# DSA Practice Question Set - 7

## 1)Next Permutation :

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be in place and use only constant extra memory.

**Example 1:**

Input: nums = [1,2,3]
Output: [1,3,2]

**Example 2:**

Input: nums = [3,2,1]
Output: [1,2,3]

**Example 3:**

Input: nums = [1,1,5]

Output: [1,5,1]

**Constraints:**

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 100

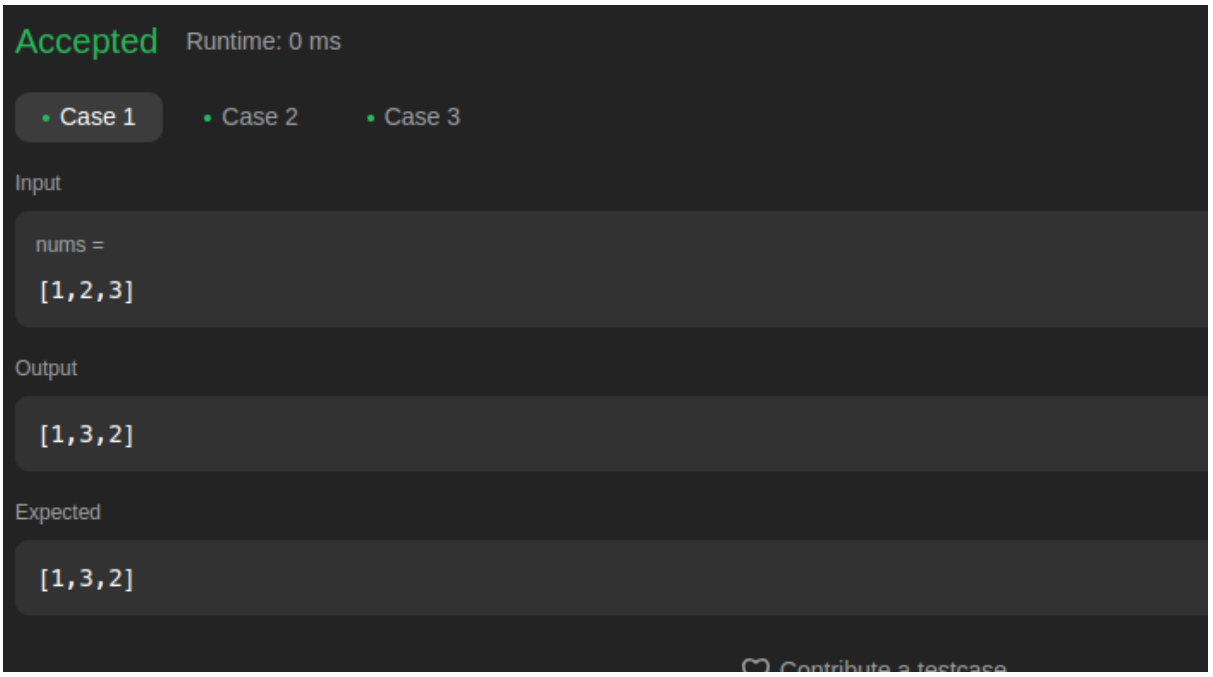
## Program :

```
import java.util.Arrays;

class Solution {

    private boolean isStrictlyIncreasing(int[] arr) {

        for (int i = 0; i < arr.length - 1; i++) {

            if (arr[i] >= arr[i + 1]) {

                return false;

            }

        }

        return true;

    }

    private boolean isStrictlyDecreasing(int[] arr) {

        for (int i = 0; i < arr.length - 1; i++) {

            if (arr[i] <= arr[i + 1]) {

                return false;

            }
```

```
    }
    return true;
}
public void nextPermutation(int[] nums) {
    int n = nums.length;
    if (isStrictlyDecreasing(nums)) {
        Arrays.sort(nums);
        return;
    }
    int i = n - 2;
    while (i >= 0 && nums[i] >= nums[i + 1]) {
        i--;
    }
    if (i >= 0) {
        int j = n - 1;
        while (nums[j] <= nums[i]) {
            j--;
        }
        int temp = nums[i];
```

```
                nums[i] = nums[j];
                nums[j] = temp;
            }
            reverse(nums, i + 1, n - 1);
        }
        private void reverse(int[] nums, int start, int end) {
            while (start < end) {
                int temp = nums[start];
                nums[start] = nums[end];
                nums[end] = temp;
                start++;
                end--;
            }
        }
    }
```

**Output :**



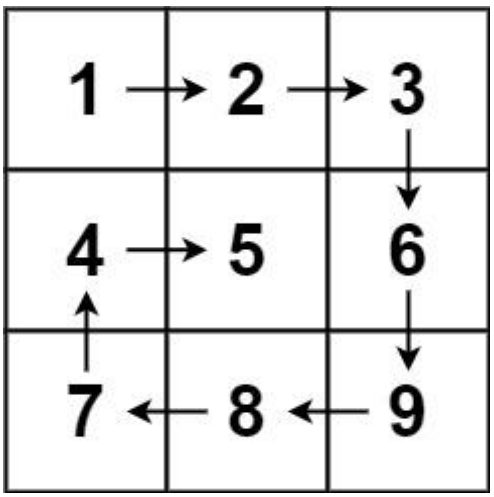

**Time Complexity : O(n)**

**Space Complexity : O(1)**

## 2)Spiral Matrix

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

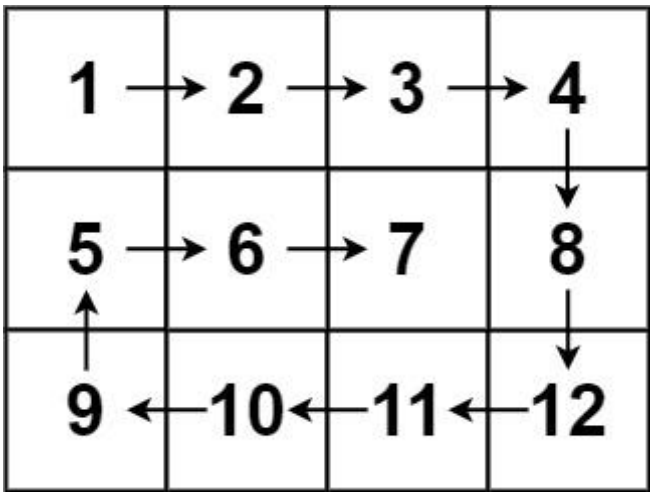
**Example 1:**



**Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]**

**Output: [1,2,3,6,9,8,7,4,5]**

**Example 2:**




**Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]**

**Output: [1,2,3,4,8,12,11,10,9,5,6,7]**

**Program :**

```
class Solution {
   public List<Integer> spiralOrder(int[][] matrix) {
       List<Integer> res = new ArrayList<>();

       int rowst = 0;
       int rowen = matrix.length-1;
       int colst = 0;
       int colen  = matrix[0].length-1;

       while(rowst<=rowen && colst<=colen){
           for(int i = colst;i<=colen;i++){
               res.add(matrix[rowst][i]);
           }
           rowst++;

           for(int i = rowst;i<=rowen;i++){
               res.add(matrix[i][colen]);
           }
```

```
            colen--;

            if(rowst<=rowen){
                for(int i = colen;i>=colst;i--){
                    res.add(matrix[rowen][i]);
                }
                rowen--;
            }

            if(colst<=colen){
                for(int i = rowen;i>=rowst;i--){
                    res.add(matrix[i][colst]);
                }
                   colst++;
            }


        }

        return res;
    }
}
```

**Output :**

**Time Complexity : O(m*n)**

**Space Complexity : O(1)**

**3)Given a string s, find the length of the longest Substring without repeating characters.**

**Example 1:**

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

**Example 2:**

Input: s = "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

**Example 3:**

Input: s = "pwwkew"

Output: 3

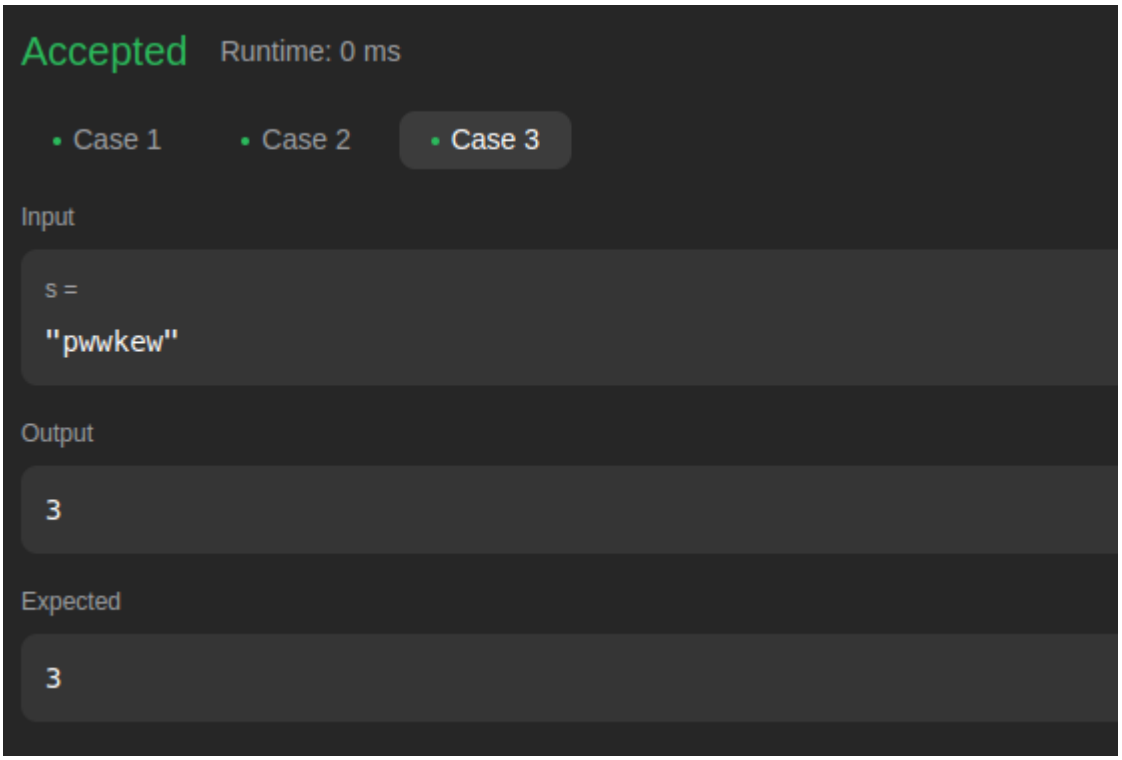
Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

## Program :

```
class Solution {
   public int lengthOfLongestSubstring(String s) {
       int a = 0;
       int b = 0;
       int max = 0;

       HashSet<Character> hash_set = new HashSet<>();

       while(b<s.length()){
           if(!hash_set.contains(s.charAt(b))){
               hash_set.add(s.charAt(b));
               b++;
               max = Math.max(hash_set.size(),max);
           }
           else{
               hash_set.remove(s.charAt(a));
               a++;
           }

       }
       return max;

   }
}
```

## Output :



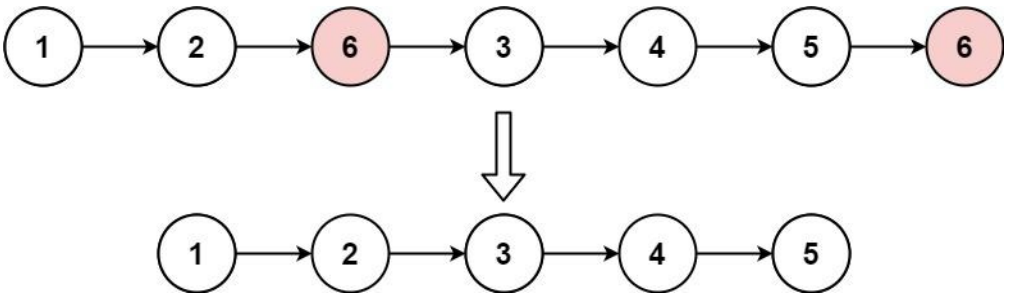

## Time Complexity : O(n)

## Space Complexity : O(1)

## 4)Remove linked list elements

**Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.**

**Example 1:**




**Input: head = [1,2,6,3,4,5,6], val = 6**

**Output: [1,2,3,4,5]**

**Example 2:**

**Input: head = [], val = 1**
**Output: []**

**Example 3:**

**Input: head = [7,7,7,7], val = 7**
**Output: []**

**Constraints:**

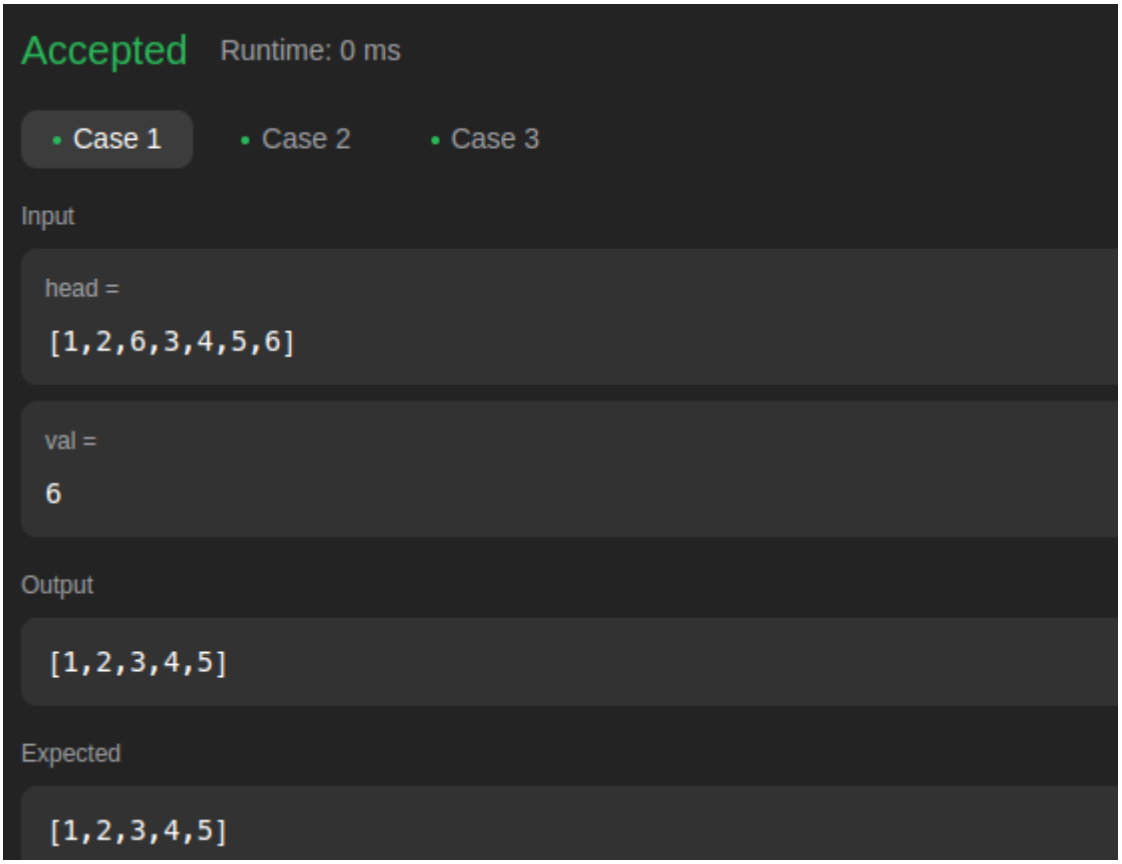
- **The number of nodes in the list is in the range [0, $10^4$].**
- **1 <= Node.val <= 50**
- **0 <= val <= 50**

# Program :

```
/**
* Definition for singly-linked list.
* public class ListNode {
*     int val;
*     ListNode next;
*     ListNode() {}
*     ListNode(int val) { this.val = val; }
*     ListNode(int val, ListNode next) { this.val = val; this.next = next;
}
* }
*/
class Solution {
   public ListNode removeElements(ListNode head, int val) {
       ListNode ans =  new ListNode(0,head);
       ListNode dummy = ans;

       while(dummy!=null){
           while(dummy.next!=null && dummy.next.val == val){
               dummy.next = dummy.next.next;
           }
           dummy = dummy.next;
       }
       return ans.next;
   }
```

```
}
```

## Output :




## Time Complexity : O(n)

## Space Complexity : O(1)

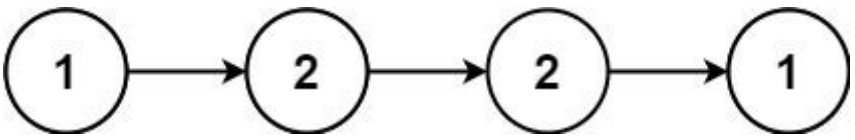
## 5)Palindrome Linked List :

**Given the head of a singly linked list, return true *if it is a***

***palindrome***

***or* false *otherwise*.**

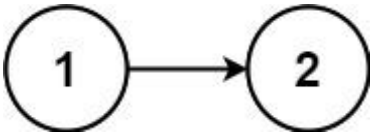
**Example 1:**




**Input: head = [1,2,2,1]**

**Output: true**

**Example 2:**




**Input: head = [1,2]**

**Output: false**

**Constraints:**

- **The number of nodes in the list is in the range [1, $10^5$].**
- **0 <= Node.val <= 9**

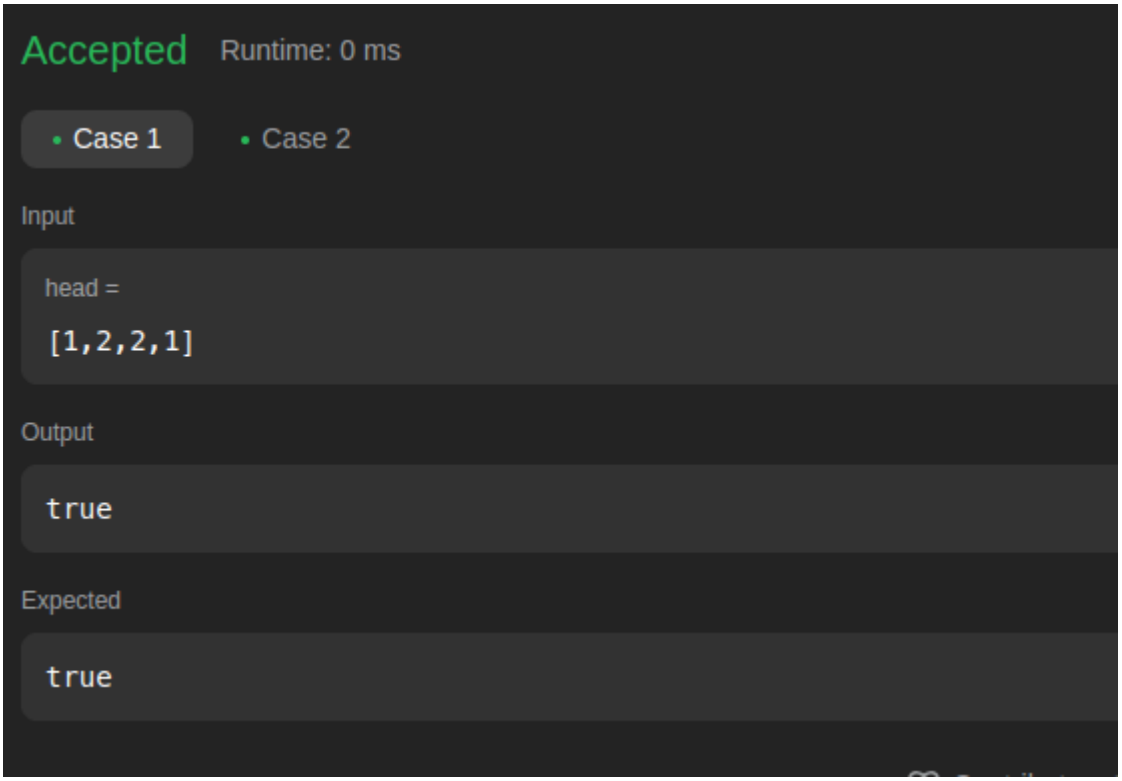
**Follow up: Could you do it in O(n) time and O(1) space?**

## Program :

```
/**
* Definition for singly-linked list.
* public class ListNode {
*     int val;
*     ListNode next;
*     ListNode() {}
*     ListNode(int val) { this.val = val; }
*     ListNode(int val, ListNode next) { this.val = val; this.next = next;
}
* }
*/
class Solution {
   ListNode curr;
   public boolean isPalindrome(ListNode head) {
        curr = head;
        return solve(head);

   }
   public boolean solve(ListNode head){
        if(head == null) return true;
```

```
        boolean ans = solve(head.next) && head.val == curr.val;
        curr = curr.next;
        return ans;
    }
}
```

**Output :**




**Time Complexity : O(n)**

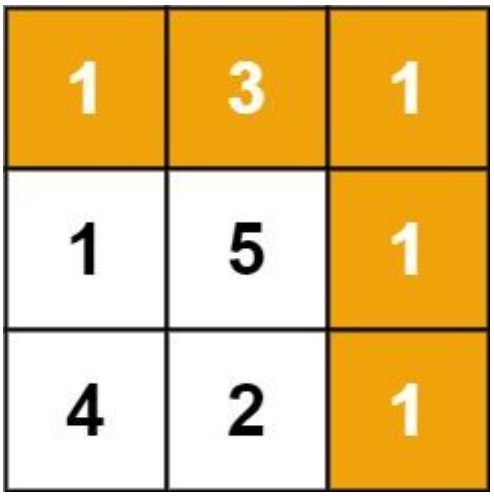
**Space Complexity : O(1)**

**6)Minimum path sum:**

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

**Example 1:**




Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

**Example 2:**

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

**Constraints:**

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 200
- 0 <= grid[i][j] <= 200

## Program :

```
class Solution {

   public int minPathSum(int[][] grid) {

       int m = grid.length, n = grid[0].length;

       for (int j = 1; j < n; j++) {

           grid[0][j] += grid[0][j - 1];
```

```
        }

        for (int i = 1; i < m; i++) {

            grid[i][0] += grid[i - 1][0];

        }

        for (int i = 1; i < m; i++) {

            for (int j = 1; j < n; j++) {

                grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);

            }

        }

        return grid[m - 1][n - 1];

    }

}
```

**Output :**

## Time Complexity : O(m*n)

## Space Complexity : O(1)

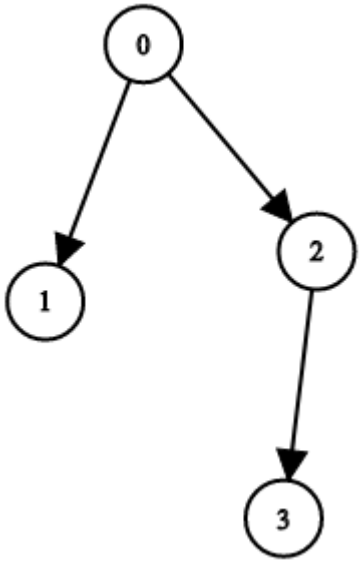
## 7)Validate Binary Tree Nodes

**You have n binary tree nodes numbered from 0 to n - 1 where node i has two children leftChild[i] and rightChild[i], return true if and only if all the given nodes form exactly one valid binary tree.**

**If node i has no left child then leftChild[i] will equal -1, similarly for the right child.**

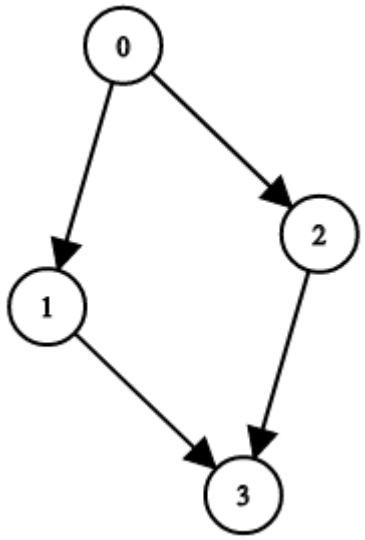
**Note that the nodes have no values and that we only use the node numbers in this problem.**

**Example 1:**




**Input: n = 4, leftChild = [1,-1,3,-1], rightChild = [2,-1,-1,-1]**
**Output: true**

**Example 2:**




**Input: n = 4, leftChild = [1,-1,3,-1], rightChild = [2,3,-1,-1]**
**Output: false**

**Example 3:**


**Input: n = 2, leftChild = [1,0], rightChild = [-1,-1]**
**Output: false**

**Constraints:**

- **n == leftChild.length == rightChild.length**
- **1 <= n <= $10^4$**
- **-1 <= leftChild[i], rightChild[i] <= n - 1**

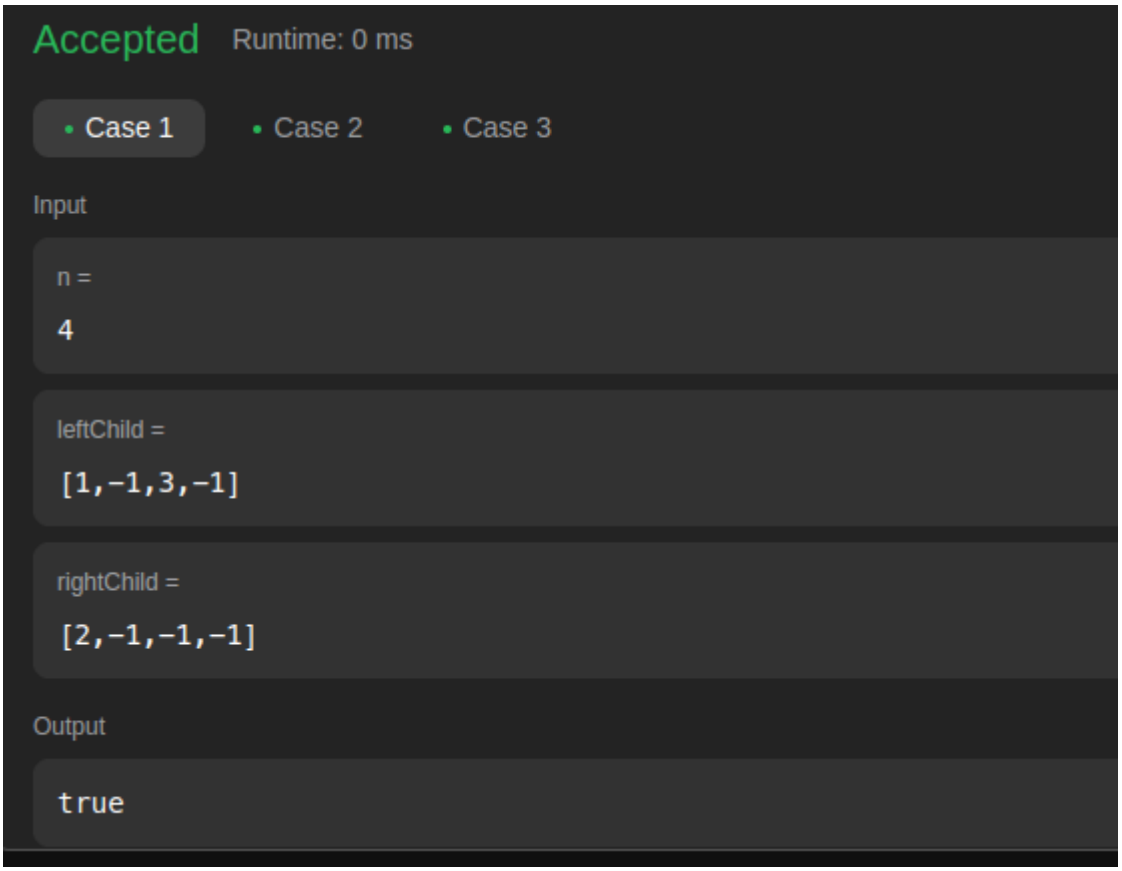
## PROGRAM :

```
class Solution {
```

```java
  public boolean validateBinaryTreeNodes(int n, int[] leftChild, int[]
rightChild) {

   int[] inDegree = new int[n];

   int root = -1;



   for (final int child : leftChild)

     if (child != -1 && ++inDegree[child] == 2)

       return false;



   for (final int child : rightChild)

     if (child != -1 && ++inDegree[child] == 2)

       return false;



   for (int i = 0; i < n; ++i)

     if (inDegree[i] == 0)

       if (root == -1)

         root = i;

       else

         return false;



   if (root == -1)

     return false;



   return countNodes(root, leftChild, rightChild) == n;

 }
```

```java
private int countNodes(int root, int[] leftChild, int[] rightChild) {

  if (root == -1)

    return 0;

  return 1 + //

      countNodes(leftChild[root], leftChild, rightChild) +

      countNodes(rightChild[root], leftChild, rightChild);

 }

}
```

**Output :**




**Time Complexity : O(n)**

**Space Complexity : O(n)**

## 8)Word Ladder :

**A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> $s_1$ -> $s_2$ -> ... -> $s_k$ such that:**

- **Every adjacent pair of words differs by a single letter.**
- **Every $s_i$ for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.**
- **$s_k$ == endWord**

**Given two words, beginWord and endWord, and a dictionary wordList, return *the number of words in the shortest transformation sequence from* beginWord *to* endWord*, or* 0 *if no such sequence exists.***

**Example 1:**

**Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]**

**Output: 5**

**Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.**

**Example 2:**

**Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]**

**Output: 0**

**Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.**

**Constraints:**

- **1 <= beginWord.length <= 10**
- **endWord.length == beginWord.length**
- **1 <= wordList.length <= 5000**
- **wordList[i].length == beginWord.length**

- **beginWord, endWord, and wordList[i] consist of lowercase English letters.**
- **beginWord != endWord**
- **All the words in wordList are unique.**

## Program :

```
class Solution {



   class TrieNode2 {

       TrieNode2[] children;

       boolean isWord;

       String word;

       public TrieNode2() {

           children = new TrieNode2[26];

           isWord = false;

           word = null;

       }

   }



   public int ladderLength(String beginWord, String endWord, List<String>
wordList) {

       TrieNode2 root = new TrieNode2();

       boolean endWordExist = false;

       for(int i=0; i<wordList.size(); i++) {

           if(wordList.get(i).equalsIgnoreCase(endWord)) {

               endWordExist = true;

           }

           addWord(root,wordList.get(i));
```

```
        }


        if(!endWordExist) {
            return 0;
        }


        Queue<String> next = new LinkedList<>();
        next.add(beginWord);
        String nextWord;
        int count = 0, n;
        while(!next.isEmpty() && endWordExist) {
            n = next.size();
            count += 1;
            while(n-- > 0) {
                nextWord = next.poll();
                if (nextWord.equals(endWord)) {
                    endWordExist = false;
                    break;
                }
                addNextWord(root, next, nextWord);
            }
        }


        if(endWordExist) {
            return 0;
        }
```

```
        return count;

    }



    public void addWord(TrieNode2 root, String word) { // Add word to trie

        TrieNode2 temp;

        for(int i =0; i<word.length(); i++) {

            int index = word.charAt(i)-'a';

            if(root.children[index] == null) {

                temp = new TrieNode2();

                root.children[index] = temp;

            }

            root = root.children[index];

        }

        root.word = word;

        root.isWord = true;

    }



    public void addNextWord(TrieNode2 root, Queue<String> next, String word)
{ // search next adjeacent word

        for(int i =0; i<word.length(); i++) {

            addNextWord(root, next, word, 0, i);

        }

    }



    public void addNextWord(TrieNode2 root, Queue<String> next, String word,
int index, int changeIndex) { // helper function for addNextWord

        if(index == word.length() ) {
```

```
            if(root.isWord) {

                next.add(root.word);

                root.isWord = false;

            }

            return;

        }



        if(index == changeIndex) {

            for(int i =0; i<26; i++) {

                if(root.children[i] != null && (word.charAt(index)-'a') !=
i) {

                    addNextWord(root.children[i], next, word, index+1,
changeIndex);

                }

            }

            return;

        }



        if(root.children[word.charAt(index) - 'a'] != null) {

            addNextWord(root.children[word.charAt(index) - 'a'], next, word,
index+1, changeIndex);

        }

    }



}
```

**Output :**


## Time Complexity : O(n)

## Space Complexity : O(n)

## 9)Word Ladder II

**A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> $s_1$ -> $s_2$ -> ... -> $s_k$ such that:**

- **Every adjacent pair of words differs by a single letter.**
- **Every $s_i$ for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.**
- **$s_k$ == endWord**

**Given two words, beginWord and endWord, and a dictionary wordList, return *all the shortest transformation sequences from* beginWord *to* endWord*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* [beginWord, $s_1$, $s_2$, ..., $s_k$].**

**Example 1:**

**Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]**

**Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]**

**Explanation: There are 2 shortest transformation sequences:**

**"hit" -> "hot" -> "dot" -> "dog" -> "cog"**

**"hit" -> "hot" -> "lot" -> "log" -> "cog"**

**Example 2:**

**Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]**

**Output: []**

**Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.**

**Constraints:**

- **1 <= beginWord.length <= 5**
- **endWord.length == beginWord.length**
- **1 <= wordList.length <= 500**
- **wordList[i].length == beginWord.length**
- **beginWord, endWord, and wordList[i] consist of lowercase English letters.**
- **beginWord != endWord**
- **All the words in wordList are unique.**
- **The sum of all shortest transformation sequences does not exceed $10^5$**

## Program :

```
class Solution {

   private int L;

   private boolean hasPath;
```

```java
   public List<List<String>> findLadders(String beginWord, String endWord,
List<String> wordList){

       this.L = beginWord.length();

       Set<String> wordSet = new HashSet<>();

       wordSet.addAll(wordList);

       if(!wordSet.contains(endWord)) return new ArrayList<>();



       // BFS

       // build a directed graph G with beginWord being the root

       // we guarantee in G, for all nodes, the dis from beginWord is the
shortest

       Map<String, List<String>> adjList = new HashMap<String,
List<String>>();

       wordSet.remove(beginWord); // beginWord in wordList is useless

       buildAdjList(beginWord, endWord, wordSet, adjList);

       if(this.hasPath==false) return new ArrayList<>();



       // DFS

       // get all paths from beginWord to endWord, knowing that all paths
have the same shortest length

       // implement a cache to save branches that have already been visited

       return backtrack(adjList, beginWord, endWord, new HashMap<>());

   }



   public List<List<String>> backtrack(

       Map<String, List<String>> adjList,

       String currWord,
```

```
        String endWord,
        Map<String, List<List<String>>> cache
    ){
        if(cache.containsKey(currWord)) return cache.get(currWord);
        List<List<String>> result = new ArrayList<>();
        if(currWord.equals(endWord)){
            result.add(new ArrayList<>(Arrays.asList(currWord)));
        }else{
            List<String> neighbors = adjList.getOrDefault(currWord, new
ArrayList<>());
            for(String neighbor: neighbors){
                List<List<String>> paths = backtrack(adjList, neighbor,
endWord, cache);
                for(List<String> path: paths){
                    List<String> copy = new ArrayList<>(path);
                    copy.add(0, currWord);
                    result.add(copy);
                }
            }
        }
        cache.put(currWord, result);
        return result;
    }

    public void buildAdjList(String beginWord, String endWord, Set<String>
unvisitedWords,  Map<String, List<String>> adjList){
        Queue<String> q = new LinkedList<>();
```

```
        q.add(beginWord);



        while(!q.isEmpty()){

            if(this.hasPath) break;

            int size = q.size();

            Set<String> nextLevelWords = new HashSet<>();

            for(int i=0; i<size; i++){

                String currWord = q.poll();

                List<String> nextLevelNeighbors=
getNextLevelNeighbors(currWord, unvisitedWords, adjList);

                // System.out.println(currWord+" neighbors: " +
nextLevelNeighbors);

                for(String nextLevelNeighbor: nextLevelNeighbors){

                    if(!nextLevelWords.contains(nextLevelNeighbor)){

                        if(nextLevelNeighbor.equals(endWord)) this.hasPath =
true;

                        nextLevelWords.add(nextLevelNeighbor);

                        q.add(nextLevelNeighbor);

                    }

                }

            }

            // only after adding all edges to next level

            // can we remove next level nodes

            for(String w: nextLevelWords){

                unvisitedWords.remove(w);

            }

        }
```

```
    }


    public List<String> getNextLevelNeighbors(String word, Set<String>
unvisitedWords, Map<String, List<String>> adjList){

        // for every char -- K *

        // replace it with 26 letters -- 26 *

        // check if it exists in wordSet -- O(1)

        List<String> neighbors = new ArrayList<>();

        char[] wordSeq = word.toCharArray();

        for(int i=0; i<this.L; i++){

            char oldC = wordSeq[i];

            for(int j=0; j<26; j++){

                char newC = (char)('a'+j);

                if(newC==oldC) continue;

                wordSeq[i]=newC;

                String newWord = new String(wordSeq);

                if(unvisitedWords.contains(newWord)){

                    neighbors.add(newWord);

                }

                wordSeq[i] = oldC;

            }

        }

        adjList.put(word, neighbors);

        return neighbors;

    }

}
```

## Output :


## Time Complexity : O(n)

## Space Complexity : O(n)

## 10) Course Schedule

**There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [$a_i$, $b_i$] indicates that you must take course $b_i$ first if you want to take course $a_i$.**

- **For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.**

**Return true if you can finish all courses. Otherwise, return false.**

**Example 1:**

**Input: numCourses = 2, prerequisites = [[1,0]]**

**Output: true**

**Explanation: There are a total of 2 courses to take.**

**To take course 1 you should have finished course 0. So it is possible.**

**Example 2:**

**Input: numCourses = 2, prerequisites = [[1,0],[0,1]]**

**Output: false**

**Explanation: There are a total of 2 courses to take.**

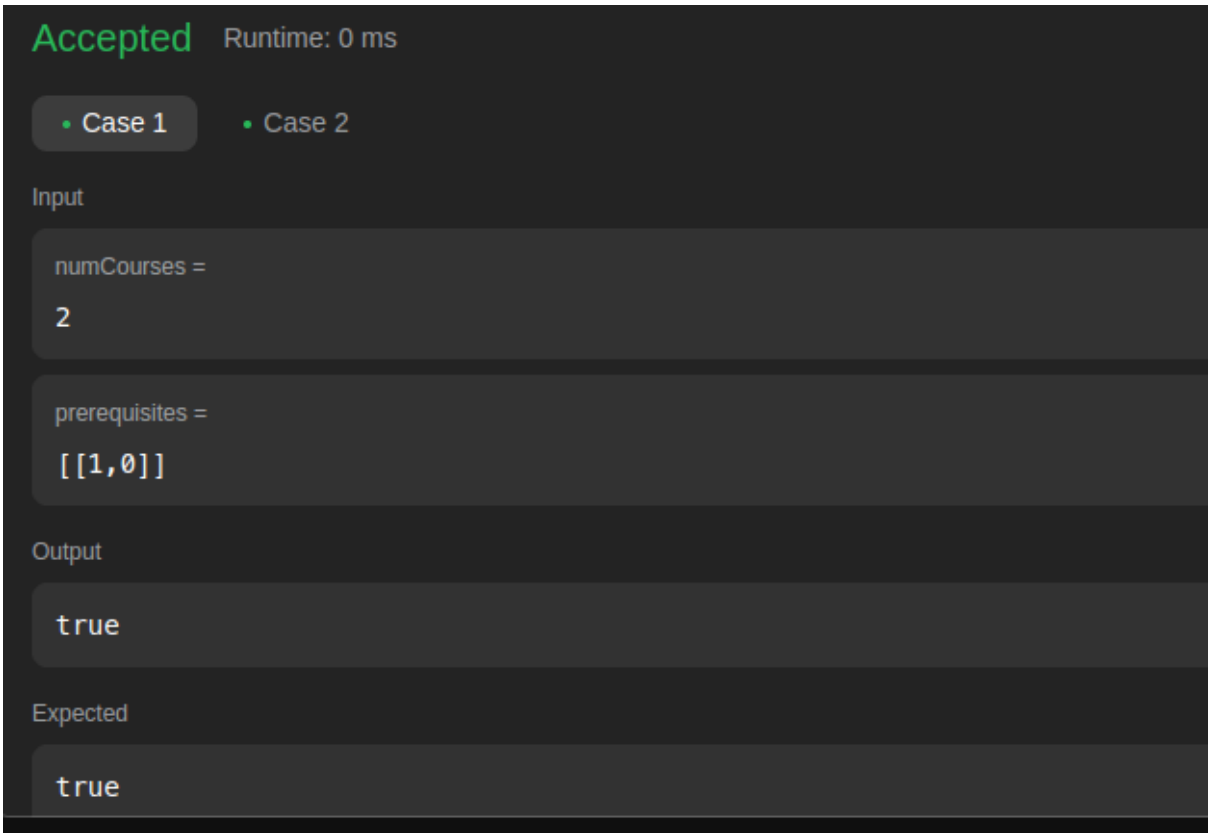
**To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible**

## Program :

```
class Solution {

   public boolean canFinish(int numCourses, int[][] prerequisites) {

        int counter = 0;

        if (numCourses <= 0) {

            return true;

        }

        int[] inDegree = new int[numCourses];

        List<List<Integer>> graph = new ArrayList<>();

        for (int i = 0; i < numCourses; i++) {

            graph.add(new ArrayList<>());

        }

        for (int[] edge : prerequisites) {

            int parent = edge[1];
```

```
            int child = edge[0];
            graph.get(parent).add(child);
            inDegree[child]++;
        }
        Queue<Integer> sources = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                sources.offer(i);
            }
        }
        while (!sources.isEmpty()) {
            int course = sources.poll();
            counter++;

            for (int child : graph.get(course)) {
                inDegree[child]--;
                if (inDegree[child] == 0) {
                    sources.offer(child);
                }
            }
        }

        return counter == numCourses;
    }
}
```

**Output :**




**Time Complexity : O(n)**

**Space Complexity : O(n)**