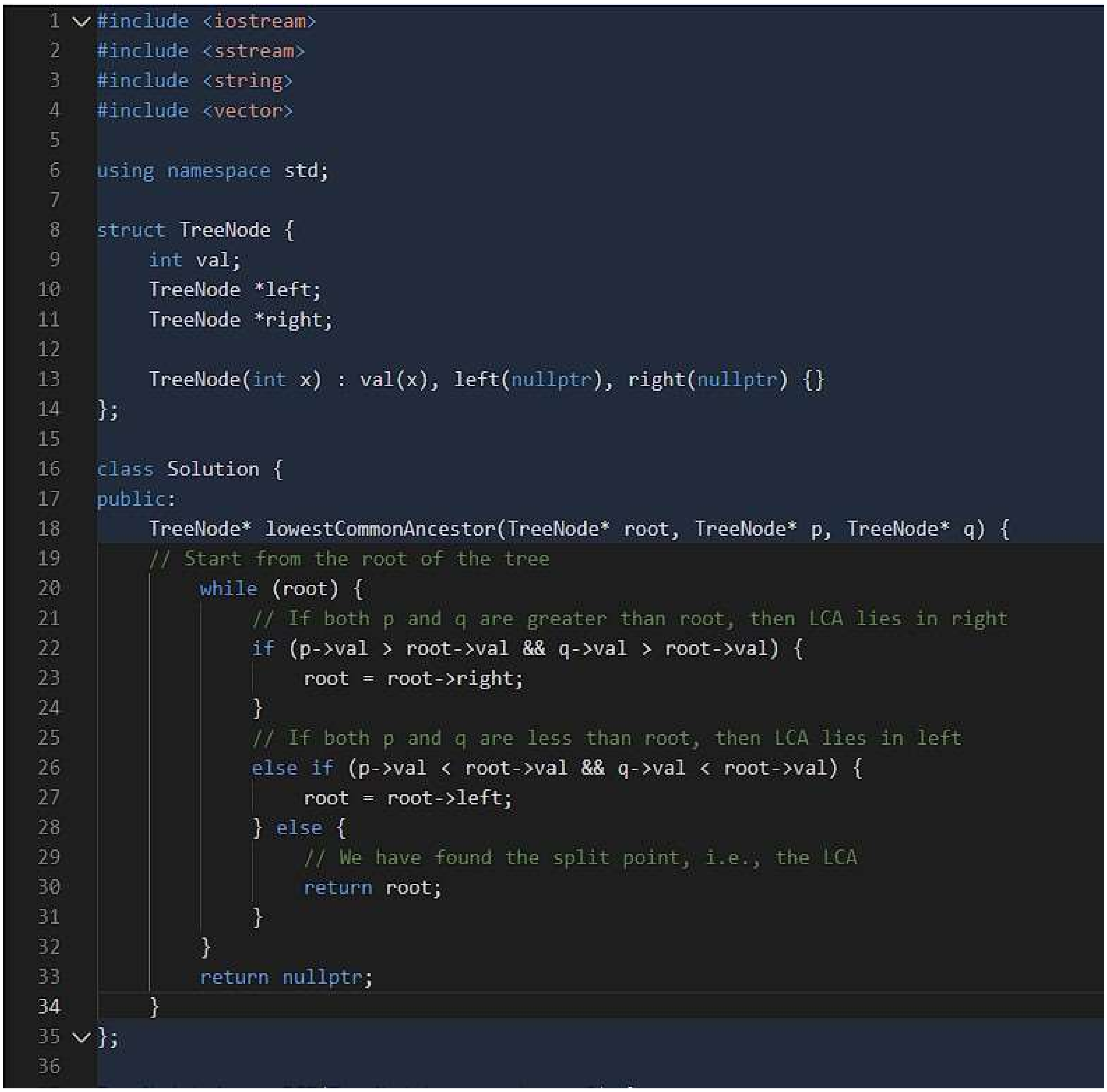


```c
        return findNode(root->right, val);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    getchar(); // To consume the newline character after reading

    char line[1000];
    fgets(line, sizeof(line), stdin); // Read the line containing

    int pVal, qVal;
    scanf("%d %d", &pVal, &qVal);

    // Build the BST
    struct TreeNode* root = NULL;
    char* token = strtok(line, ",");
    while (token != NULL) {
        if (strcmp(token, "null") != 0) {
            int val = atoi(token);
            root = insertBST(root, val);
        }
        token = strtok(NULL, ",");
    }

    // Find nodes p and q based on their values
```

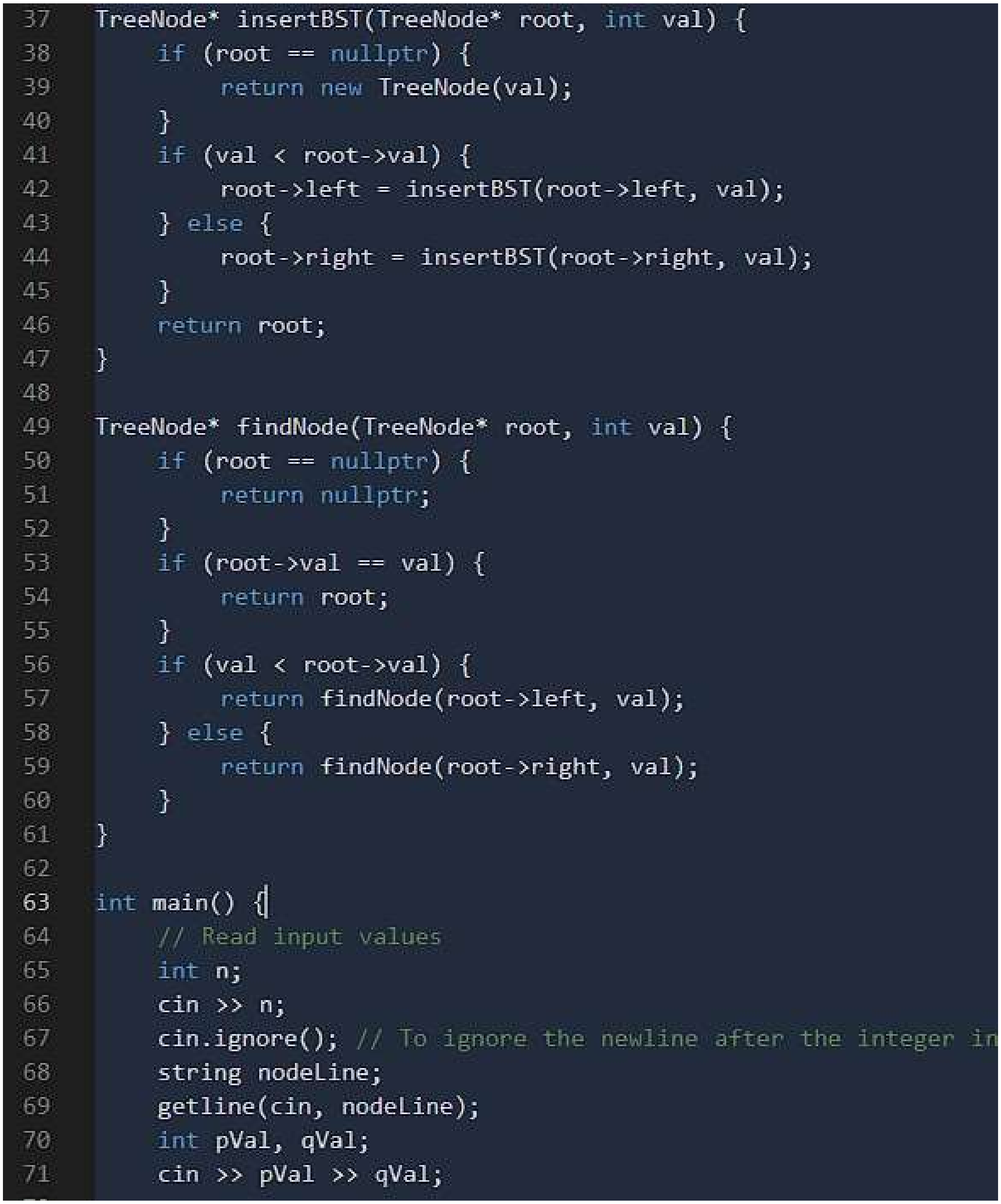


```c
    struct TreeNode* p = findNode(root, pVal);
    struct TreeNode* q = findNode(root, qVal);

    // Find the LCA
    struct TreeNode* lca = lowestCommonAncestor(root, p, q);

    // Output the value of the LCA
    if (lca != NULL) {
        printf("%d\n", lca->val);
    } else {
        printf("-1\n"); // Return -1 if LCA not found
    }

    return 0;
}
```

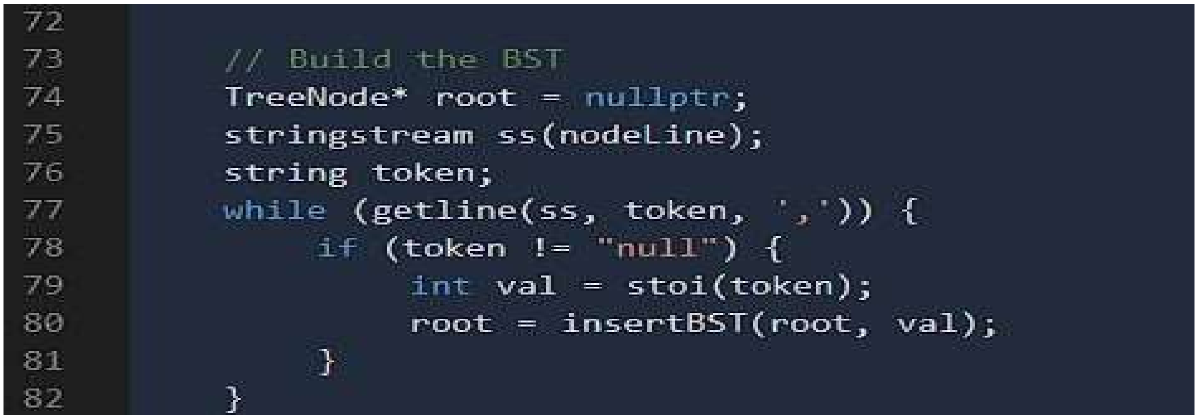
C++ :



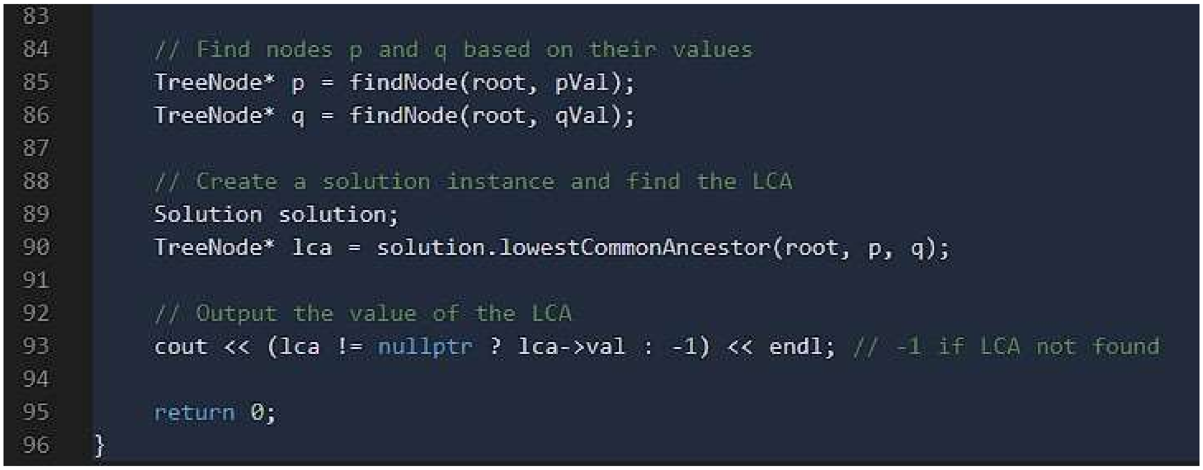
```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // Start from the root of the tree
        while (root) {
            // If both p and q are greater than root, then LCA lies in right
            if (p->val > root->val && q->val > root->val) {
                root = root->right;
            }
            // If both p and q are less than root, then LCA lies in left
            else if (p->val < root->val && q->val < root->val) {
                root = root->left;
            } else {
                // We have found the split point, i.e., the LCA
                return root;
            }
        }
        return nullptr;
    }
};

```



```cpp
TreeNode* insertBST(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }
    if (val < root->val) {
        root->left = insertBST(root->left, val);
    } else {
        root->right = insertBST(root->right, val);
    }
    return root;
}

TreeNode* findNode(TreeNode* root, int val) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->val == val) {
        return root;
    }
    if (val < root->val) {
        return findNode(root->left, val);
    } else {
        return findNode(root->right, val);
    }
}

int main() {
    // Read input values
    int n;
    cin >> n;
    cin.ignore(); // To ignore the newline after the integer in
    string nodeLine;
    getline(cin, nodeLine);
    int pVal, qVal;
    cin >> pVal >> qVal;
```

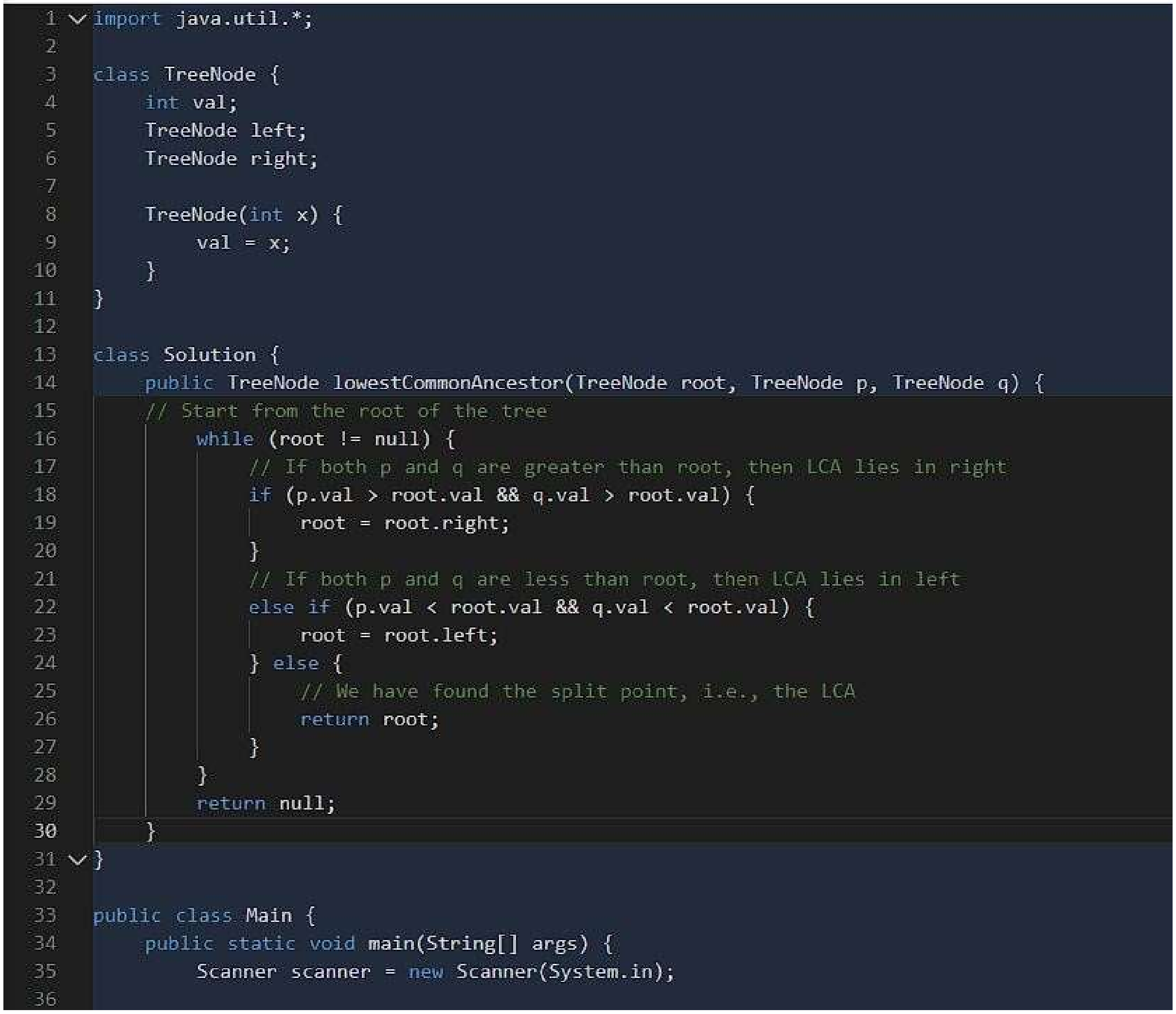


```cpp

    // Build the BST
    TreeNode* root = nullptr;
    stringstream ss(nodeLine);
    string token;
    while (getline(ss, token, ',')) {
        if (token != "null") {
            int val = stoi(token);
            root = insertBST(root, val);
        }
    }
```

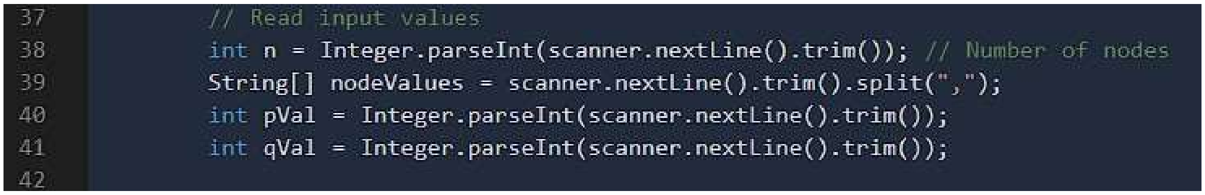


```cpp

    // Find nodes p and q based on their values
    TreeNode* p = findNode(root, pVal);
    TreeNode* q = findNode(root, qVal);

    // Create a solution instance and find the LCA
    Solution solution;
    TreeNode* lca = solution.lowestCommonAncestor(root, p, q);

    // Output the value of the LCA
    cout << (lca != nullptr ? lca->val : -1) << endl; // -1 if LCA not found

    return 0;
}
```

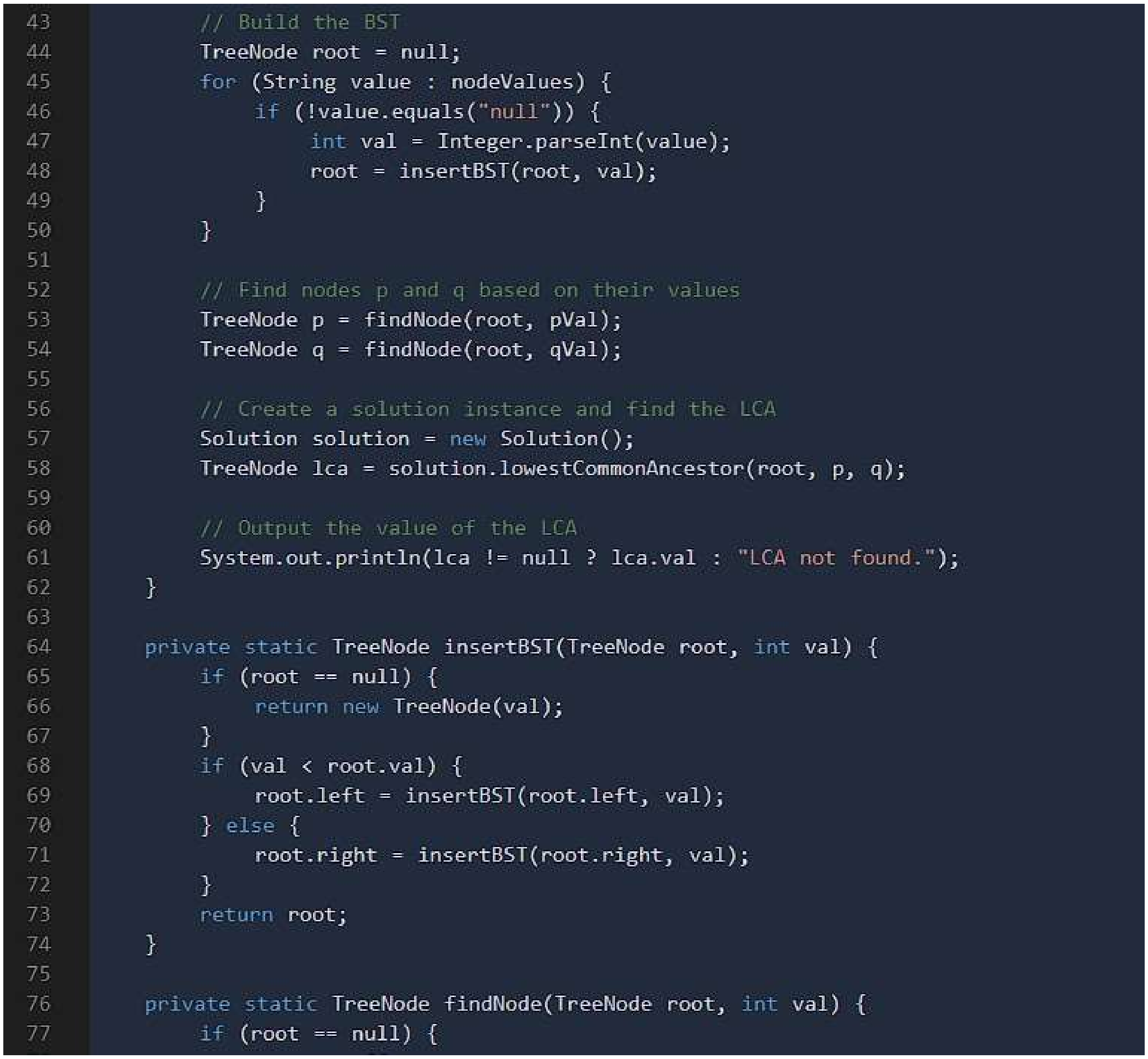
Java :



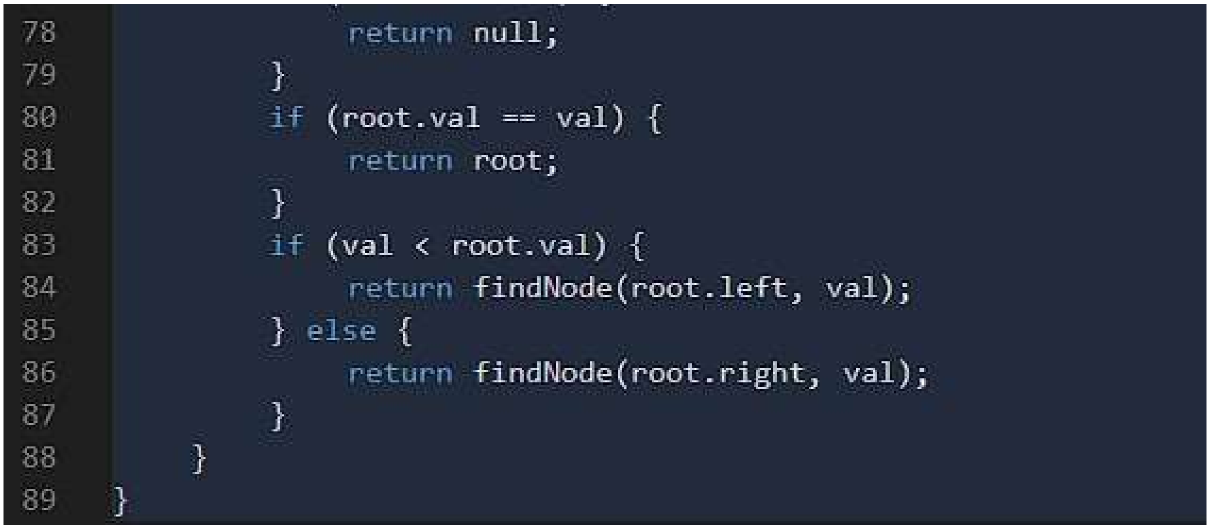
```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // Start from the root of the tree
        while (root != null) {
            // If both p and q are greater than root, then LCA lies in right
            if (p.val > root.val && q.val > root.val) {
                root = root.right;
            }
            // If both p and q are less than root, then LCA lies in left
            else if (p.val < root.val && q.val < root.val) {
                root = root.left;
            } else {
                // We have found the split point, i.e., the LCA
                return root;
            }
        }
        return null;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

```



```java
        // Read input values
        int n = Integer.parseInt(scanner.nextLine().trim()); // Number of nodes
        String[] nodeValues = scanner.nextLine().trim().split(",");
        int pVal = Integer.parseInt(scanner.nextLine().trim());
        int qVal = Integer.parseInt(scanner.nextLine().trim());

```

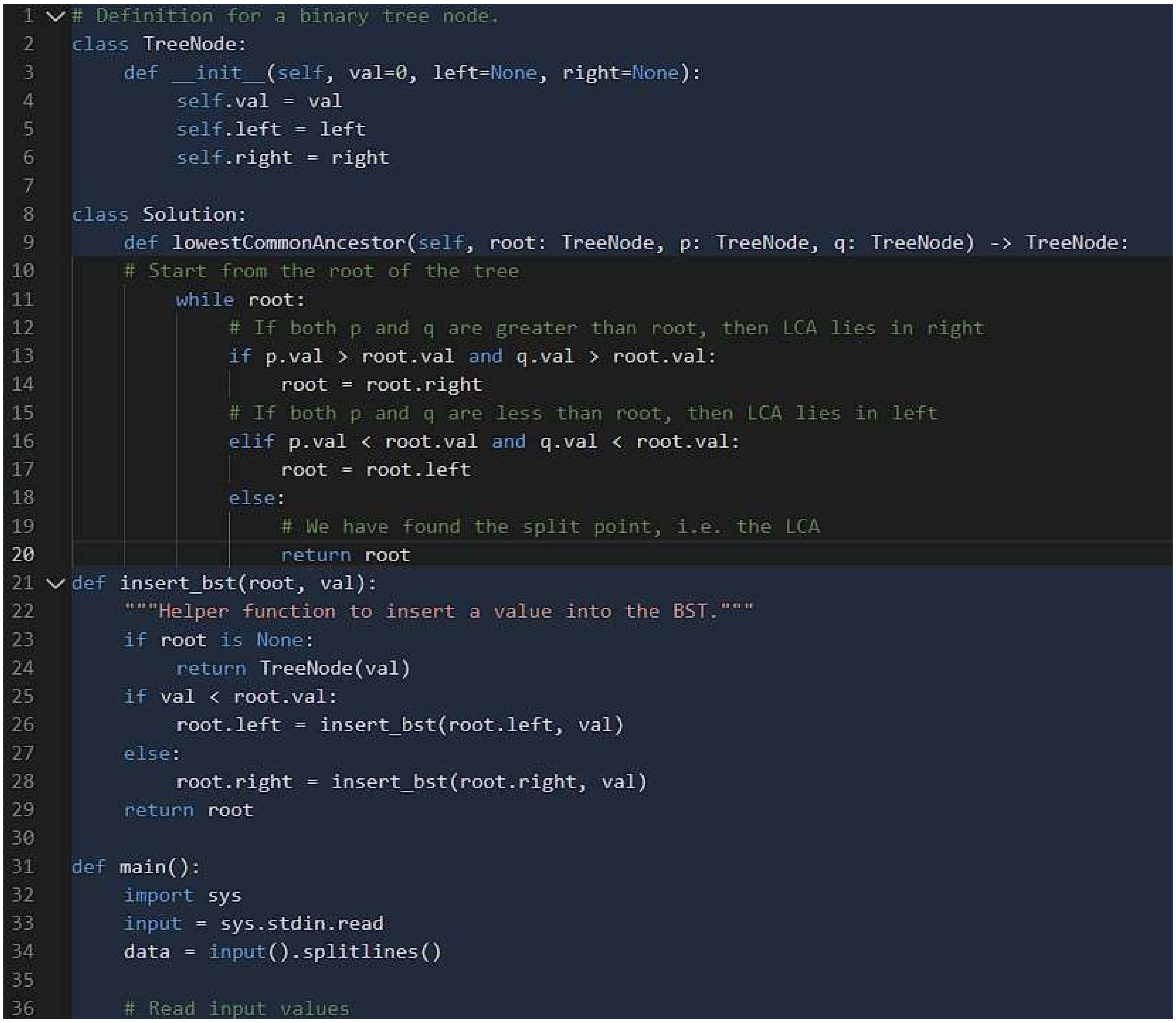


```java
        // Build the BST
        TreeNode root = null;
        for (String value : nodeValues) {
            if (!value.equals("null")) {
                int val = Integer.parseInt(value);
                root = insertBST(root, val);
            }
        }

        // Find nodes p and q based on their values
        TreeNode p = findNode(root, pVal);
        TreeNode q = findNode(root, qVal);

        // Create a solution instance and find the LCA
        Solution solution = new Solution();
        TreeNode lca = solution.lowestCommonAncestor(root, p, q);

        // Output the value of the LCA
        System.out.println(lca != null ? lca.val : "LCA not found.");
    }

    private static TreeNode insertBST(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }
        if (val < root.val) {
            root.left = insertBST(root.left, val);
        } else {
            root.right = insertBST(root.right, val);
        }
        return root;
    }

    private static TreeNode findNode(TreeNode root, int val) {
        if (root == null) {
```

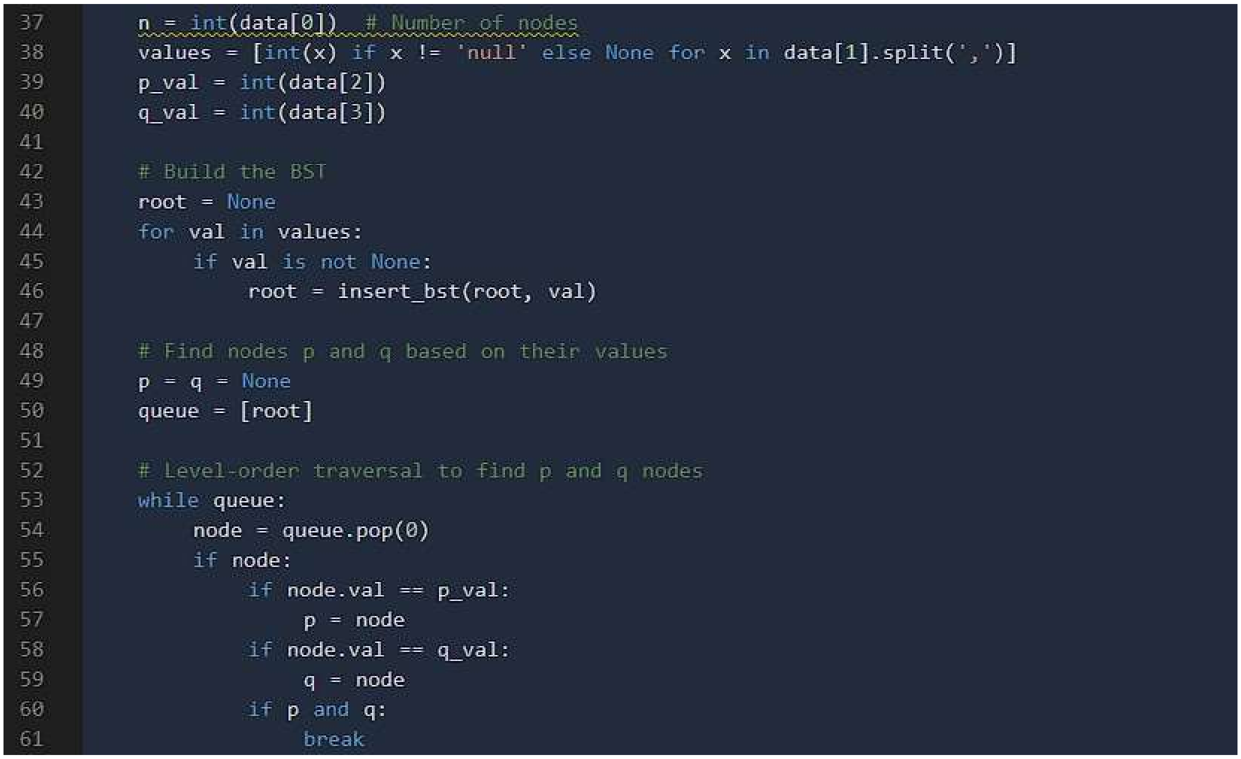


```java
            return null;
        }
        if (root.val == val) {
            return root;
        }
        if (val < root.val) {
            return findNode(root.left, val);
        } else {
            return findNode(root.right, val);
        }
    }
}
```

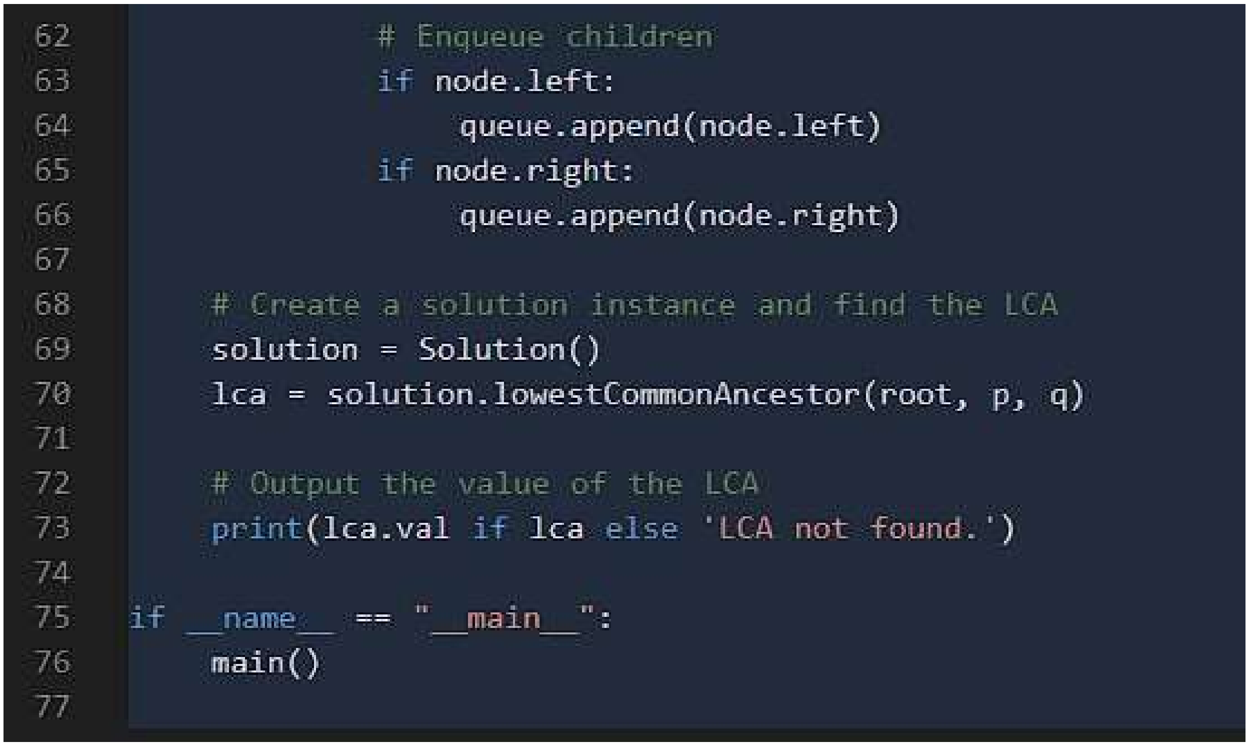
Python :



```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
    # Start from the root of the tree
        while root:
            # If both p and q are greater than root, then LCA lies in right
            if p.val > root.val and q.val > root.val:
                root = root.right
            # If both p and q are less than root, then LCA lies in left
            elif p.val < root.val and q.val < root.val:
                root = root.left
            else:
                # We have found the split point, i.e. the LCA
                return root
def insert_bst(root, val):
    """Helper function to insert a value into the BST."""
    if root is None:
        return TreeNode(val)
    if val < root.val:
        root.left = insert_bst(root.left, val)
    else:
        root.right = insert_bst(root.right, val)
    return root

def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()

    # Read input values
```



```python
    n = int(data[0])  # Number of nodes
    values = [int(x) if x != 'null' else None for x in data[1].split(',')]
    p_val = int(data[2])
    q_val = int(data[3])

    # Build the BST
    root = None
    for val in values:
        if val is not None:
            root = insert_bst(root, val)

    # Find nodes p and q based on their values
    p = q = None
    queue = [root]

    # Level-order traversal to find p and q nodes
    while queue:
        node = queue.pop(0)
        if node:
            if node.val == p_val:
                p = node
            if node.val == q_val:
                q = node
            if p and q:
                break
```



```
            # Enqueue children
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    # Create a solution instance and find the LCA
    solution = Solution()
    lca = solution.lowestCommonAncestor(root, p, q)

    # Output the value of the LCA
    print(lca.val if lca else 'LCA not found.')

if __name__ == "__main__":
    main()

```

Problem 3 : Minimum Number of Operations to Move All Balls to Each Box

Approach :

The goal is to find the minimum number of moves to gather all balls in each box. To solve this efficiently, we break the problem into two parts:

- Imagine moving all balls from the left side towards each box. As you pass each box, accumulate the number of moves needed to bring balls from earlier boxes.
- For example, if a ball is 5 steps away, it takes 5 operations to move it to the current box.
- Do the same from the right, moving all balls from the right side towards each box.
- For each box, the total moves are the sum of moves from the left and the right passes. This gives the minimum operations to gather all balls at that box.

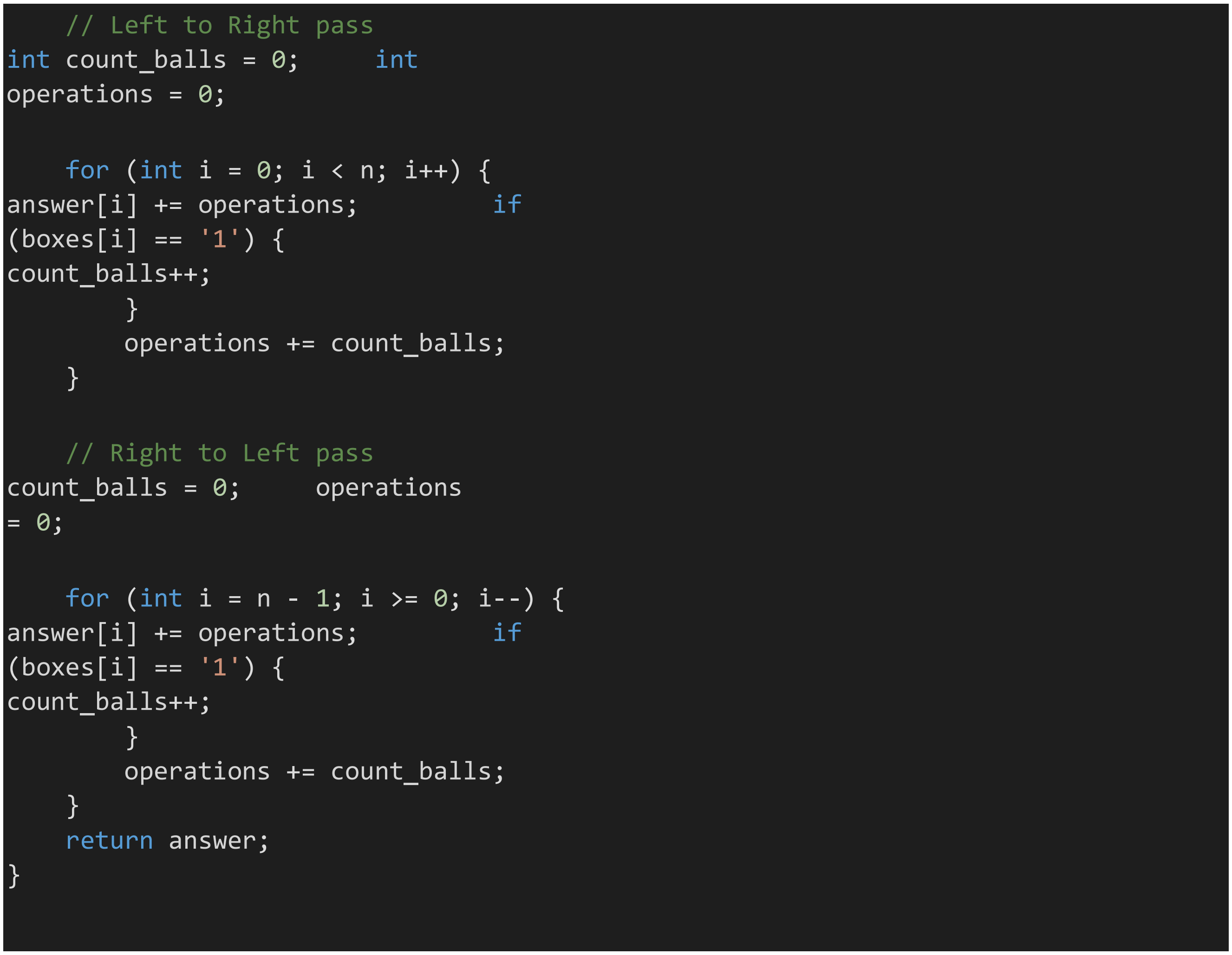
Reference Video : Click Here

C :

```
int* minOperations(char* boxes, int* returnSize) {
    //Write your code here     int n =
strlen(boxes);     int* answer = (int*)malloc(n
* sizeof(int));

    // Initialize the answer array to 0
for (int i = 0; i < n; i++) {
answer[i] = 0;
    }

    // Left to Right pass
int count_balls = 0;     int
operations = 0;

    for (int i = 0; i < n; i++) {
answer[i] += operations;         if
(boxes[i] == '1') {
count_balls++;
        }
        operations += count_balls;
    }

    // Right to Left pass
count_balls = 0;     operations
= 0;

    for (int i = n - 1; i >= 0; i--) {
answer[i] += operations;         if
(boxes[i] == '1') {
count_balls++;
        }
        operations += count_balls;
    }

    *returnSize = n;
return answer;
}
```

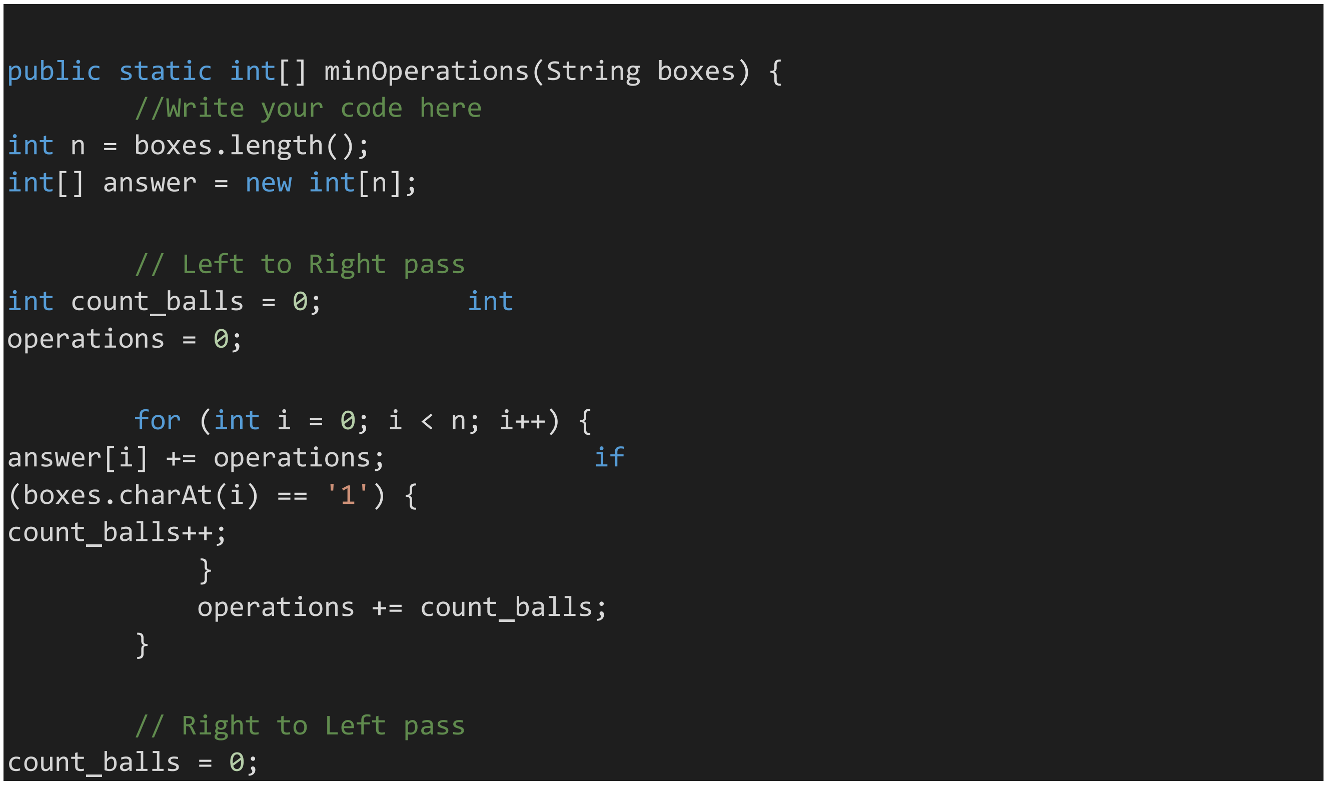
C++ :

```
vector<int> minOperations(string boxes) {
    //Write your code here
int n = boxes.size();
vector<int> answer(n, 0);
```



```
    // Left to Right pass
int count_balls = 0;         int
operations = 0;

    for (int i = 0; i < n; i++) {
answer[i] += operations;         if
(boxes[i] == '1') {
count_balls++;
        }
        operations += count_balls;
    }

    // Right to Left pass
count_balls = 0;     operations
= 0;

    for (int i = n - 1; i >= 0; i--) {
answer[i] += operations;         if
(boxes[i] == '1') {
count_balls++;
        }
        operations += count_balls;
    }
    return answer;
}
```

Java :



```
public static int[] minOperations(String boxes) {
        //Write your code here
int n = boxes.length();
int[] answer = new int[n];

        // Left to Right pass
int count_balls = 0;             int
operations = 0;

        for (int i = 0; i < n; i++) {
answer[i] += operations;             if
(boxes.charAt(i) == '1') {
count_balls++;
            }
            operations += count_balls;
        }

        // Right to Left pass
count_balls = 0;
```

```
        operations = 0;

        for (int i = n - 1; i >= 0; i--) {
answer[i] += operations;             if
(boxes.charAt(i) == '1') {
count_balls++;
            }
            operations += count_balls;
        }
        return answer;
    }
```

Python :

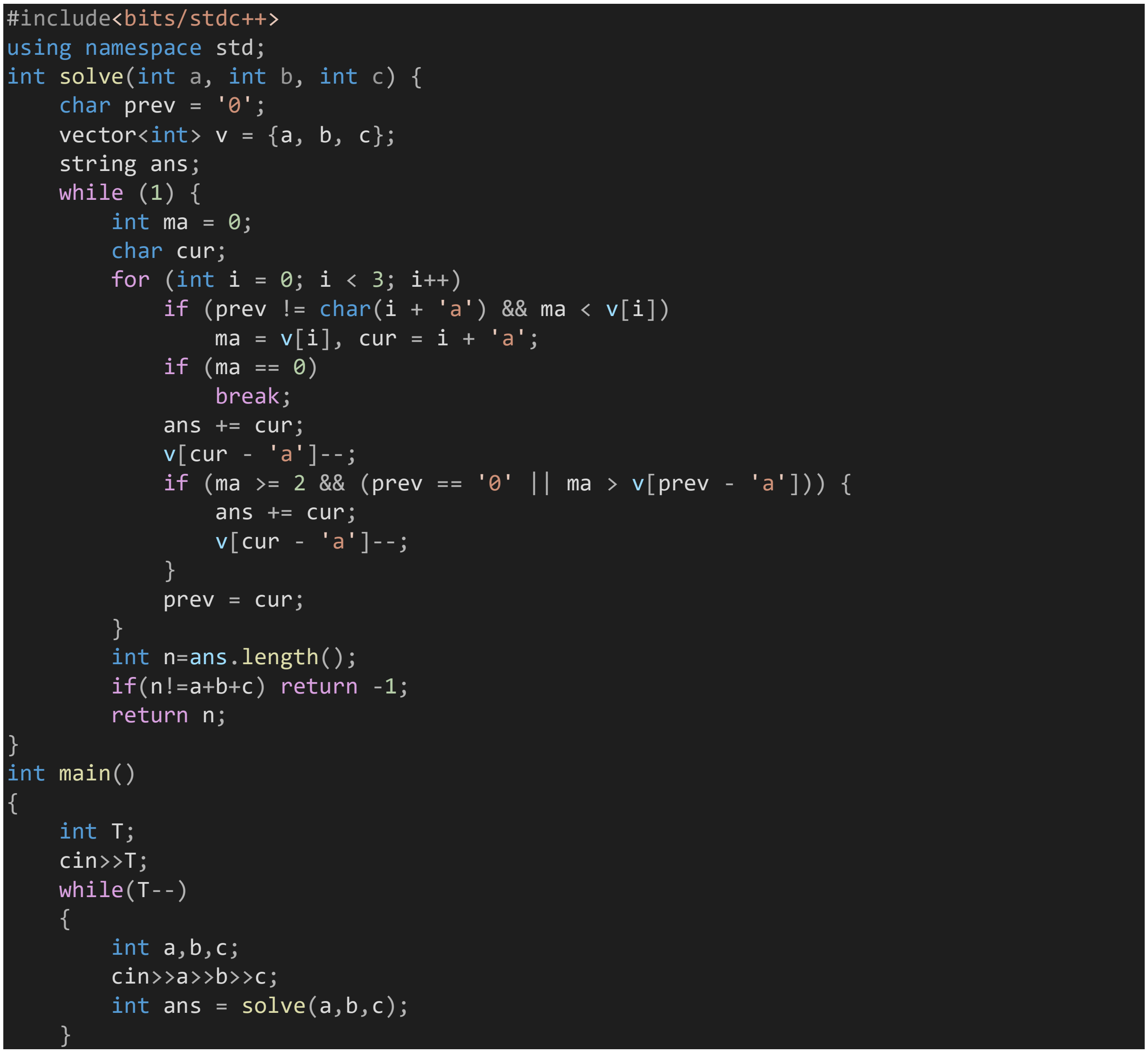
```
 def
minOperations(boxes):
#Write your code here
n = len(boxes)     answer
= [0] * n

    # Left to Right pass
count_balls = 0     operations
= 0
     for i in
range(n):
        answer[i] += operations
if boxes[i] == '1':
count_balls += 1
operations += count_balls

    # Right to Left pass
count_balls = 0     operations
= 0
     for i in range(n - 1, -1, -
1):         answer[i] +=
operations         if boxes[i] ==
'1':              count_balls += 1
operations += count_balls

    return answer
```

4. Problem link : Constraint string construction

Approach:

1. **Track character counts**: For each character ('a', 'b', 'c'), count how many instances we can place in the result string.
2. **Build the string iteratively**: At each step, choose the character that has the highest count but is different from the previously added character (to avoid three consecutive identical characters).
3. **Handle tie-breaking**: If we can safely add more than one of the same character, we do so, but only when it's safe (i.e., it doesn't form three consecutive characters).
4. **Check validity**: If we can successfully build the string that respects the constraints, return its length; otherwise, return -1.

C++ Code



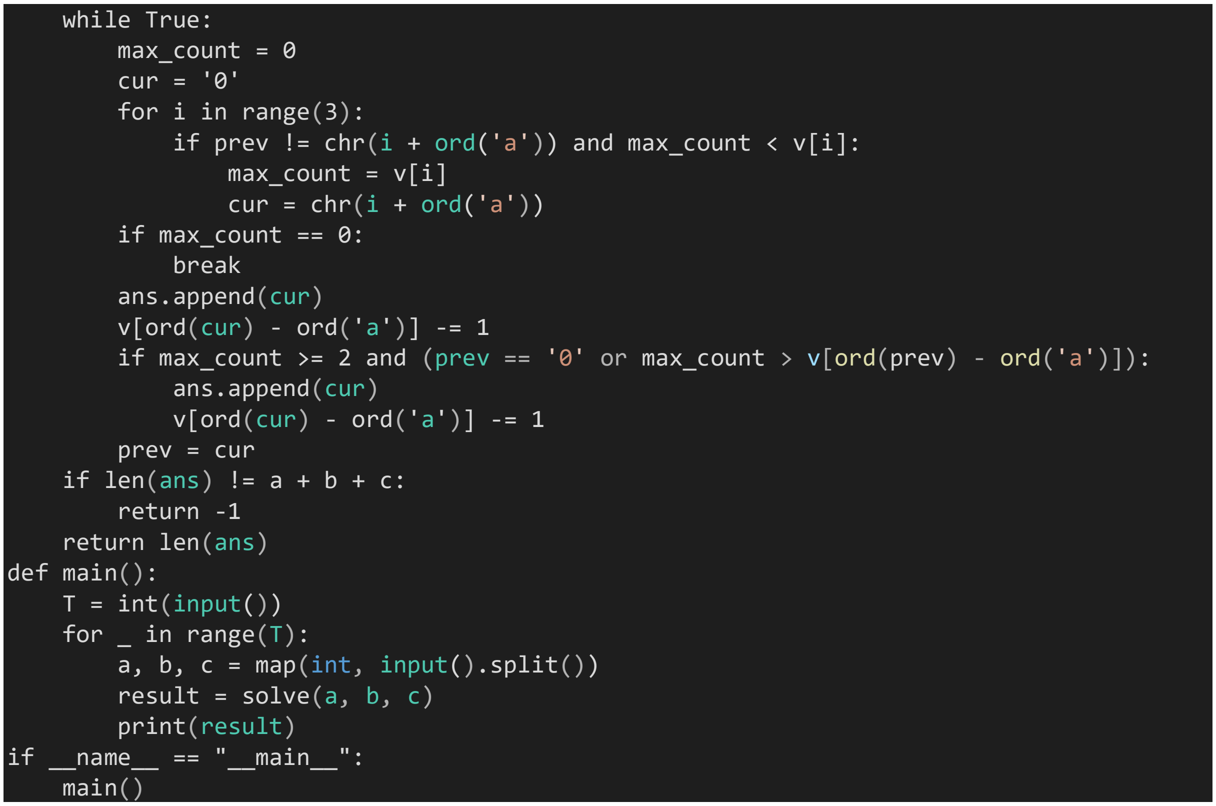
```cpp
#include<bits/stdc++>
using namespace std;
int solve(int a, int b, int c) {
    char prev = '0';
    vector<int> v = {a, b, c};
    string ans;
    while (1) {
        int ma = 0;
        char cur;
        for (int i = 0; i < 3; i++)
            if (prev != char(i + 'a') && ma < v[i])
                ma = v[i], cur = i + 'a';
            if (ma == 0)
                break;
            ans += cur;
            v[cur - 'a']--;
            if (ma >= 2 && (prev == '0' || ma > v[prev - 'a'])) {
                ans += cur;
                v[cur - 'a']--;
            }
            prev = cur;
        }
        int n=ans.length();
        if(n!=a+b+c) return -1;
        return n;
}
int main()
{
    int T;
    cin>>T;
    while(T--)
    {
        int a,b,c;
        cin>>a>>b>>c;
        int ans = solve(a,b,c);
    }
    return 0;
}
```

Java Code

```java
import java.util.*;
public class Main {
    public static int solve(int a, int b, int c) {
        char prev = '0';
        int[] v = {a, b, c};
        StringBuilder ans = new StringBuilder();
        while (true) {
            int maxCount = 0;
            char cur = '0';
            for (int i = 0; i < 3; i++) {
                if (prev != (char) (i + 'a') && maxCount < v[i]) {
                    maxCount = v[i];
                    cur = (char) (i + 'a');
                }
            }
            if (maxCount == 0) break;
            ans.append(cur);
            v[cur - 'a']--;

            if (maxCount >= 2 && (prev == '0' || maxCount > v[prev - 'a'])) {
                ans.append(cur);
                v[cur - 'a']--;
            }
            prev = cur;
        }
        if (ans.length() != a + b + c) return -1;
        return ans.length();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();
        while (T-- > 0) {
            int a = sc.nextInt();
            int b = sc.nextInt();
            int c = sc.nextInt();
            int result = solve(a, b, c);
            System.out.println(result);
        }
        sc.close();
    }
}
```

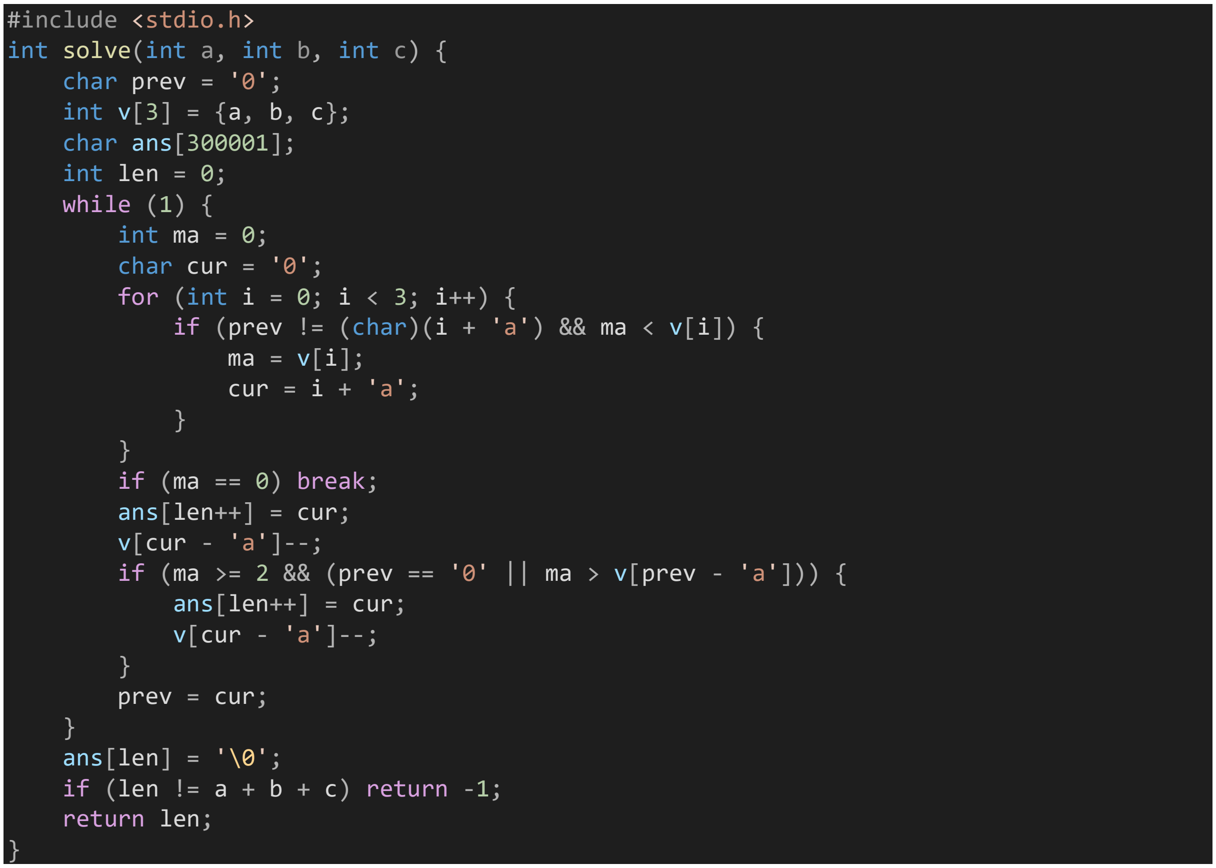
Python Code

```python
def solve(a, b, c):
    prev = '0'
    v = [a, b, c]
    ans = []
```



```python
    while True:
        max_count = 0
        cur = '0'
        for i in range(3):
            if prev != chr(i + ord('a')) and max_count < v[i]:
                max_count = v[i]
                cur = chr(i + ord('a'))
        if max_count == 0:
            break
        ans.append(cur)
        v[ord(cur) - ord('a')] -= 1
        if max_count >= 2 and (prev == '0' or max_count > v[ord(prev) - ord('a')]):
            ans.append(cur)
            v[ord(cur) - ord('a')] -= 1
        prev = cur
    if len(ans) != a + b + c:
        return -1
    return len(ans)
def main():
    T = int(input())
    for _ in range(T):
        a, b, c = map(int, input().split())
        result = solve(a, b, c)
        print(result)
if __name__ == "__main__":
    main()
```

C Code



```c
#include <stdio.h>
int solve(int a, int b, int c) {
    char prev = '0';
    int v[3] = {a, b, c};
    char ans[300001];
    int len = 0;
    while (1) {
        int ma = 0;
        char cur = '0';
        for (int i = 0; i < 3; i++) {
            if (prev != (char)(i + 'a') && ma < v[i]) {
                ma = v[i];
                cur = i + 'a';
            }
        }
        if (ma == 0) break;
        ans[len++] = cur;
        v[cur - 'a']--;
        if (ma >= 2 && (prev == '0' || ma > v[prev - 'a'])) {
            ans[len++] = cur;
            v[cur - 'a']--;
        }
        prev = cur;
    }
    ans[len] = '\0';
    if (len != a + b + c) return -1;
    return len;
}
```