VISVESVARAYATECHNOLOGICALUNIVERSITY
BELAGAVI

Leetcode Problem Solving Report

Submitted in the partial fulfillment for the requirements of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted By

Madhu gowda A
25UG1BYCS446-T

Under the guidance of

Ms Brunda S
PROFESSOR
Department of CSE, BMSIT&M

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT
YELAHANKA, BENGALURU - 560064.

2025-2026

Evaluation Sheet

Sl. No.	Problem Type	Problem Name	Marks (Code)	Marks (Test Cases)	Marks (Output)	Total
1	Easy	**Valid Parentheses**				
2	Easy	**Implement Stack using Queue**				
3	Easy	**Implement Queue using Stacks**				
4	Easy	**Merge Two Sorted Lists**				
5	Easy	**Remove Duplicates from Sorted List**				
6	Easy	**Linked List Cycle**				
7	Easy	**Binary Tree Inorder Traversal.**				
8	Easy	**Find if the path exists in graph**				
9	Easy	**Flood Fill**				
10	Easy	**Maximum Depth of Binary Tree**				
11	Medium	**Min Stack.**				
12	Medium	**Basic Calculator II**				
13	Medium	**Design Circular Queue**				
14	Medium	**Remove Duplicate from Sorted List II**				
15	Medium	**Add Two Numbers.**				

Evaluation Sheet

Sl. No.	Problem Type	Problem Name	Marks (Code)	Marks (Test Cases)	Marks (Output)	Total
16	Medium	Remove Nth Node From End of List.				
17	Medium	Rotting Oranges				
18	Medium	Number of Island				
19	Medium	Unique Binary				
20	Medium	Unique Binary Search Trees II				
21	Hard	Trapping Rain Water				
22	Hard	Reconstruct Itinerary				
23	Hard	DesignCircular Dequeue				
24	Hard	Merge k Sorted Lists				
25	Hard	Reverse Nodes in k-Group				
26	Hard	Binary Tree Maximum Path Sum				
27	Hard	Alien Dictionary				
28	Hard	LRU Cache				
29	Hard	N-Queens				
30	Hard	Longest Valid Parentheses.				
Total						

Problem 1:- (EASY)20. VALID PARANETHESES

a. Problem Statement: (EASY)20.Valid Parentheses

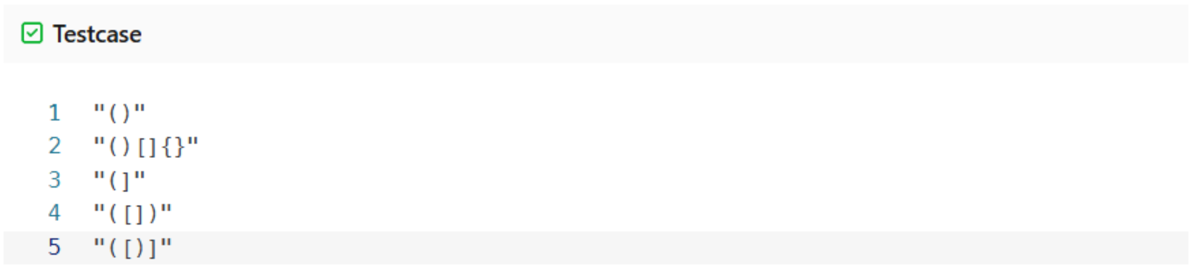
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type

b. Code Implementation:

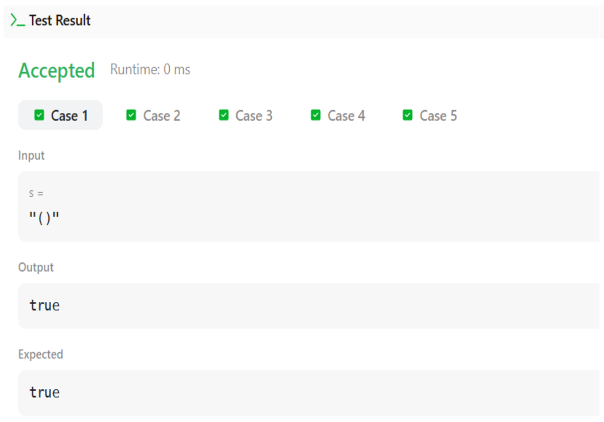
```
bool isMatch(char open, char close) {
    if (open == '(' && close == ')') return true;
    if (open == '{' && close == '}') return true;
    if (open == '[' && close == ']') return true;
    return false;
}

bool isValid(char * s){
    int n = strlen(s);
    char stack[n];
    int top = -1;

    for (int i = 0; i < n; i++) {
        char c = s[i];
        if (c == '(' || c == '{' || c == '[') {
            stack[++top] = c;
        }
        else {
            if (top == -1) return false;
           if (!isMatch(stack[top], c))
                return false;

            top--;
        }
    }

    // If stack empty → valid
    return top == -1;
}
```

c. Test Cases Considered:


Testcase

```
"()"
"()[]{}"
"(]"
"([])"
"([)]"
```

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5

Input

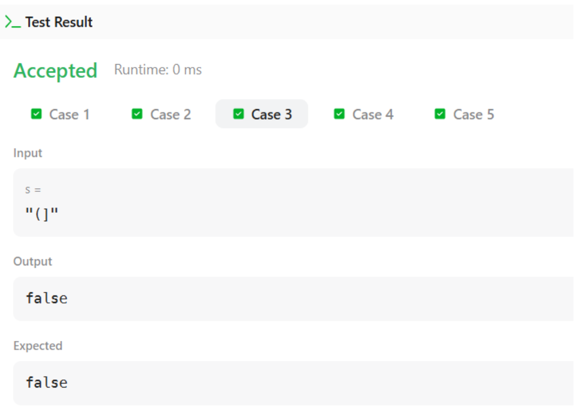
s =
"()"

Output

true

Expected

true


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5

Input

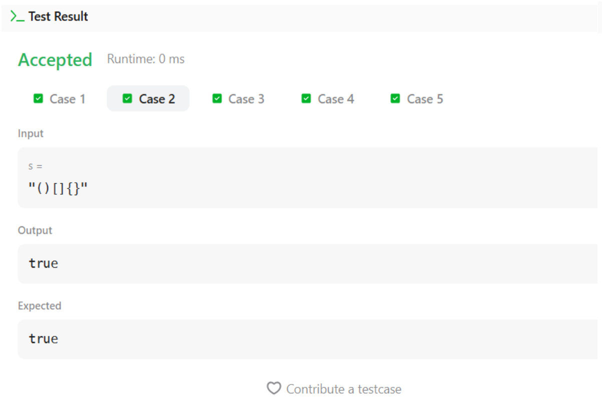
s =
"(]"

Output

false

Expected

false


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5

Input

s =
"()[]{}"

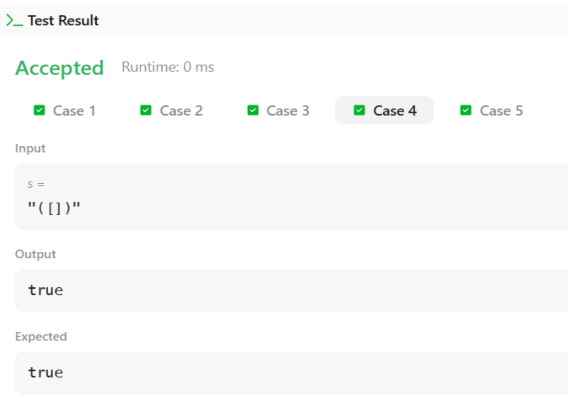
Output

true

Expected

true

Contribute a testcase


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5

Input

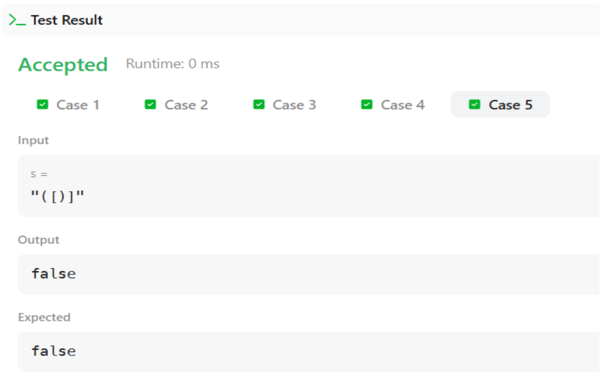
s =
"([])"

Output

true

Expected

true


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5

Input

s =
"([)]"

Output

false

Expected

false

e. Explanation:

Goal

Check if a string of brackets is valid:

- Every opening bracket has a matching closing bracket.
- Brackets close in the correct order.

Approach

- Use a **stack** to track opening brackets.
- For each character:

- If it's an opening bracket → push to stack.
- If it's a closing bracket:

- Check if stack is empty → invalid.
- Check if it matches the top → if not, invalid.
- If matched → pop from stack.

Final Check

- If the stack is empty after processing → valid.
- Else → invalid.

The program uses the Last-In-First-Out (LIFO) property of a stack.

- The most recent opening bracket must be closed first.
- This makes a stack the ideal data structure for bracket validation.

Problem 2 :(EASY) 225. Implementing stack using queues

a. Problem Statement:(EASY) 232. Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).
Implement the MyStack class:

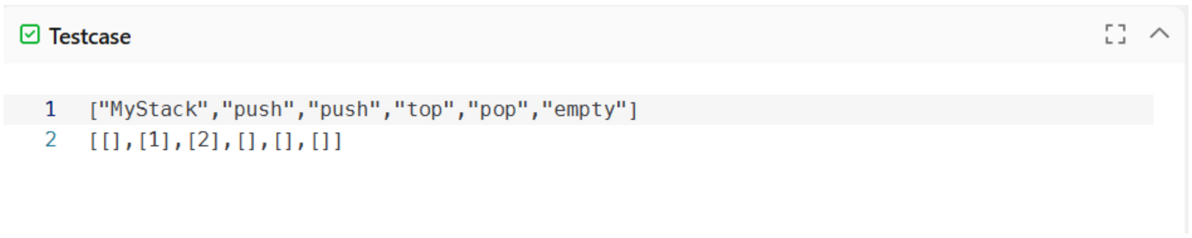
- void push(int x) Pushes element x to the top of the stack.
- int pop() Removes the element on the top of the stack and returns it.
- int top() Returns the element on the top of the stack.
- boolean empty() Returns true if the stack is empty, false otherwise.

b. Code Implementation:

```
typedef struct {
   int *q1;
```

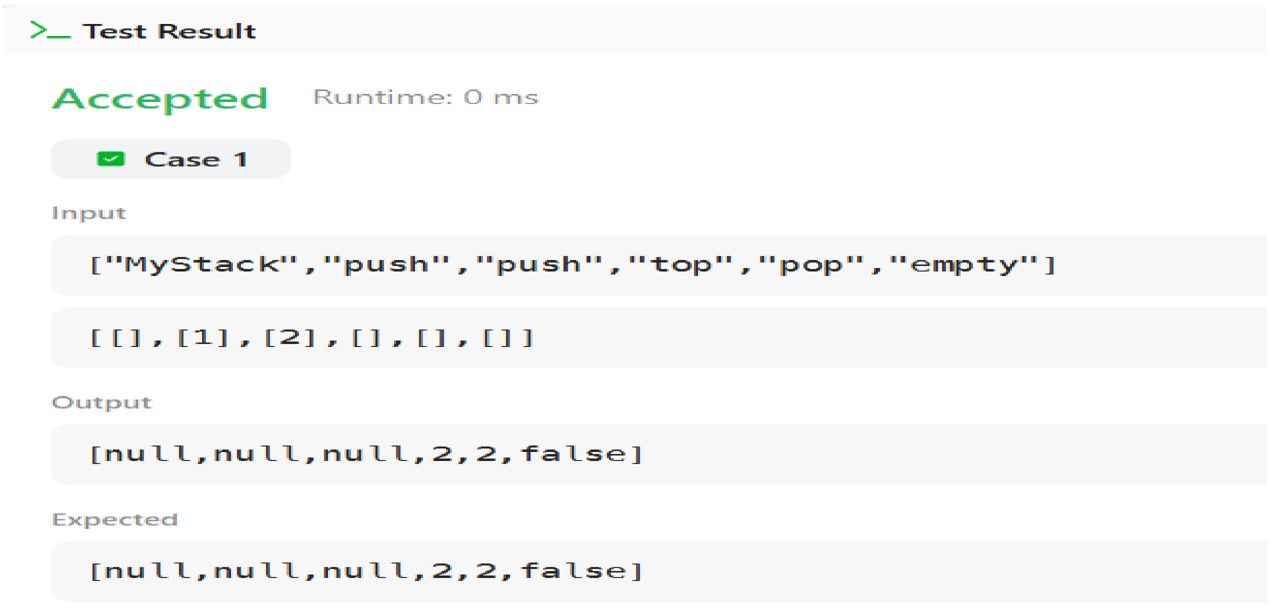
```
    int *q2;
    int front1, rear1;
    int front2, rear2;
    int size;
} MyStack;

MyStack* myStackCreate() {
    MyStack* obj = (MyStack*)malloc(sizeof(MyStack));
    obj->size = 1000;
    obj->q1 = (int*)malloc(sizeof(int) * obj->size);
    obj->q2 = (int*)malloc(sizeof(int) * obj->size);
    obj->front1 = obj->rear1 = 0;
    obj->front2 = obj->rear2 = 0;
    return obj;
}

void myStackPush(MyStack* obj, int x) {
    obj->q2[obj->rear2++] = x;
    while (obj->front1 < obj->rear1)
        obj->q2[obj->rear2++] = obj->q1[obj->front1++];
    int *temp = obj->q1;
    obj->q1 = obj->q2;
    obj->q2 = temp;
    obj->front2 = obj->rear2 = 0;
    obj->front1 = 0;
}

int myStackPop(MyStack* obj) {
    return obj->q1[obj->front1++];
}

int myStackTop(MyStack* obj) {
    return obj->q1[obj->front1];
}

bool myStackEmpty(MyStack* obj) {
    return obj->front1 == obj->rear1;
}

void myStackFree(MyStack* obj) {
    free(obj->q1);
    free(obj->q2);
    free(obj);
}
```

c. Test Cases Considered:


Testcase

```
["MyStack","push","push","top","pop","empty"]
[[],[1],[2],[],[],[]]
```

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

Case 1

Input

```
["MyStack","push","push","top","pop","empty"]
```

```
[[],[1],[2],[],[],[]]
```

Output

```
[null,null,null,2,2,false]
```

Expected

```
[null,null,null,2,2,false]
```

e. Explanation:

Introduction

The goal of this problem is to implement a stack using only queue operations.
A stack follows LIFO (Last In, First Out) behavior, while a queue follows FIFO (First In, First Out) behavior. The challenge is to use queue operations to simulate stack behavior efficiently. This implementation uses two queues to achieve correct ordering.

Concept

Simulate a stack (LIFO) using two queues:

- `q1`: main queue (top of stack is at the front)
- `q2`: helper queue (used during push)

Push Operation

1. Enqueue new element to `q2`.
2. Move all elements from `q1` to `q2`.
3. Swap `q1` and `q2`.

➡ Ensures newest element is always at the front of `q1`

Working Principle Summary

- Each push rearranges the queue so that the newest element is always at the front.
- pop(), top(), and empty() operations become simple queue operations.
- The use of two queues ensures correct LIFO behavior while obeying queue constraints.

Pop Operation

- Dequeue from `q1`.

  ➡ O(1) since top is at the front.

Top Operation

- Return front of `q1`.

Empty Operation

- Return true if `q1` is empty.

Problem 3 : (EASY) 232.Implementing Queue using Stacks.

a. Problem Statement:(EASY) 232. Implement Queue using Stacks

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).
Implement the MyQueue class:

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

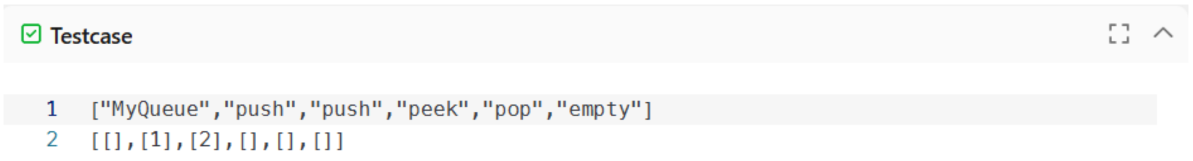
Notes:

- You must use only standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

b. Code Implementation:

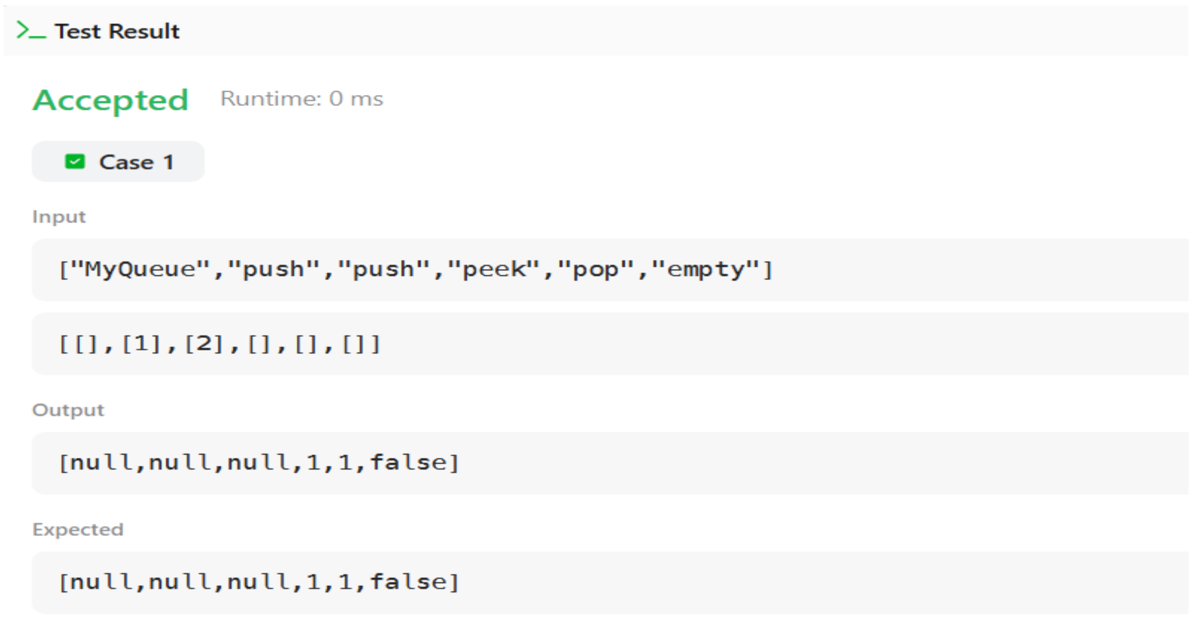
```
typedef struct {
   int *in;
   int *out;
   int topIn;
   int topOut;
   int size;
} MyQueue;

MyQueue* myQueueCreate() {
    MyQueue* obj = (MyQueue*)malloc(sizeof(MyQueue));
    obj->size = 30000;
    obj->in = (int*)malloc(sizeof(int) * obj->size);
    obj->out = (int*)malloc(sizeof(int) * obj->size);
    obj->topIn = -1;
    obj->topOut = -1;

    return obj;
}
void myQueuePush(MyQueue* obj, int x) {
    obj->in[++(obj->topIn)] = x;
}

int myQueuePop(MyQueue* obj) {
    if (obj->topOut == -1) {
        while (obj->topIn != -1) {
            obj->out[++(obj->topOut)] = obj->in[(obj->topIn)--];
        }
    }
    return obj->out[(obj->topOut)--];
}
int myQueuePeek(MyQueue* obj) {
    if (obj->topOut == -1) {
        while (obj->topIn != -1) {
            obj->out[++(obj->topOut)] = obj->in[(obj->topIn)--];
        }
    }
    return obj->out[obj->topOut];
}
bool myQueueEmpty(MyQueue* obj) {
    return obj->topIn == -1 && obj->topOut == -1;
}
void myQueueFree(MyQueue* obj) {
    free(obj->in);
    free(obj->out);
    free(obj);
}
```

c. Test Cases Considered:



```
["MyQueue","push","push","peek","pop","empty"]
[[],[1],[2],[],[],[]]
```

d. Output Screenshots / Console Output:



Input

```
["MyQueue","push","push","peek","pop","empty"]
```

```
[[],[1],[2],[],[],[]]
```

Output

```
[null,null,null,1,1,false]
```

Expected

```
[null,null,null,1,1,false]
```

e. Explanation:

Introduction

The objective of this program is to implement a **FIFO queue** using only two **LIFO stacks**, as required by the problem constraints. A normal queue removes elements in the order they were inserted (first-in, first-out), while a stack removes elements in reverse order (last-in, first-out). The challenge is to simulate correct queue behavior using only stack operations such as push, pop, top, and empty.

Working Principle Summary

- Push always goes to stack_in.
- Pop/Peek pull from stack_out.
- When stack_out is empty, transfer elements from stack_in.
- This guarantees correct FIFO behavior while only using stack operations.

Even though transfer operations may take O(n) time, each element is moved at most once, making average (amortized) time **O(1)**.

Problem 4 : (EASY) 21.Merge Two Sorted Lists

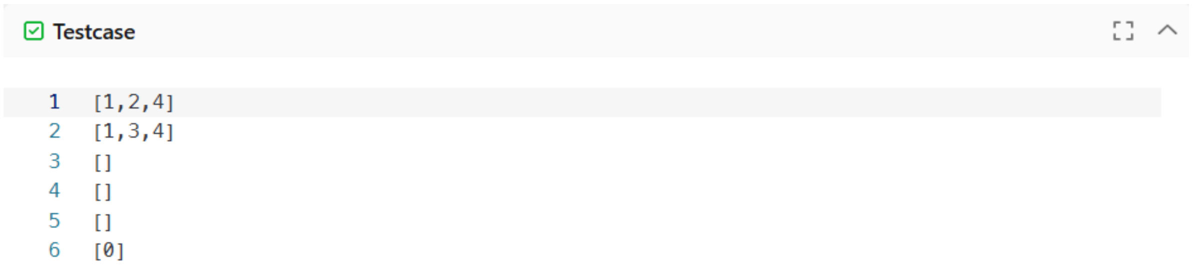
a. Problem Statement:(EASY) 21. Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.
Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.
Return *the head of the merged linked list.*

b. Code Implementation:

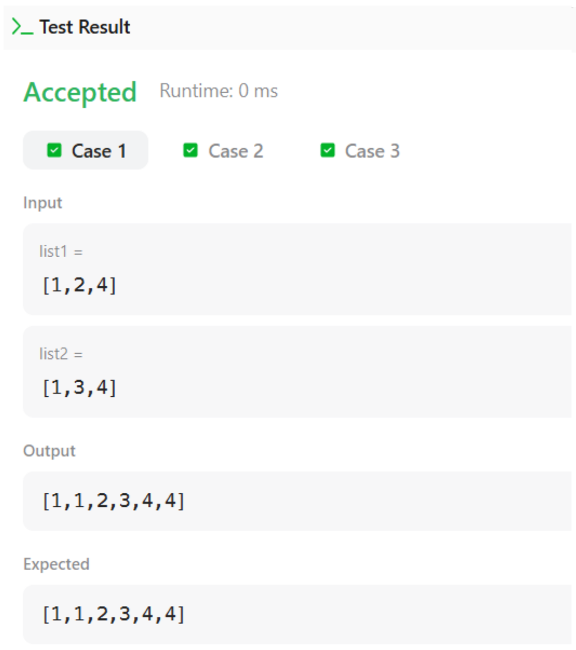
```
struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2) {
    struct ListNode *dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode *k = dummy;

    if (list1 == NULL) return list2;
    if (list2 == NULL) return list1;

    while (list1 != NULL && list2 != NULL) {
        if (list1->val < list2->val) {
            k->next = list1;
            list1 = list1->next;
             k = k->next;
        } else {
            k->next = list2;
            list2 = list2->next;
            k = k->next;
        }
    }
    if (list1 != NULL)
        k->next = list1;
    else
        k->next = list2;
    struct ListNode *result = dummy->next;
    free(dummy);
    return result;
}
```

c. Test Cases Considered:



```
[1,2,4]
[1,3,4]
[]
[]
[]
[0]
```

d. Output Screenshots / Console Output:



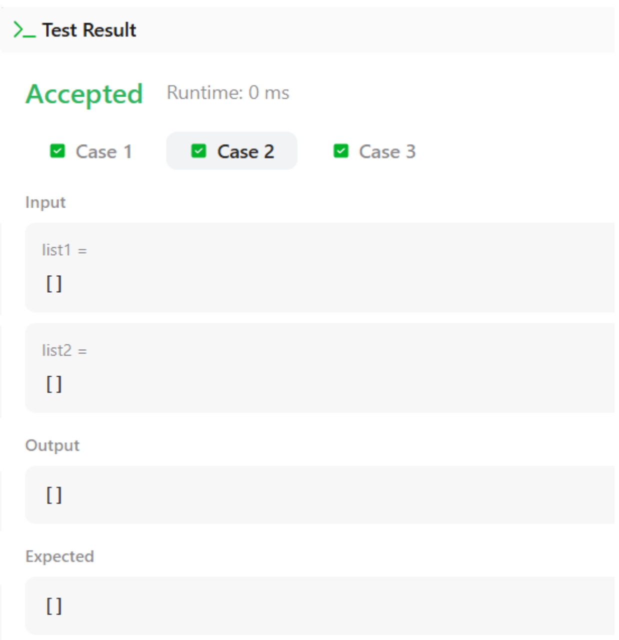
>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

list1 =
```
[1,2,4]
```
list2 =
```
[1,3,4]
```
Output
```
[1,1,2,3,4,4]
```
Expected
```
[1,1,2,3,4,4]
```



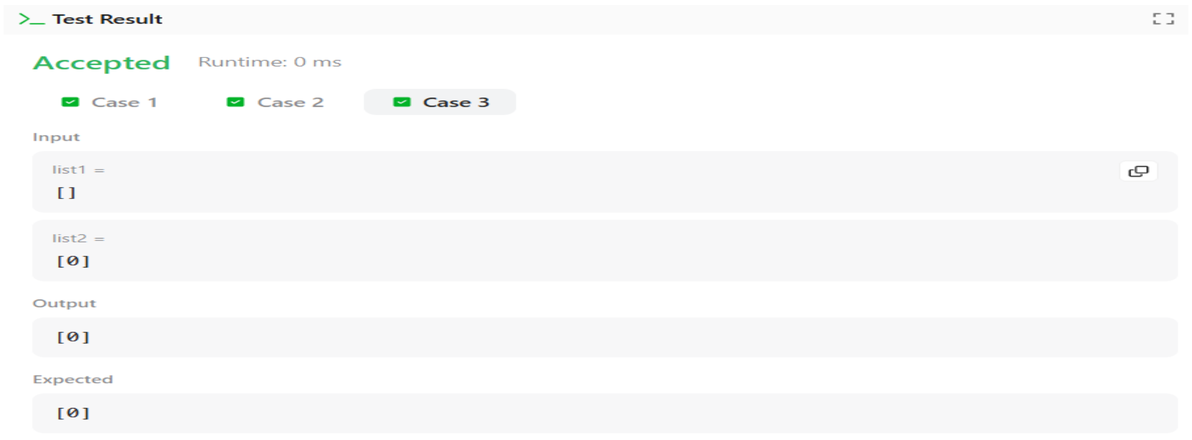
>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

list1 =
```
[]
```
list2 =
```
[]
```
Output
```
[]
```
Expected
```
[]
```



>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

list1 =
```
[]
```
list2 =
```
[0]
```
Output
```
[0]
```
Expected
```
[0]
```

e. Explanation:

Goal

Merge two sorted singly linked lists into one sorted list.

Approach

- Use a **dummy node** to simplify list construction.
- Use a pointer `k` to build the merged list.
- Compare nodes from both lists:
  - Attach the smaller one to `k->next`.
  - Move the pointer in the list from which the node was taken.
- After one list ends, attach the remaining nodes from the other list.

Final Step

- Return `dummy->next` as the head of the merged list.
- Free the dummy node to avoid memory leaks.

Problem 5: (EASY) 83. Remove Duplicates from Sorted List.

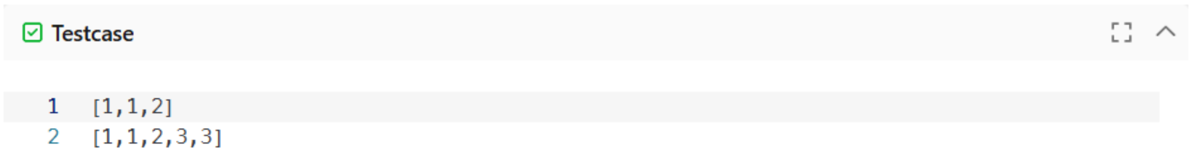
a. Problem Statement: (EASY) 83. Remove Duplicates from Sorted List

Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list sorted as well*.

b. Code Implementation:

```
struct ListNode* deleteDuplicates(struct ListNode* head) {
    if (head == NULL) {
        return head;
    }
    struct ListNode *temp = head;
    struct ListNode *cur, *prev;
    int item;
    while (temp != NULL) {
        item = temp->val;
        prev = temp;
        cur = temp->next;
        while (cur != NULL) {
            if (cur->val == item) {
                prev->next = cur->next;
                free(cur);
                cur = prev->next;
            } else {
                prev = cur;
                cur = cur->next;
            }
        }
        temp = temp->next;
    }
    return head;
}
```

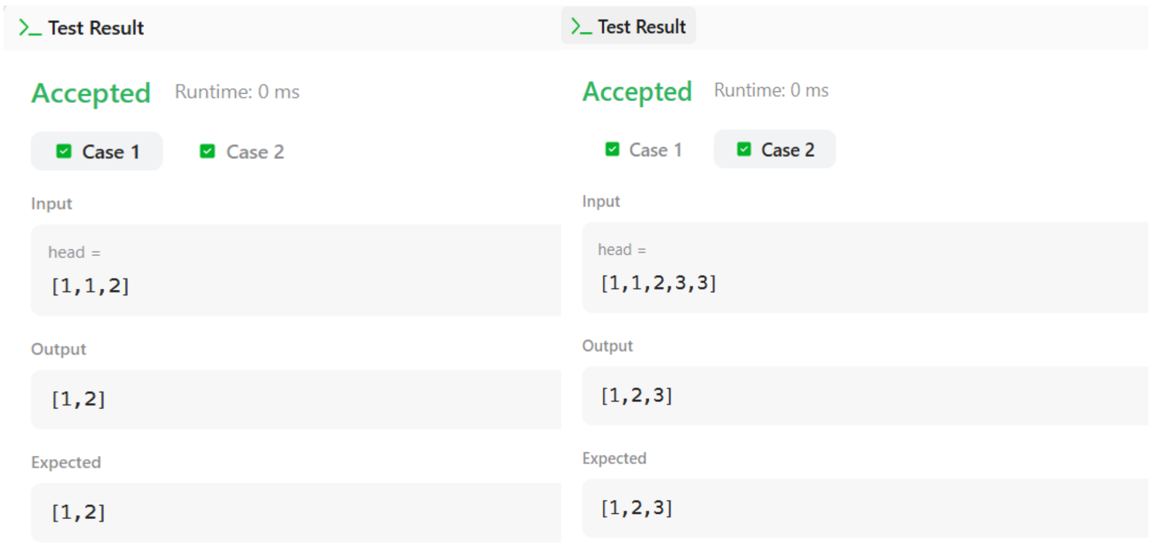
c. Test Cases Considered:



Testcase

```
[1,1,2]
[1,1,2,3,3]
```

d. Output Screenshots / Console Output:



```
>_ Test Result

Accepted   Runtime: 0 ms

Case 1   Case 2

Input
head =
[1,1,2]

Output
[1,2]

Expected
[1,2]
```

```
>_ Test Result

Accepted   Runtime: 0 ms

Case 1   Case 2

Input
head =
[1,1,2,3,3]

Output
[1,2,3]

Expected
[1,2,3]
```

e. Explanation:

Introduction

Goal

Delete all duplicate nodes from an unsorted singly linked list so each value appears only once.

Approach

- Use **two nested loops**:
  - Outer loop (`temp`) picks each node.
  - Inner loop (`cur`) checks for duplicates of `temp->val` in the rest of the list.
- Use a `prev` pointer to help delete duplicate nodes safely.

Problem 6 : (EASY) 141. Linked list cycle

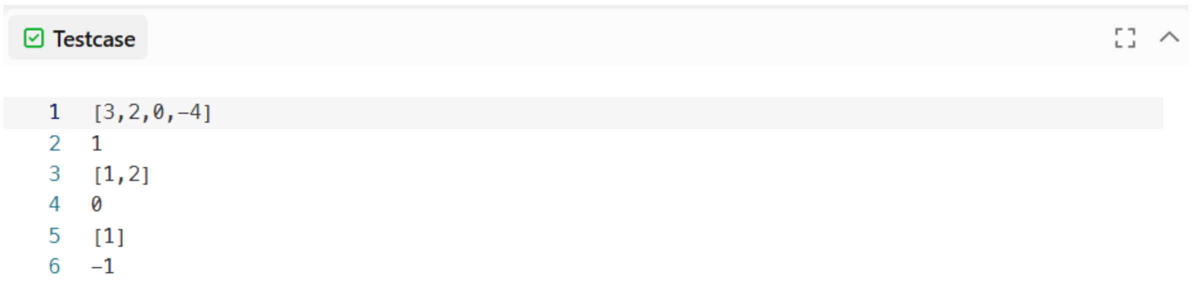
a. Problem Statement: (EASY)141. Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it.
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.
Return true *if there is a cycle in the linked list*. Otherwise, return false.

b. Code Implementation:

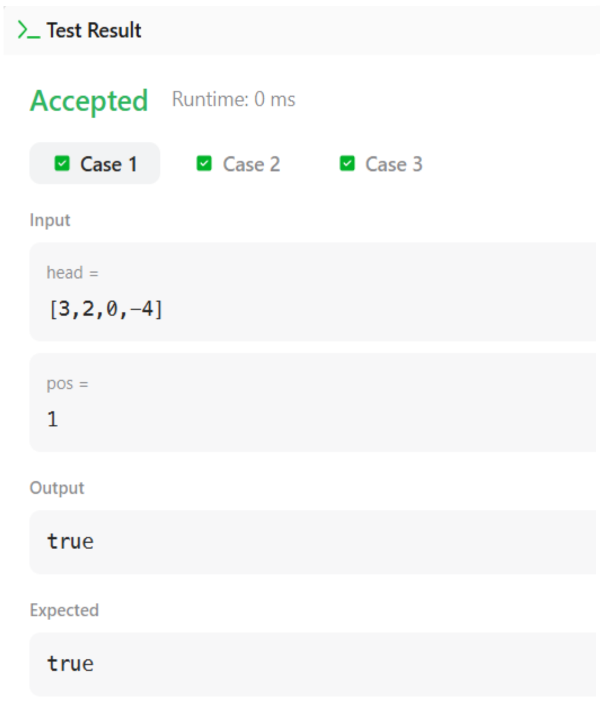
```
bool hasCycle(struct ListNode *head) {
    if (head == NULL || head->next == NULL)
        return false;
    struct ListNode *slow = head;
    struct ListNode *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            return true;
    }
    return false;
}
```

c. Test Cases Considered:


Testcase

```
[3,2,0,-4]
1
[1,2]
0
[1]
-1
```

d. Output Screenshots / Console Output:

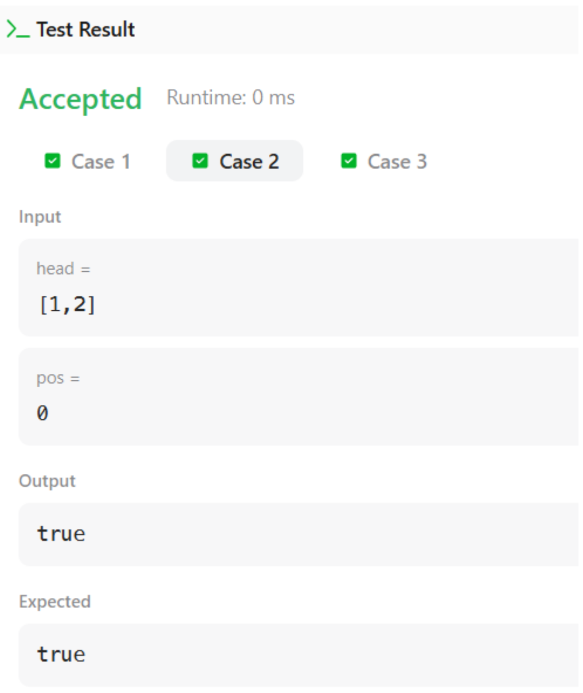

>_ Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3

Input

head =
```
[3,2,0,-4]
```
pos =
```
1
```
Output
```
true
```
Expected
```
true
```

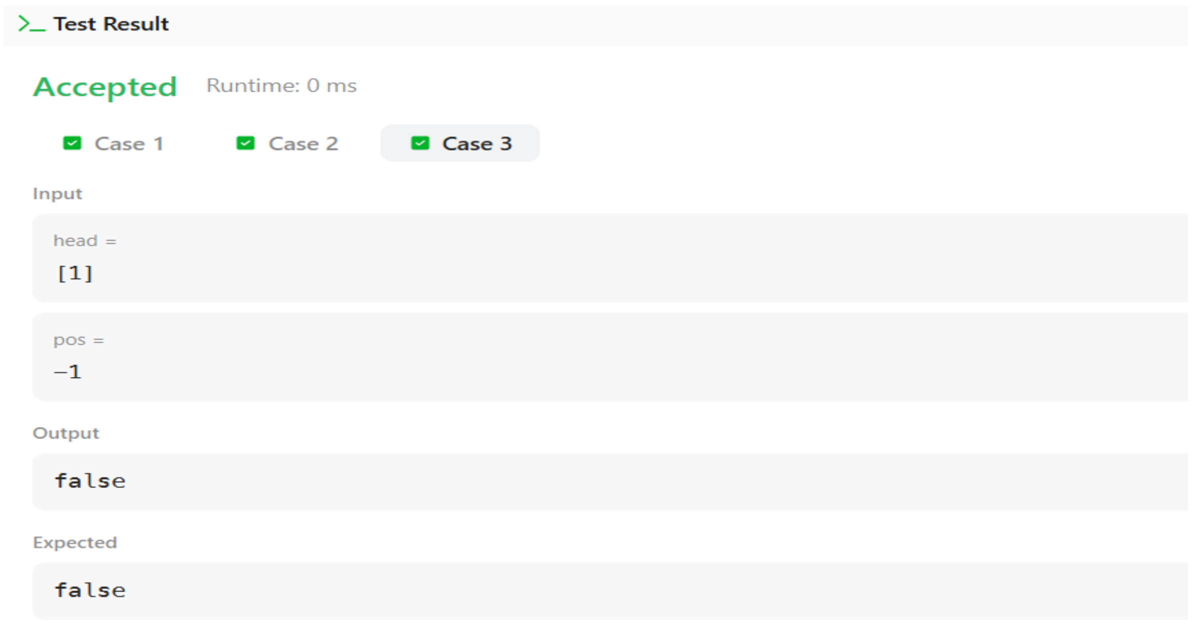

>_ Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3

Input

head =
```
[1,2]
```
pos =
```
0
```
Output
```
true
```
Expected
```
true
```


>_ Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3

Input

head =
```
[1]
```
pos =
```
-1
```
Output
```
false
```
Expected
```
false
```

e. Explanation:

Introduction
This program checks whether a singly linked list contains a cycle, meaning a node's next pointer loops back to a previous node, causing infinite traversal.
Example of a cycle:

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[d] \\
 & 5 \arrow[u] & 4 \arrow[l]
\end{tikzcd}$$

The goal is to detect a cycle without using extra memory.

Algorithm Overview
The solution uses Floyd's Cycle Detection Algorithm (Tortoise and Hare):

- slow pointer moves 1 step at a time
- fast pointer moves 2 steps at a time
- If a cycle exists, slow and fast will eventually meet
- If fast reaches NULL, no cycle exists

This approach is efficient and uses constant space.

Key Steps
1. Initialize Pointers
```
struct ListNode* slow = head;
struct ListNode* fast = head;
```

2. Move Pointers

- slow → 1 step
- fast → 2 steps

If they meet at the same node → cycle detected.

3. End Condition
If fast becomes NULL or fast->next == NULL, the list has no cycle.

Working Principle Summary

- The fast pointer moves twice as quickly and will eventually catch up to slow if the list cycles.
- If the list is linear, fast naturally reaches NULL.
- The method requires no extra memory, only two pointers.

Problem 7 : (EASY) 94.Binary Tree Inorder Traversal.

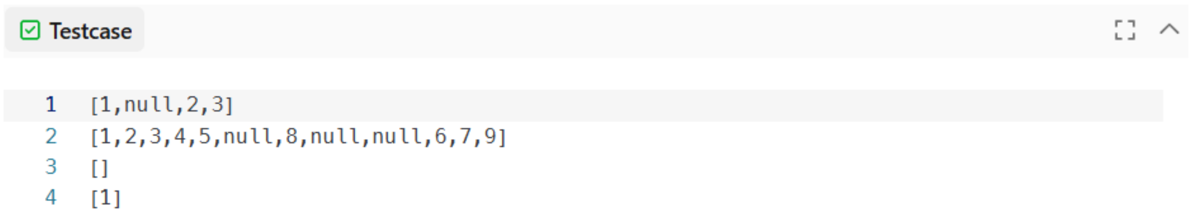
a. Problem Statement: Binary Tree Inorder Traversal

Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

b. Code Implementation:

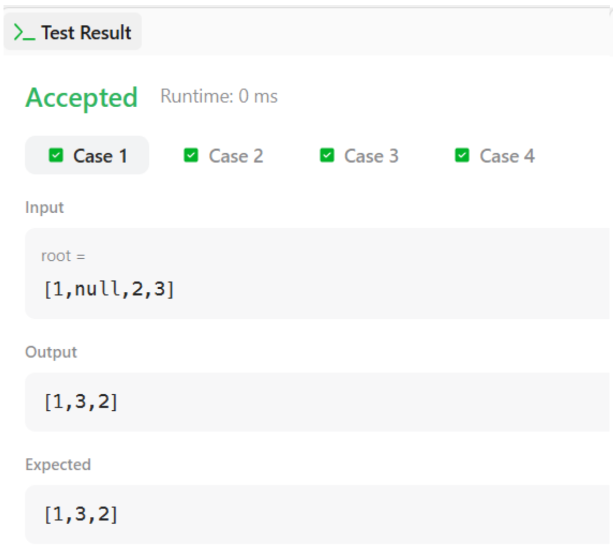
```
void inorder(struct TreeNode* root, int* arr, int* returnSize) {
    if (root == NULL) return;
    inorder(root->left, arr, returnSize);
    arr[(*returnSize)++] = root->val;
    inorder(root->right, arr, returnSize);
}
int* inorderTraversal(struct TreeNode* root, int* returnSize) {
    *returnSize = 0;
    int* arr = (int*)malloc(sizeof(int) * 110);
    inorder(root, arr, returnSize);
    return arr;
}
```

c. Test Cases Considered:


Testcase

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

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4

Input

root =

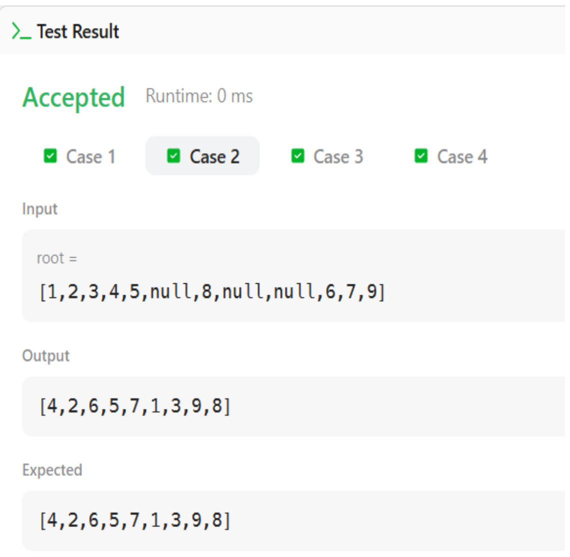
```
[1,null,2,3]
```

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4

Input

root =

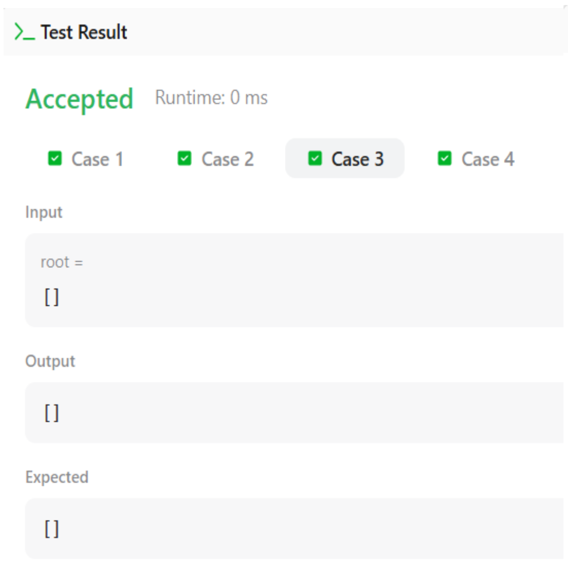
```
[1,2,3,4,5,null,8,null,null,6,7,9]
```

Output

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

Expected

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


>_ Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3 ✓ Case 4

Input

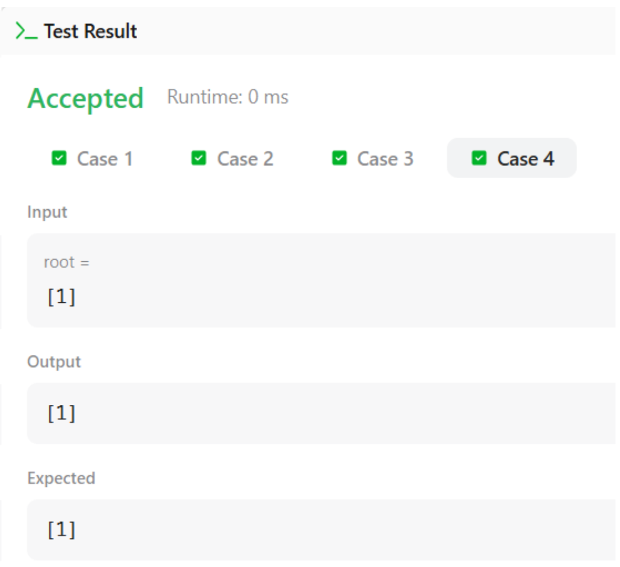
```
root =
[]
```

Output

```
[]
```

Expected

```
[]
```


>_ Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3 ✓ Case 4

Input

```
root =
[1]
```

Output

```
[1]
```

Expected

```
[1]
```

e. Explanation:

Introduction
This program performs an inorder traversal of a binary tree:
Left Subtree → Node → Right Subtree.
For binary search trees, this produces values in sorted order.
The task is to return all visited values in correct inorder sequence.

Algorithm Overview
The algorithm uses:

- Recursion → naturally follows inorder structure
- Dynamic array → stores output values
- returnSize pointer → tracks how many values have been recorded

Traversal continues until all nodes are processed.

Key Steps
1. Traverse Left Subtree
Call:
inorder(root->left);

2. Visit Current Node
arr[*returnSize] = root->val;
(*returnSize)++;

3. Traverse Right Subtree
Call:
inorder(root->right);

Working Principle Summary

- Recursion strictly follows Left → Node → Right order.
- Values are appended in traversal sequence.

- Null children stop recursion naturally.
  This guarantees a correct inorder output for any binary tree.

Problem 8 : Problem: 1971. Find if Path Exists in Graph (Easy)

a. Problem Statement:

Given an undirected graph with n nodes and a list of edges, determine whether there is a valid path between two given vertices source and destination.

b. Code Implementatio

```
#include <stdbool.h>

#include <stdlib.h

void dfs(int node, int **adj, int *visited, int n) {

    visited[node] = 1;

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

        if (adj[node][i] && !visited[i]) {

            dfs(i, adj, visited, n);

        }

    }

}

bool validPath(int n, int** edges, int edgesSize, int* edgesColSize,

               int source, int destination)

    int *adj = (int)malloc(n * sizeof(int));

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

        adj[i] = (int*)calloc(n, sizeof(int));

    }


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

       adj[edges[i][0]][edges[i][1]] = 1;

        adj[edges[i][1]][edges[i][0]] = 1;
```

```
     }

    int visited = (int)calloc(n, sizeof(int));
    dfs(source, adj, visited, n);

    return visited[destination];
}
```

c. Test Cases Considered:

Case 1:

Input: n = 3, edges = [[0,1],[1,2]], source = 0, destination = 2

Output: true

Case 2:

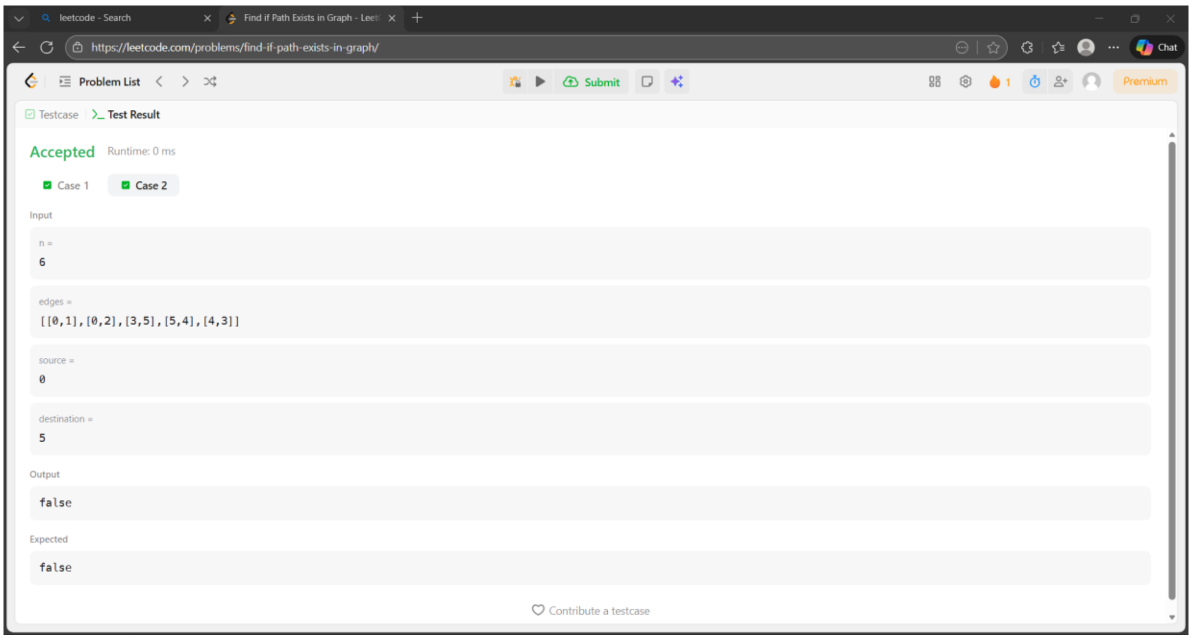
Input: n = 3, edges = [[0,1],[1,2]], source = 0, destination = 3

Output: false

d. Explanation:

The graph is represented using an adjacency matrix. Depth First Search (DFS) is started from the source node. If the destination node is visited during DFS traversal, then a valid path exists. The visited array avoids revisiting nodes. DFS ensures all reachable nodes are explored. The time complexity is $O(n^2)$.

d. Output Screenshots / Console Output:



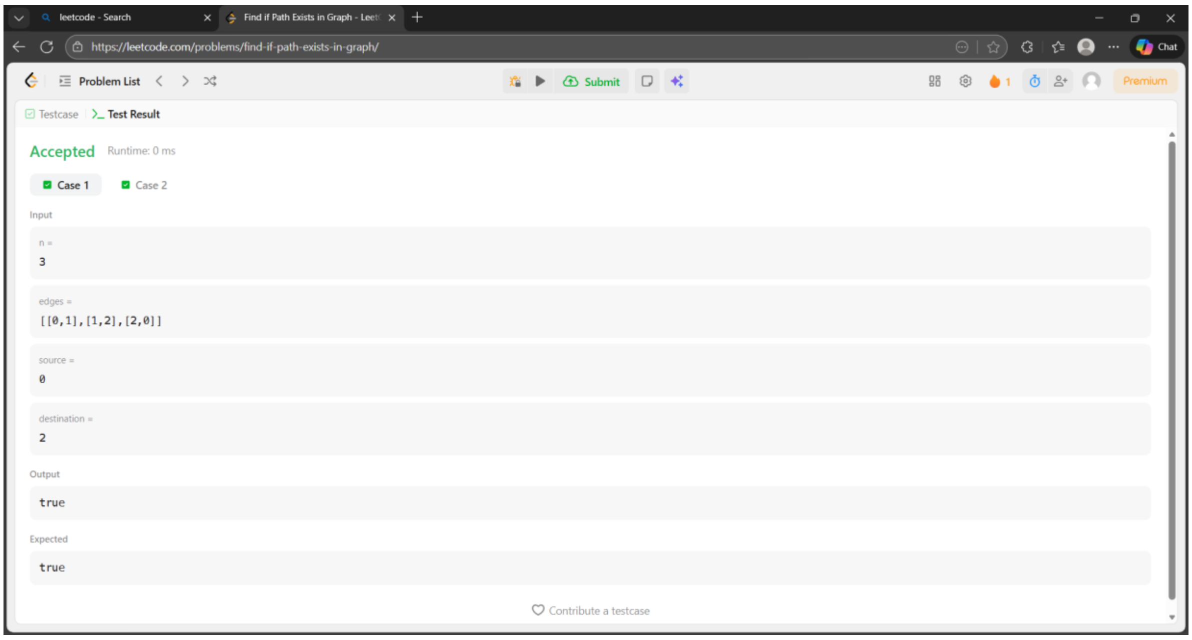
leetcode - Search
Find if Path Exists in Graph - Leet
https://leetcode.com/problems/find-if-path-exists-in-graph/
Problem List
Submit
Premium
Testcase
Test Result
Contribute a testcase

Accepted Runtime: 0 ms

Case 1 Case 2

Input

n =
```
6
```
edges =
```
[[0,1],[0,2],[3,5],[5,4],[4,3]]
```
source =
```
0
```
destination =
```
5
```
Output
```
false
```
Expected
```
false
```



leetcode - Search
Find if Path Exists in Graph - Leet
https://leetcode.com/problems/find-if-path-exists-in-graph/
Problem List
Submit
Premium
Testcase
Test Result
Contribute a testcase

Accepted Runtime: 0 ms

Case 1 Case 2

Input

n =
```
3
```
edges =
```
[[0,1],[1,2],[2,0]]
```
source =
```
0
```
destination =
```
2
```
Output
```
true
```
Expected
```
true
```

Problem 9)*733. FloodFill (Easy)*

a. Problem Statement:

Given a 2D image and a starting pixel, replace the color of the starting pixel and all connected pixels of the same color with a new color.

b. Code Implementation

```
void dfs(int** image, int r, int c, int sr, int sc,
         int oldColor, int newColor) {

    if (sr < 0 || sc < 0 || sr >= r || sc >= c)
        return;
    if (image[sr][sc] != oldColor)
        return;

    image[sr][sc] = newColor;

    dfs(image, r, c, sr + 1, sc, oldColor, newColor);
    dfs(image, r, c, sr - 1, sc, oldColor, newColor);
    dfs(image, r, c, sr, sc + 1, oldColor, newColor);
    dfs(image, r, c, sr, sc - 1, oldColor, newColor);
}
int** floodFill(int** image, int imageSize, int* imageColSize,
                int sr, int sc, int color) {
    int oldColor = image[sr][sc];
    if (oldColor != color)
        dfs(image, imageSize, *imageColSize, sr, sc, oldColor,
color);
```

```
    return image;
}
```

c. Test Cases Considered:

Case 1:

Input:

```
image = [[1,1,1],
         [1,1,0],
         [1,0,1]]
sr = 1, sc = 1, newColor = 2
```

Output:

```
[[2,2,2],
 [2,2,0],
 [2,0,1]]
```

Case 2:

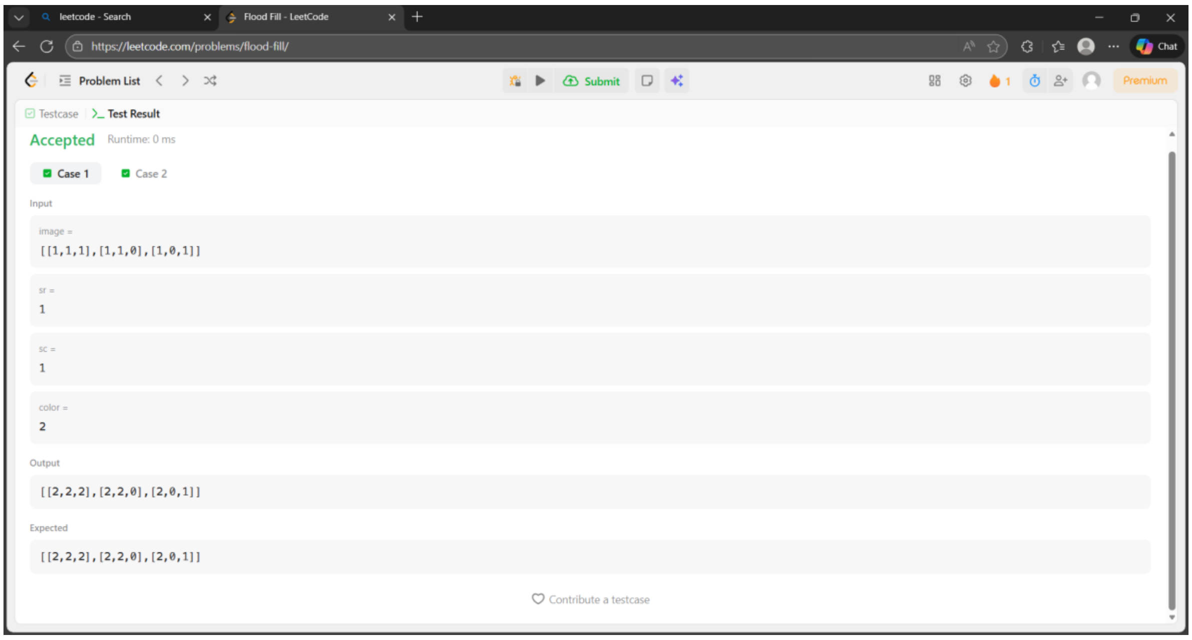
Input: [[0,0,0]], sr = 0, sc = 0, color = 1

Output: [[1,1,1]]

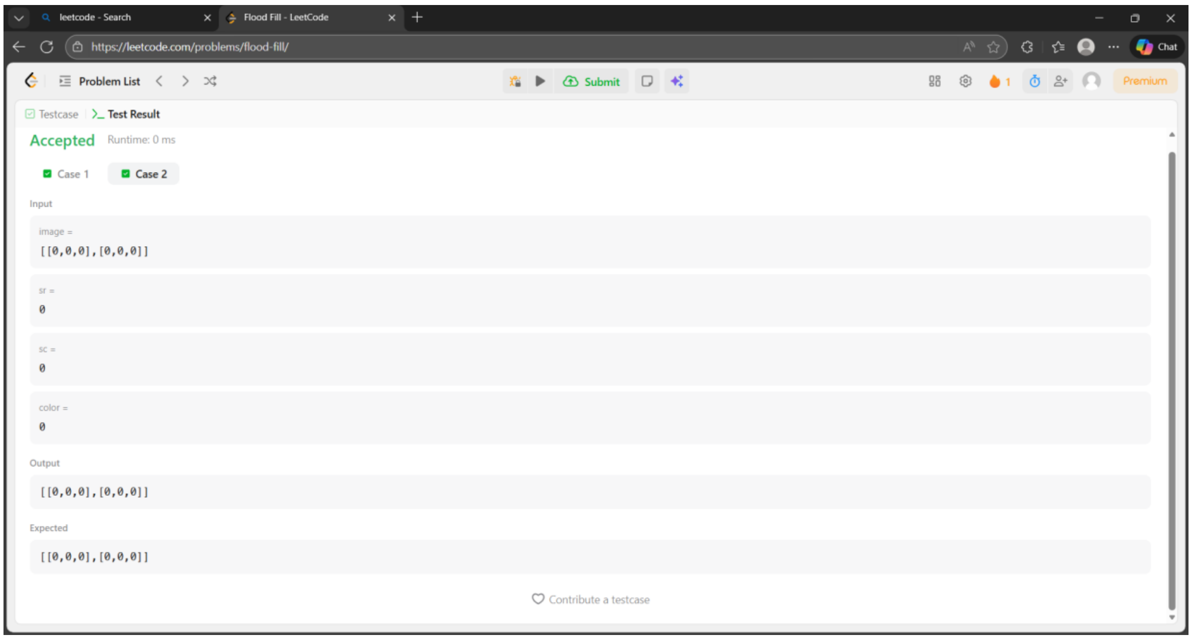
d. Explanation:

Flood Fill uses Depth First Search (DFS) to change the color of the starting pixel and all connected pixels having the same original color. The function checks boundary conditions and ensures only matching color pixels are updated. DFS explores all four directions (up, down, left, right). Time complexity is O(m × n).

d. Output Screenshots / Console Output:



```
Accepted    Runtime: 0 ms
Case 1    Case 2
Input
image =
[[1,1,1],[1,1,0],[1,0,1]]
sr =
1
sc =
1
color =
2
Output
[[2,2,2],[2,2,0],[2,0,1]]
Expected
[[2,2,2],[2,2,0],[2,0,1]]
```



```
Accepted    Runtime: 0 ms
Case 1    Case 2
Input
image =
[[0,0,0],[0,0,0]]
sr =
0
sc =
0
color =
0
Output
[[0,0,0],[0,0,0]]
Expected
[[0,0,0],[0,0,0]]
```

Problem 10 : (EASY) 104. Maximum Depth of Binary Tree.

a. Problem Statement:(EASY) 104. Maximum Depth of Binary Tree

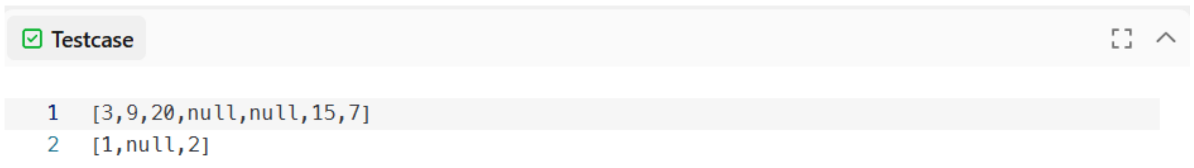
Given the root of a binary tree, return *its maximum depth*.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node

b. Code Implementation:

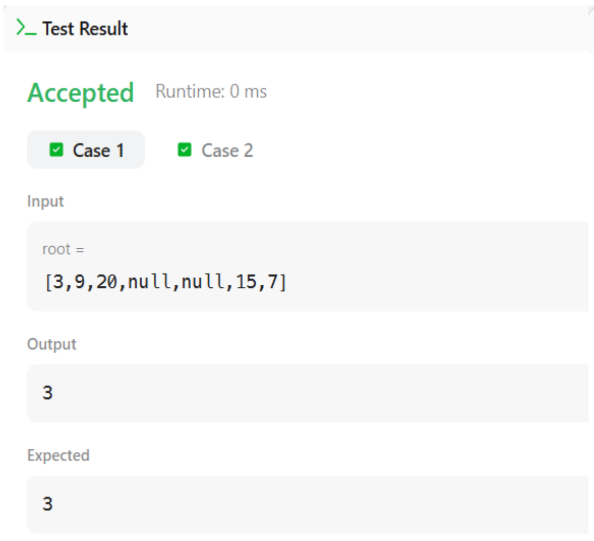
```
int maxDepth(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);

    return 1 + (leftDepth > rightDepth ? leftDepth : rightDepth);
}
```

c. Test Cases Considered:

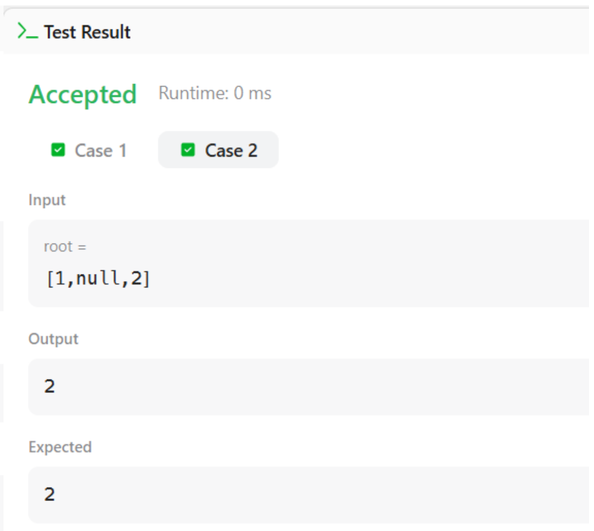


```
[3,9,20,null,null,15,7]
[1,null,2]
```

d. Output Screenshots / Console Output:



```
Input
root =
[3,9,20,null,null,15,7]
Output
3
Expected
3
```



```
Input
root =
[1,null,2]
Output
2
Expected
2
```

e. Explanation:

Introduction
This program computes the maximum depth (or height) of a binary tree.
Depth is defined as the number of nodes along the longest path from the root down to a leaf.

Example:
Tree height = longest root-to-leaf path length.

Algorithm Overview
The algorithm uses **recursion**:

- If the node is NULL, depth is 0.
- Otherwise, compute the left and right subtree depths.
- The depth of the current node is:

1 + max(leftDepth, rightDepth)
This captures the longest path from the current node downward.

Key Steps
1. Base Case
if (root == NULL) return 0;
Empty subtree → depth 0.

2. Recursive Depth Calculation
leftDepth = maxDepth(root->left);
rightDepth = maxDepth(root->right);

3. Return Maximum Path
return 1 + max(leftDepth, rightDepth);

Working Principle Summary

- Recursion travels to all leaf nodes.
- Each recursive call returns subtree depth.
- The maximum depth is built upward from leaves to the root.
- The final result is the height of the entire tree.

Problem 11 : (MEDIUM) 155. Min Stack.

a. Problem Statement:(MEDIUM) 155. Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
Implement the MinStack class:

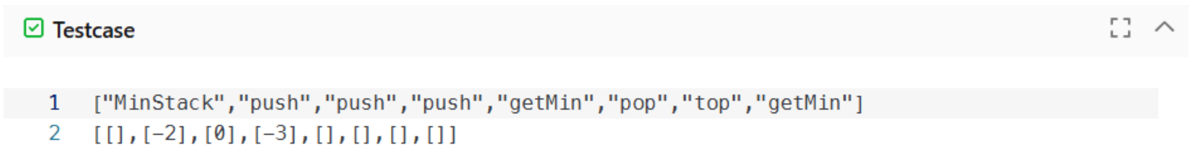
- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

b. Code Implementation:

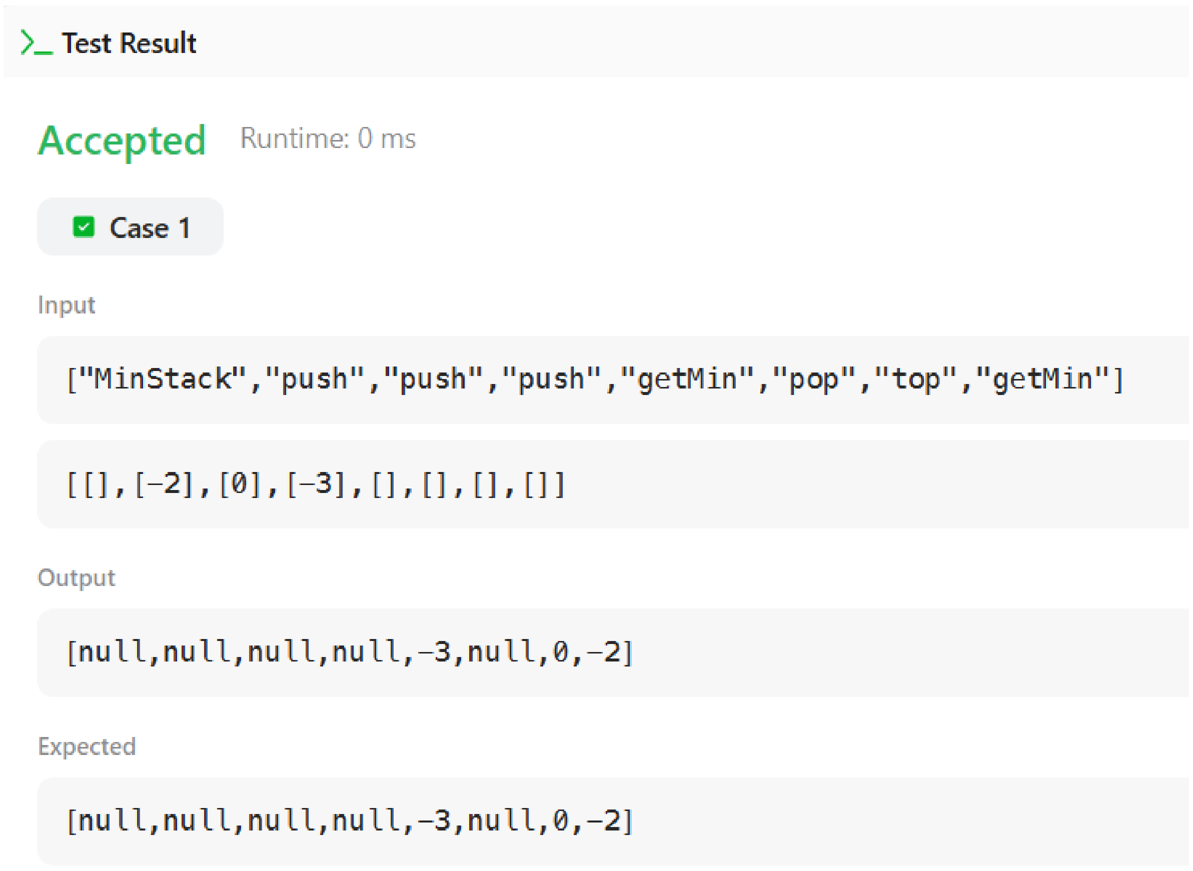
```
typedef struct {
    int *stack;
    int *minStack;
    int top;
    int size;
} MinStack;

MinStack* minStackCreate() {
    int maxSize = 30000;
    MinStack* obj = (MinStack*)malloc(sizeof(MinStack));
    obj->stack = (int*)malloc(sizeof(int) * maxSize);
    obj->minStack = (int*)malloc(sizeof(int) * maxSize);
    obj->top = -1;
    obj->size = maxSize;
    return obj;
}

void minStackPush(MinStack* obj, int val) {
    obj->stack[++(obj->top)] = val;
    if (obj->top == 0)
        obj->minStack[obj->top] = val;
    else {
        int prevMin = obj->minStack[obj->top - 1];
        obj->minStack[obj->top] = (val < prevMin) ? val : prevMin;
    }
}

void minStackPop(MinStack* obj) {
    if (obj->top >= 0)
        obj->top--;
}

int minStackTop(MinStack* obj) {
    return obj->stack[obj->top];
}
int minStackGetMin(MinStack* obj) {
    return obj->minStack[obj->top];
}
void minStackFree(MinStack* obj) {
    free(obj->stack);
    free(obj->minStack);
    free(obj);
}
```

c. Test Cases Considered:


Testcase

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]
```

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

Case 1

Input

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
```

```
[[],[-2],[0],[-3],[],[],[],[]]
```

Output

```
[null,null,null,null,-3,null,0,-2]
```

Expected

```
[null,null,null,null,-3,null,0,-2]
```

e. Explanation:

Introduction

The Min Stack is a special stack data structure that supports retrieving the **minimum element in O(1) time**, in addition to the standard stack operations such as push, pop, and top. The main challenge is to track the minimum value efficiently without scanning the entire stack. This implementation uses two parallel stacks to achieve constant-time operations.

Explanation of the Algorithm

The Min Stack uses **two integer stacks**:

1. **Main Stack (stack)** → Stores all pushed values
2. **Minimum Stack (minStack)** → Tracks the minimum value at each position of the main stack

At any index i:

- stack[i] holds the actual value
- minStack[i] holds the minimum of all values from index 0 to i

This allows instant retrieval of the minimum element by simply checking the top of minStack.

Push Operation

When a value is pushed:

- It is added to the main stack.
- The new minimum is computed by comparing the pushed value with the previous minimum.
- The result is stored on minStack[top].

This ensures that the minimum element is always available at the top of the minStack.

Pop Operation

When popping:

- Only the top pointer is decremented.
- Both stacks remain properly synchronized.
- There is no need for extra computation since minStack already tracked the history of minimums.

Top Operation

Simply returns stack[top], the most recently pushed value.

GetMin Operation

Returns the **current minimum value**, stored at minStack[top].

Data Structure Summary

- stack[] → Actual stored values
- minStack[] → Minimum values so far
- top → Index of the current top of both stacks
- size → Maximum allowed elements (30,000 in LeetCode constraints)

The Min Stack supports all operations in **constant time O(1)**.

Problem 12 : (MEDIUM) 227. Basic Calculator II

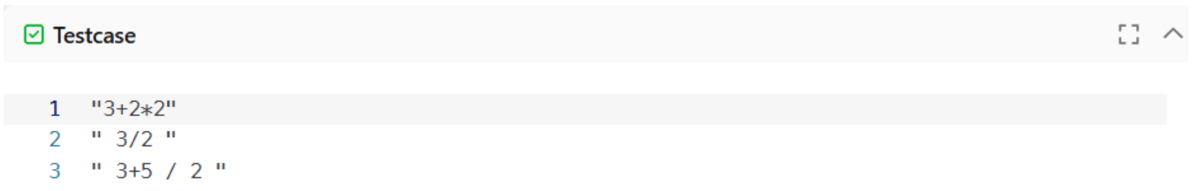
a. Problem Statement: (MEDIUM)227. Basic Calculator II

Given a string s which represents an expression, *evaluate this expression and return its value*.
The integer division should truncate toward zero.
You may assume that the given expression is always valid. All intermediate results will be in the range of [-2^{31}, 2^{31} - 1].
Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

b. Code Implementation:

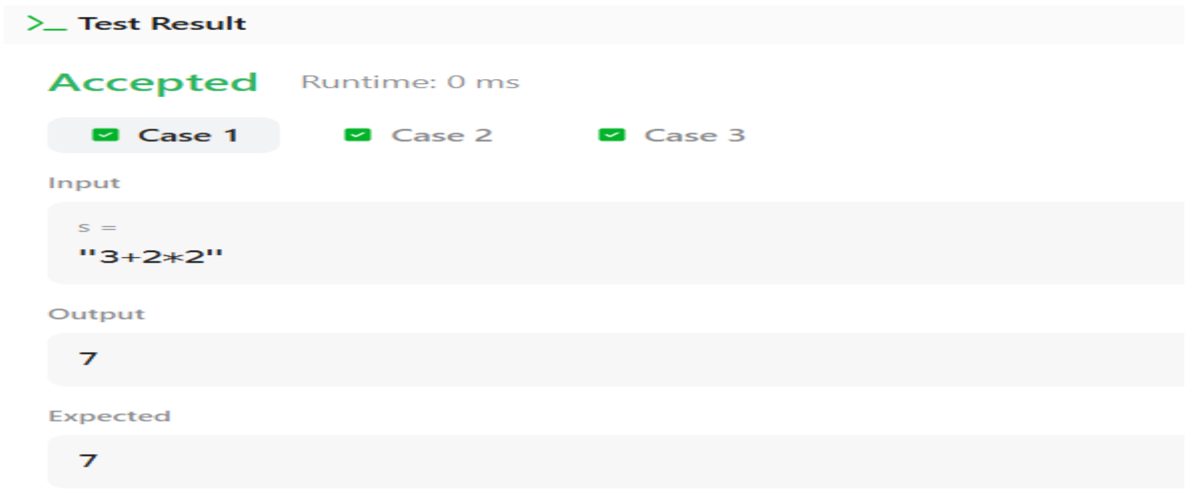
```
int calculate(char* s) {
    int stack[300000];
    int top = -1;
    long num = 0;
    char op = '+';

    for (int i = 0; s[i]; i++) {
        if (isdigit(s[i])) num = num * 10 + (s[i] - '0');

        if ((!isdigit(s[i]) && s[i] != ' ') || s[i+1] == '\0') {
            if (op == '+') stack[++top] = num;
            else if (op == '-') stack[++top] = -num;
            else if (op == '*') stack[top] = stack[top] * num;
            else if (op == '/') stack[top] = stack[top] / num;

            op = s[i];
            num = 0;
        }
    }

    int sum = 0;
    for (int i = 0; i <= top; i++) sum += stack[i];
    return sum;
}
```

c. Test Cases Considered:



```
"3+2*2"
" 3/2 "
" 3+5 / 2 "
```

d. Output Screenshots / Console Output:



Input

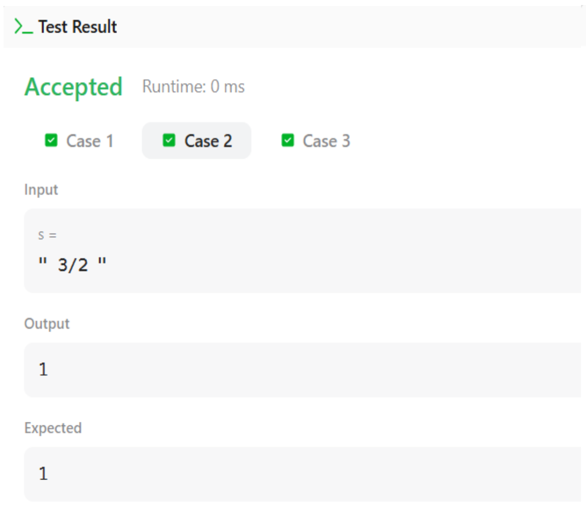
```
s =
"3+2*2"
```

Output

```
7
```

Expected

```
7
```



Input

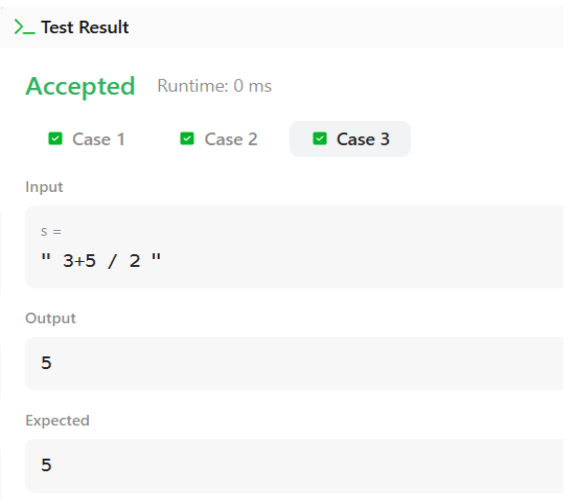
```
s =
" 3/2 "
```

Output

```
1
```

Expected

```
1
```



Input

```
s =
" 3+5 / 2 "
```

Output

```
5
```

Expected

```
5
```

e. Explanation:

Introduction

The objective of this program is to evaluate a mathematical expression containing integers, addition (+), subtraction (-), multiplication (*), and division (/). The expression does not include parentheses, but operator precedence must be respected. Division follows truncation toward zero as specified. The algorithm must process the expression in a single pass while maintaining correct precedence rules.

Algorithm Overview
The program uses a **stack-based approach** to manage operator precedence.

- Lower-priority operators (+ and -) push values directly to the stack.
- Higher-priority operators (* and /) modify the **most recent** value on the stack.

This ensures that multiplication and division are evaluated before addition and subtraction, which matches standard arithmetic rules.

Explanation of Key Components
Building Multi-Digit Numbers
Characters are scanned one by one.
Whenever a digit is found, it is appended to the current number:
num = num * 10 + (s[i] - '0');
This handles multi-digit integers correctly.

Operator Handling
An operator or the end of the string triggers evaluation of the previous number using the last recorded operator (op):

Operator	**Action**
+	Push number to stack
-	Push negative number to stack
*	Multiply number with top of stack
/	Divide top of stack by number

For example, encountering a * means the last pushed value participates in immediate multiplication.
Final Summation
At the end of the scan, all partial results remain on the stack.
Summing them gives the final evaluated result:
for (int i = 0; i <= top; i++) sum += stack[i];
Addition is safe because all multiplication and division have already been resolved.

Working Principle Summary
1. Parse the string from left to right.
2. Build numbers digit by digit.
3. On encountering an operator or the end of the string:
   - Compute using the previous operator.
   - Push or update stack entries based on precedence.
4. Sum the stack to compute the final result.

This method ensures correct handling of mixed operators while operating in a single pass.

Problem 13 : (MEDIUM) 316. Remove Duplicate Letters.

Problem: 622. Design Circular Queue (Medium)
a. Problem Statement:
Design a circular queue and implement its operations. The circular queue should support the following operations:
- **enQueue(value): Insert an element into the circular queue.**
- **deQueue(): Delete an element from the circular queue.**
- **Front(): Get the front item.**
- **Rear(): Get the last item.**
- **isEmpty(): Check whether the circular queue is empty.**

- **isFull(): Check whether the circular queue is full.**

The queue should be implemented using a fixed-size array.

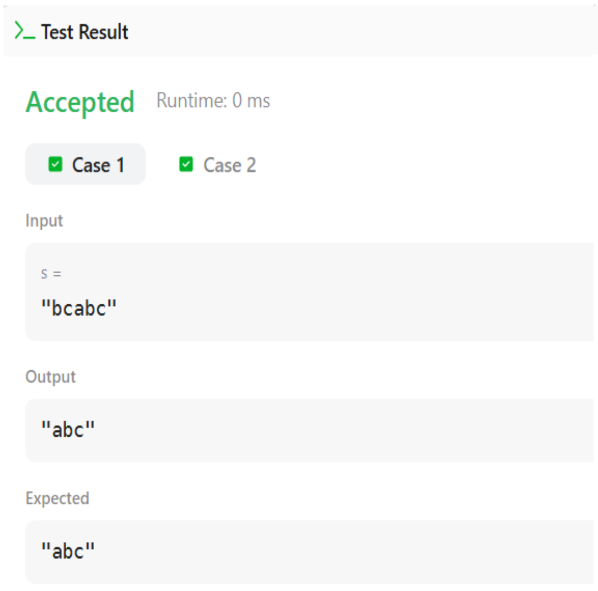
b. Code Implementation (C):

```
typedef struct {
    int *arr;
    int front;
    int rear;
    int size;
    int capacity;
} MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k) {
    MyCircularQueue* q = (MyCircularQueue*)malloc(sizeof(MyCircularQueue));
    q->capacity = k;
    q->size = 0;
    q->front = 0;
    q->rear = -1;
    q->arr = (int*)malloc(sizeof(int) * k);
    return q;
}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
    if (obj->size == obj->capacity)
        return false;

    obj->rear = (obj->rear + 1) % obj->capacity;
    obj->arr[obj->rear] = value;
    obj->size++;
    return true;
}

bool myCircularQueueDeQueue(MyCircularQueue* obj) {
    if (obj->size == 0)
        return false;

    obj->front = (obj->front + 1) % obj->capacity;
    obj->size--;
    return true;
}

int myCircularQueueFront(MyCircularQueue* obj) {
    if (obj->size == 0)
        return -1;
    return obj->arr[obj->front];
}

int myCircularQueueRear(MyCircularQueue* obj) {
    if (obj->size == 0)
        return -1;
    return obj->arr[obj->rear];
}
```

```
bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
    return obj->size == 0;
}

bool myCircularQueueIsFull(MyCircularQueue* obj) {
    return obj->size == obj->capacity;
}

void myCircularQueueFree(MyCircularQueue* obj) {
    free(obj->arr);
    free(obj);
}
```

c. Test Cases Considered:
Case 1:
Input: enQueue(1), enQueue(2), enQueue(3), enQueue(4)
Output: true, true, true, false
Case 2:
Input: Rear(), isFull()
Output: 3, true
Case 3:
Input: deQueue(), enQueue(4), Rear()
Output: true, true, 4

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

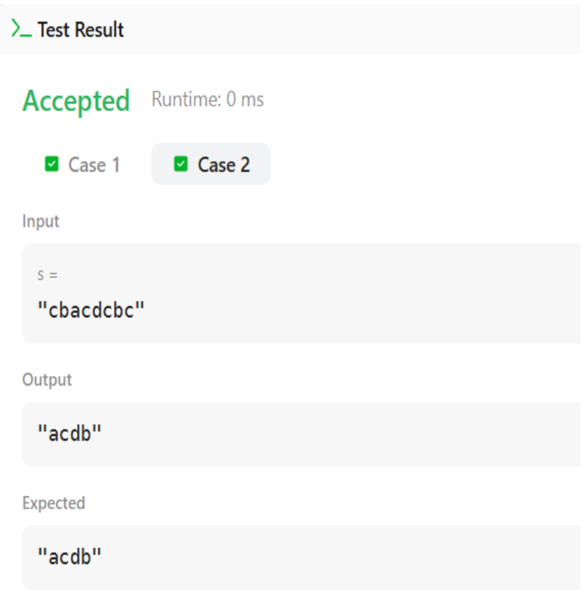
Case 1 Case 2

Input

s =
"bcabc"

Output
"abc"

Expected
"abc"


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

s =
"cbacdcbc"

Output
"acdb"

Expected
"acdb"

e. Explanation:

The Design Circular Queue problem implements a queue using a fixed-size array where the last position connects back to the first, forming a circle. Two pointers, front and rear, are used to track the start and end of the queue. The modulo operator % is used to move the pointers circularly when they reach the end of the array. This avoids wasted space that occurs in a normal queue. The size variable helps in checking whether the queue is full or empty. All operations such as insertion, deletion, and access take O(1) time, making the circular queue efficient.

Problem 14 : (MEDIUM) 82. Remove Duplicates from Sorted List II

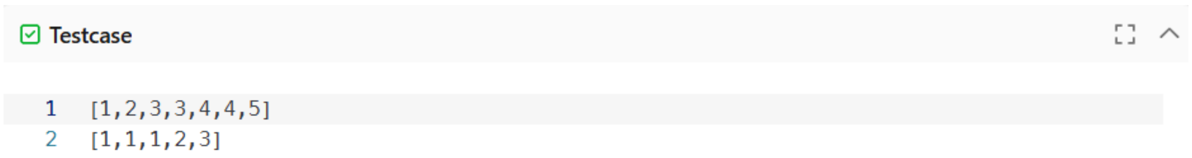
a. Problem Statement: (MEDIUM) 82. Remove Duplicates from Sorted List II

Given the head of a sorted linked list, *delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list*. Return *the linked list sorted as well*.

b. Code Implementation:

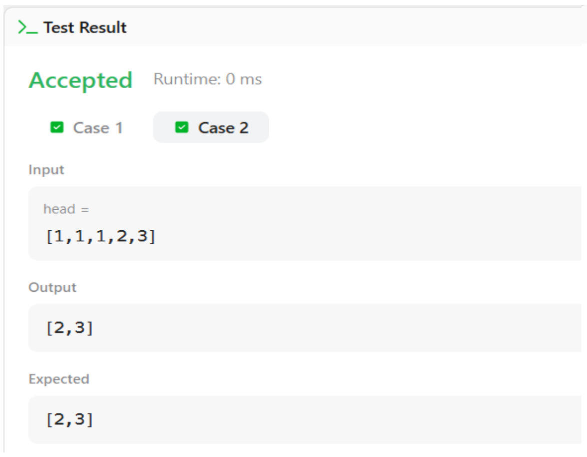
```
struct ListNode* deleteDuplicates(struct ListNode* head) {
    if (!head) return head;
    struct ListNode *dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;
    struct ListNode *prev = dummy;
    struct ListNode *cur = head;
    while (cur != NULL) {
        if (cur->next != NULL && cur->val == cur->next->val) {
            int dup = cur->val;
            while (cur != NULL && cur->val == dup) {
                struct ListNode* temp = cur;
                cur = cur->next;
                free(temp);
            }
            prev->next = cur;
        } else {
            prev = cur;
            cur = cur->next;
        }
    }
    head = dummy->next;
    free(dummy);
    return head;
}
```

c. Test Cases Considered:

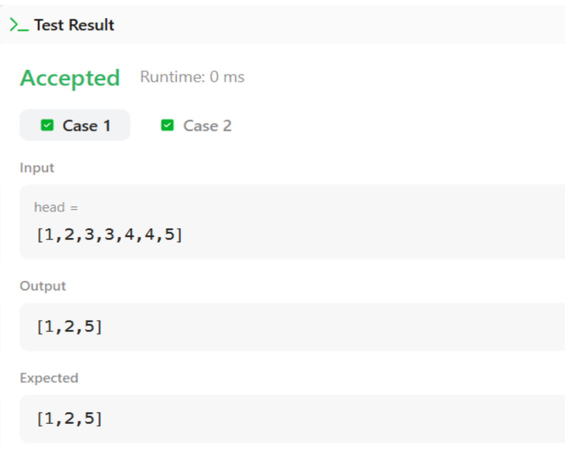


```
[1,2,3,3,4,4,5]
[1,1,1,2,3]
```

d. Output Screenshots / Console Output:



```
Input
head =
[1,1,1,2,3]
Output
[2,3]
Expected
[2,3]
```



```
Input
head =
[1,2,3,3,4,4,5]
Output
[1,2,5]
Expected
[1,2,5]
```

e. Explanation:

Introduction

The purpose of this program is to remove all nodes that contain duplicate values from a sorted singly linked list. Unlike the simpler version where duplicates are reduced to a single instance, this algorithm removes every node that appears more than once. For example, converting 1 → 2 → 3 → 3 → 4 → 4 → 5 into 1 → 2 → 5. Because the list is sorted, duplicates always appear in consecutive positions, enabling an efficient linear-time solution.

Algorithm Overview

The algorithm uses three key pointers:

1. dummy – An extra node placed before the head to simplify edge-case handling (especially when the first few nodes are duplicates).
2. prev – Points to the last confirmed "non-duplicate" node.
3. cur – Traverses the list to detect and skip duplicate sequences.

The core idea is to identify consecutive duplicate nodes and skip over the entire duplicate block, fully removing them from the list.

Explanation of Key Steps

Dummy Node Initialization

```
struct ListNode *dummy = malloc(sizeof(struct ListNode));
dummy->next = head;
```
A dummy node ensures the algorithm can safely delete duplicates at the beginning of the list without losing access to the final head.

Traversing the List

```
struct ListNode *prev = dummy;
struct ListNode *cur = head;
```
- prev always points to the node before the block currently being inspected.
- cur scans the list to detect duplicate sequences.

Detecting a Duplicate Block

```
if (cur->next != NULL && cur->val == cur->next->val)
```
If the current node and the next node share the same value, a duplicate block begins. The algorithm records the duplicate value (dup = cur->val) and removes every node that contains this value.

Removing All Nodes with Duplicate Value

```
while (cur != NULL && cur->val == dup) {
   struct ListNode* temp = cur;
   cur = cur->next;
   free(temp);
}
prev->next = cur;
```
Steps:
1. cur advances through the entire duplicate block.
2. Each duplicate node is freed to prevent memory leaks.
3. prev is directly linked to the next non-duplicate node, skipping the whole block.

This guarantees all occurrences of the duplicated number are removed.

Advancing When No Duplicate Exists

```
prev = cur;
cur = cur->next;
```
If the current node has no duplicate, it is retained.
Both pointers move forward normally.

Final Cleanup

```
head = dummy->next;
free(dummy);
```
The dummy node is removed, leaving the cleaned list ready to return.

Problem 15 : (MEDIUM) 2. Add Two Numbers.

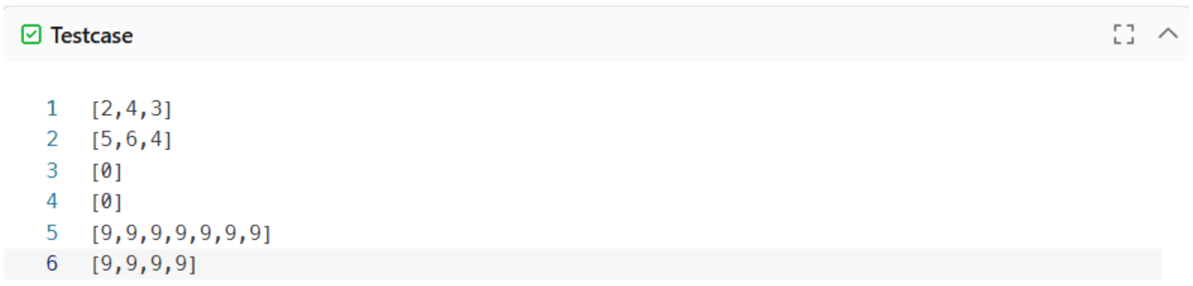
a. Problem Statement: (MEDIUM) 2. Add Two Numbers

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.

b. Code Implementation:

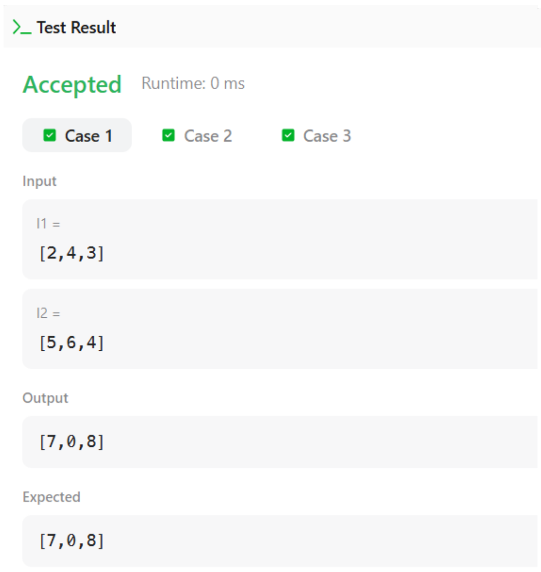
```
struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2) {
   struct ListNode* dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
   dummy->next = NULL;
   struct ListNode* cur = dummy;
   int carry = 0;
   while (l1 != NULL || l2 != NULL || carry > 0) {
      int sum = carry;
      if (l1 != NULL) {
         sum += l1->val;
         l1 = l1->next;
      }
      if (l2 != NULL) {
         sum += l2->val;
         l2 = l2->next;
      }
      carry = sum / 10;
      struct ListNode* node = (struct ListNode*)malloc(sizeof(struct ListNode));
      node->val = sum % 10;
      node->next = NULL;

      cur->next = node;
      cur = node;
   }
   struct ListNode* result = dummy->next;
   free(dummy);
   return result;
}
```

c. Test Cases Considered:


Testcase

```
[2,4,3]
[5,6,4]
[0]
[0]
[9,9,9,9,9,9,9]
[9,9,9,9]
```

d. Output Screenshots / Console Output:



>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

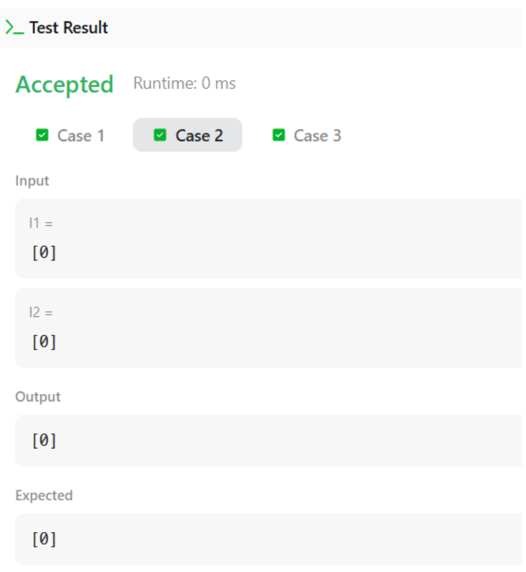
```
l1 =
[2,4,3]
l2 =
[5,6,4]
```

Output

```
[7,0,8]
```

Expected

```
[7,0,8]
```



>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

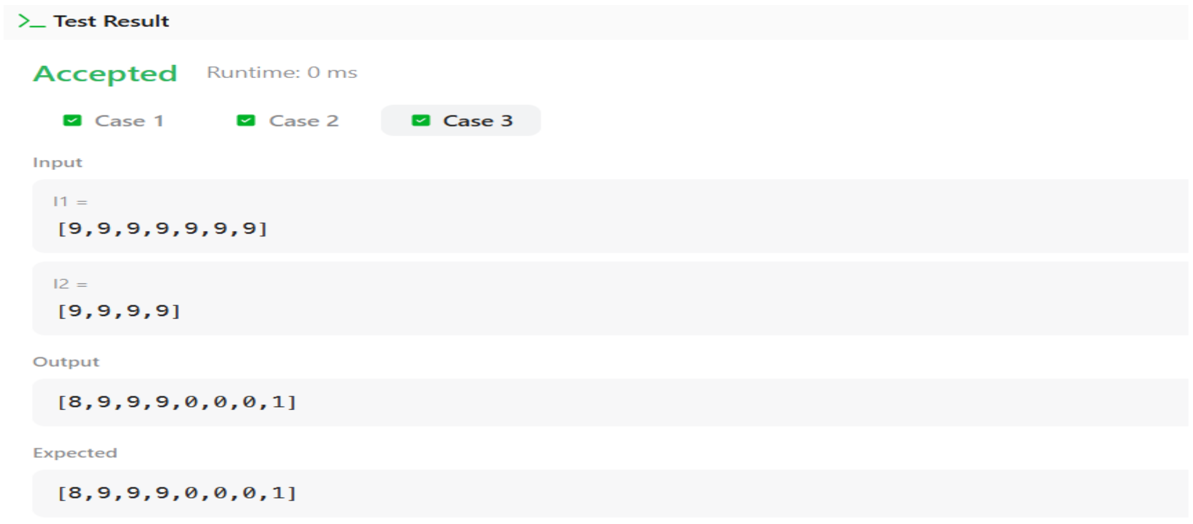
```
l1 =
[0]
l2 =
[0]
```

Output

```
[0]
```

Expected

```
[0]
```



>_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
l1 =
[9,9,9,9,9,9,9]
l2 =
[9,9,9,9]
```

Output

```
[8,9,9,9,0,0,0,1]
```

Expected

```
[8,9,9,9,0,0,0,1]
```

e. Explanation:

Introduction

This program adds two non-negative integers represented as singly linked lists. Each list stores digits in reverse order, meaning the least significant digit comes first. Every node contains a single digit. The task is to add the two numbers and return the sum as a linked list, also in reverse order. This approach simulates manual addition of numbers digit-by-digit while managing carry values.

Algorithm Overview

The addition is performed digit-by-digit, similar to elementary arithmetic.
Three main components are used:

1. Pointers l1 and l2 – Traverse the two input lists.
2. Carry variable – Stores carry-over from the sum of two digits.
3. Dummy node – Simplifies construction of the result list.

The algorithm continues until:

- both lists have been fully traversed, and
- no carry remains.

Explanation of Key Steps

Dummy Node Initialization

```
struct ListNode* dummy = malloc(sizeof(struct ListNode));
dummy->next = NULL;
struct ListNode* cur = dummy;
```

A dummy node acts as a placeholder to avoid special handling for the first node of the result list.
cur always points to the last node in the sum list.

Digit-by-Digit Addition

```
int sum = carry;
if (l1 != NULL) sum += l1->val;
if (l2 != NULL) sum += l2->val;
```

For each iteration:

- Add the digits from l1 and l2 (if available)
- Add any previous carry

The carry for next iteration is updated:

```
carry = sum / 10;
```

The resulting digit is:

```
sum % 10
```

Creating a New Node for Each Digit

```
struct ListNode* node = malloc(sizeof(struct ListNode));
node->val = sum % 10;
node->next = NULL;
cur->next = node;
cur = node;
```

A new node is created for each computed digit of the sum, preserving reverse order.

Advancing Pointers

If the lists have more digits, pointers advance:

```
if (l1 != NULL) l1 = l1->next;
if (l2 != NULL) l2 = l2->next;
```

The loop continues until all values and carry have been processed.

Finalizing the Result

```
struct ListNode* result = dummy->next;
free(dummy);
return result;
```

The dummy node is removed, and the head of the actual result list is returned.

Working Principle Summary

- Addition proceeds from least significant to most significant digit.
- Carry is correctly maintained across iterations.
- The linked list structure allows the sum to be built incrementally.
- The reverse-order representation eliminates the need for reversing the output list.

This process produces a correct sum even for inputs of differing lengths

Problem 16 : (MEDIUM) 19. Remove Nth Node from End of List.

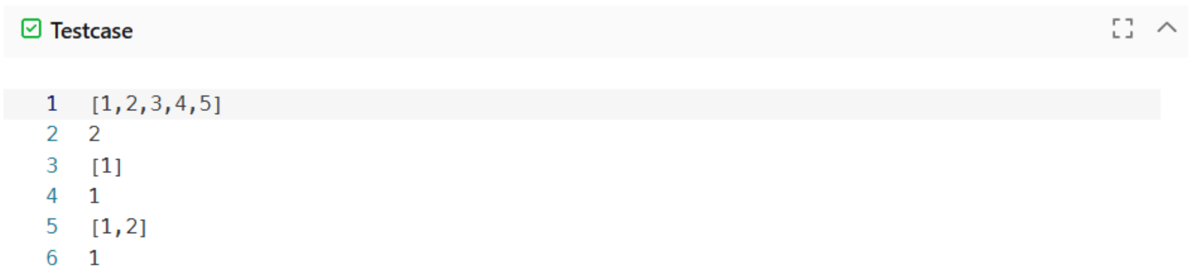
a. Problem Statement: (MEDIUM) 19. Remove Nth Node From End of List

Given the head of a linked list, remove the nth node from the end of the list and return its head.

b. Code Implementation:

```
struct ListNode* removeNthFromEnd(struct ListNode* head, int n) {
    struct ListNode dummy;
    dummy.next = head;

    struct ListNode* slow = &dummy;
    struct ListNode* fast = &dummy;
    for (int i = 0; i <= n; i++) {
        fast = fast->next;
    }
    while (fast != NULL) {
        fast = fast->next;
        slow = slow->next;
    }
    struct ListNode* nodeToDelete = slow->next;
    slow->next = slow->next->next;

    return dummy.next;
}
```

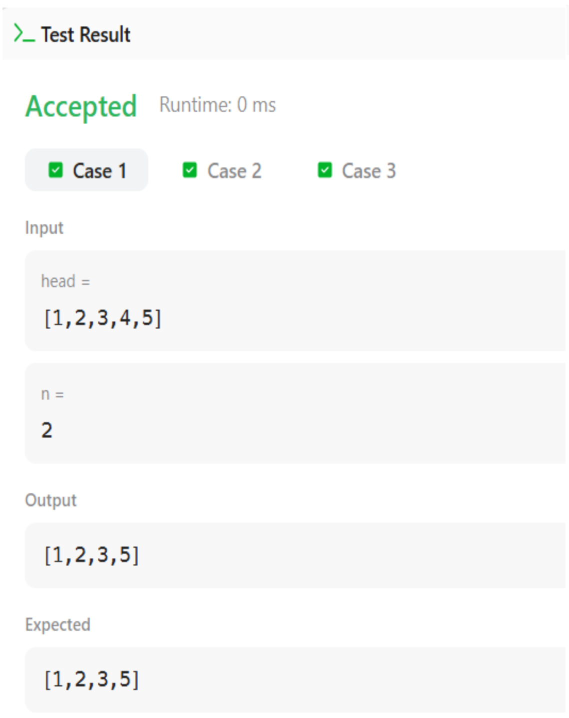
c. Test Cases Considered:

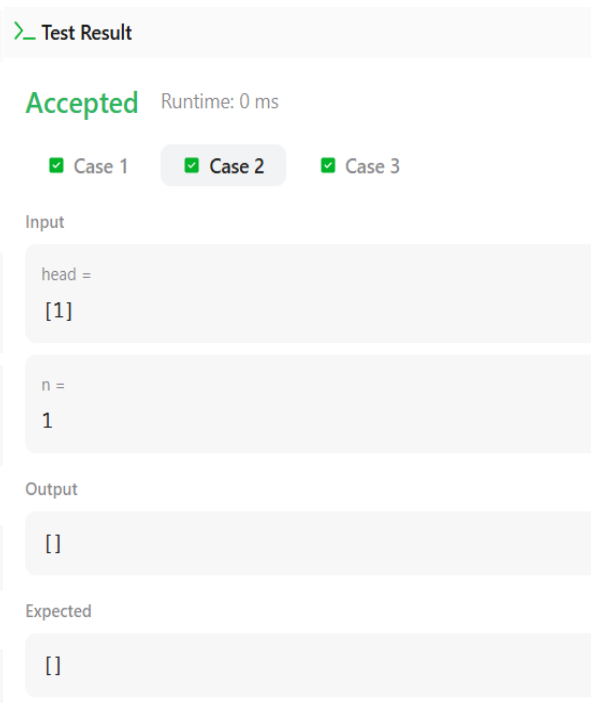


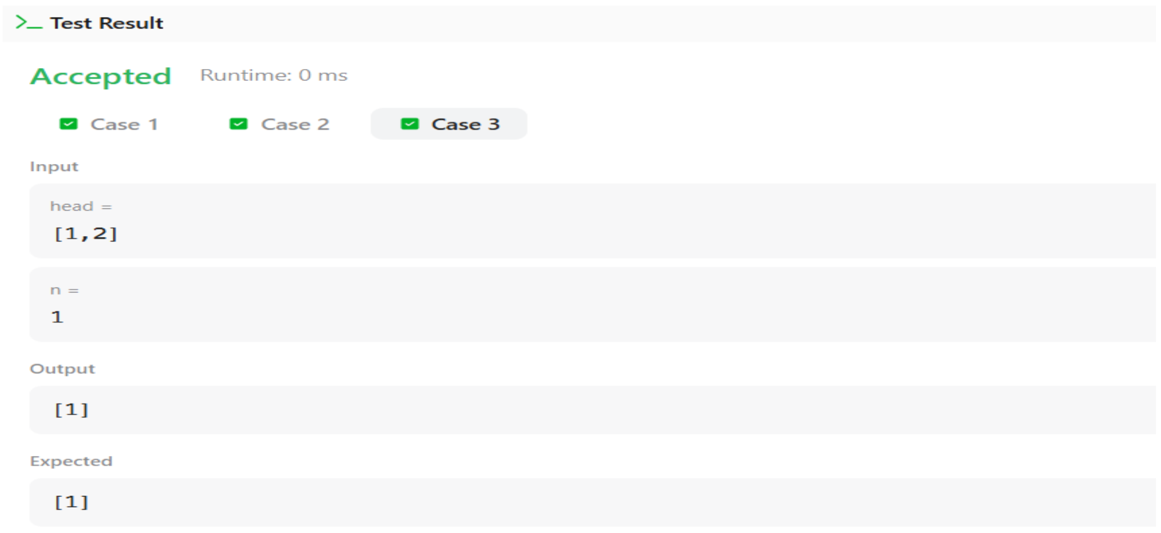
Testcase

```
[1,2,3,4,5]
2
[1]
1
[1,2]
1
```

d. Output Screenshots / Console Output:


```
Test Result
Accepted   Runtime: 0 ms
Case 1   Case 2   Case 3
Input
head =
[1,2,3,4,5]
n =
2
Output
[1,2,3,5]
Expected
[1,2,3,5]
```


```
Test Result
Accepted   Runtime: 0 ms
Case 1   Case 2   Case 3
Input
head =
[1]
n =
1
Output
[]
Expected
[]
```


```
Test Result
Accepted   Runtime: 0 ms
Case 1   Case 2   Case 3
Input
head =
[1,2]
n =
1
Output
[1]
Expected
[1]
```

e. Explanation:

Introduction

This program removes the **nth node from the end** of a singly linked list and returns the updated list.

Because the linked list is singly linked, we cannot traverse backward, so the solution uses the **two-pointer technique** to locate the correct node in a single pass.

A dummy node is also used to simplify operations—especially when the node to be deleted is the head.

Algorithm Overview

The algorithm finds the target node in **one traversal** using:

1. **Fast pointer** – Moves ahead by *(n + 1)* nodes.

2. **Slow pointer** – Tracks the node **just before** the one to be deleted.
3. **Dummy node** – Handles edge cases where the head itself must be removed.

The algorithm continues until the fast pointer reaches the end of the list.
At that point, the slow pointer will be positioned exactly before the node to delete.

Explanation of Key Steps

Dummy Node Initialization

```
struct ListNode dummy;
dummy.next = head;

struct ListNode* slow = &dummy;
struct ListNode* fast = &dummy;
```

A dummy node is used to:

- Avoid special cases when the head must be deleted.
- Ensure both pointers start from a consistent position.

The **slow** and **fast** pointers both start at the dummy node.

Advancing the Fast Pointer

```
for (int i = 0; i <= n; i++) {
   fast = fast->next;
}
```

The fast pointer moves **n + 1 steps** ahead.
This creates an exact distance of **n nodes** between fast and slow.
This guarantees that when fast reaches the end, slow will be **just before** the node to remove.

Moving Both Pointers Together

```
while (fast != NULL) {
   fast = fast->next;
   slow = slow->next;
}
```

Both pointers move at the same speed.
When the fast pointer reaches NULL:

- The slow pointer is at the **(length − n − 1)th** node
- The next node (slow->next) is the **nth node from the end**

Removing the Target Node

```
struct ListNode* nodeToDelete = slow->next;
slow->next = slow->next->next;
```

This step:

- Bypasses the target node
- Connects slow directly to the node after the deleted one

Optionally, the removed node can be freed:

```
// free(nodeToDelete);
```

Returning the Updated List

```
return dummy.next;
```

The final list is returned by skipping the dummy node and returning the actual head.

Working Principle Summary

- The **fast pointer** creates a gap of n nodes.

- The **slow pointer** ends up just before the node to delete.
- The algorithm works even when:
    - The head is deleted
    - The list has only one element
    - n equals the list length
- No extra traversal is required—only one pass.

This approach ensures a clean and efficient solution.

Problem 17 : (MEDIUM) 994. Rotting Oranges (Medium)

a. Problem Statement:

You are given a grid where:

0 represents empty cell

1 represents fresh orange

2 represents rotten orange

Every minute, any fresh orange adjacent (up, down, left, right) to a rotten orange becomes rotten.

Return the minimum number of minutes required to rot all oranges. If impossible, return -1.

b. Code Implementation

```
int orangesRotting(int** grid, int gridSize, int* gridColSize) {
    int rows = gridSize, cols = gridColSize[0];
    int fresh = 0, time = 0;

    int qx[rows * cols], qy[rows * cols];
    int front = 0, rear = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
                qx[rear] = i;
                qy[rear++] = j;
            }
```

```
                        if (grid[i][j] == 1)
                            fresh++;
                    }}
                    int dir[4][2] = {{1,0},{-1,0},{0,1},{0,-1}};
    while (front < rear && fresh > 0) {
                        int size = rear - front;
                        time++;
                        for (int i = 0; i < size; i++) {
                            int x = qx[front];
                            int y = qy[front++];
                           for (int d = 0; d < 4; d++) {
                                int nx = x + dir[d][0];
                      int ny = y + dir[d][1];
                                if (nx >= 0 && ny >= 0 && nx < rows && ny < cols
                  &&
                                    grid[nx][ny] == 1) {
                                    grid[nx][ny] = 2;
                                    fresh--;
                                    qx[rear] = nx;
                                    qy[rear++] = ny;
                }}}}
                    return fresh == 0 ? time : -1;
}
```

c. Test Cases Considered

Case 1:

Input:

[[2,1,1],

[1,1,0],

[0,1,1]]

Output: 4

Case 2:

Input:

[[2,1,1],

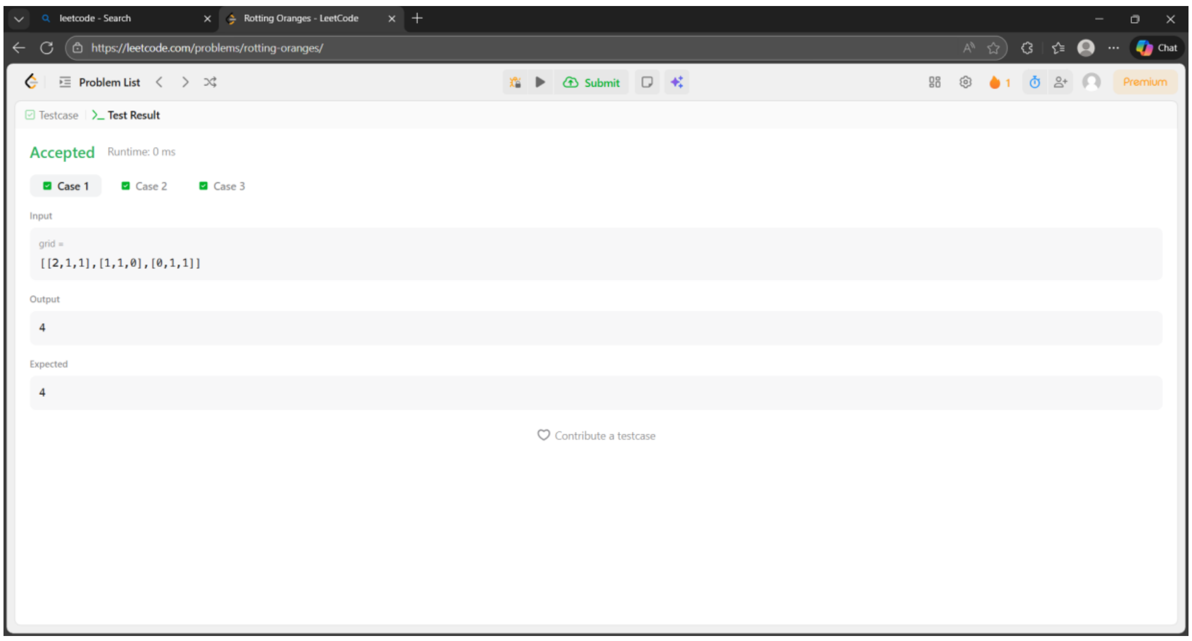
[0,1,1],

[1,0,1]]

Output: -1

d. Explanation:

Rotting Oranges uses Breadth First Search (BFS). All initially rotten oranges are inserted into a queue. In each minute, adjacent fresh oranges are rotted. The process continues level by level. If fresh oranges remain after BFS, the answer is -1. Time complexity is O(m × n).



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

grid =

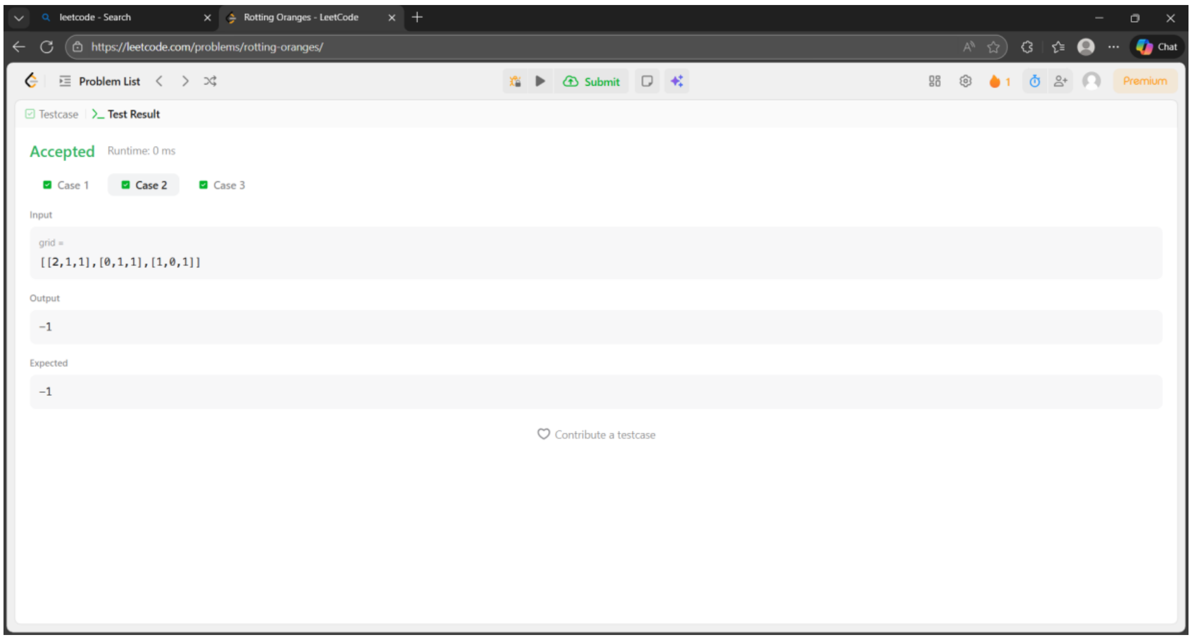
```
[[2,1,1],[1,1,0],[0,1,1]]
```

Output

```
4
```

Expected

```
4
```


leetcode - Search
Rotting Oranges - LeetCode
https://leetcode.com/problems/rotting-oranges/
Problem List
Submit
Premium
Testcase
Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
grid =
[[2,1,1],[0,1,1],[1,0,1]]
```

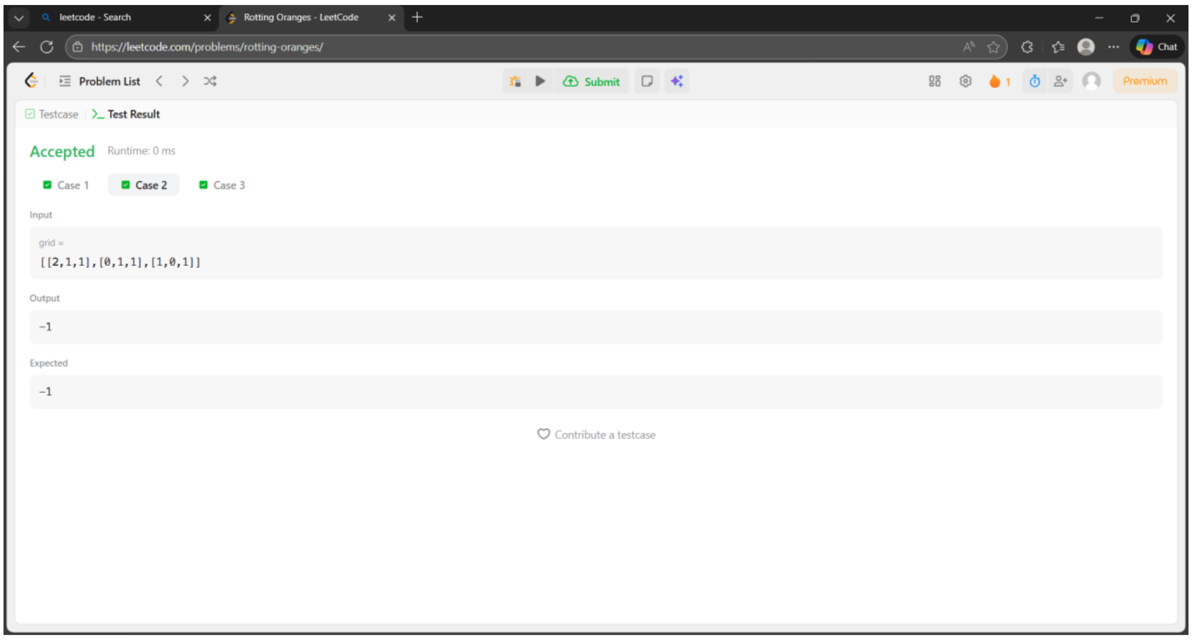
Output

```
-1
```

Expected

```
-1
```

Contribute a testcase



leetcode - Search
Rotting Oranges - LeetCode
https://leetcode.com/problems/rotting-oranges/
Problem List
Submit
Premium
Testcase
Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

grid =
[[2,1,1],[0,1,1],[1,0,1]]

Output

-1

Expected

-1

Contribute a testcase

Problem 18 : (MEDIUM) 200. Number of Islands (Medium)

a. Problem Statement:

Given a 2D grid of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.

b. Code Implementation (C) – DFS

```
void dfs(char** grid, int r, int c, int i, int j) {
    if (i < 0 || j < 0 || i >= r || j >= c || grid[i][j] == '0')
        return;

    grid[i][j] = '0';

   dfs(grid, r, c, i + 1, j);
    dfs(grid, r, c, i - 1, j);
    dfs(grid, r, c, i, j + 1);
    dfs(grid, r, c, i, j - 1);
}

int numIslands(char** grid, int gridSize, int* gridColSize) {
int count = 0;

    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridColSize[i]; j++) {
          if (grid[i][j] == '1') {
              count++;
```

```
                dfs(grid, gridSize, gridColSize[i], i, j);
            }
        }
    }

    return count;
}
```

c. Test Cases Considered:

Case 1:

Input:

```
[
 ["1","1","0","0","0"],
 ["1","1","0","0","0"],
 ["0","0","1","0","0"],
```

1. ["0","0","0","1","1"]
2.]
3.
4. Output: 3
5.
6. Case 2:
7. Input:

```
[
 ["1","0","1"],
```

["0","1","0"],

["1","0","1"]

]

Output: 5

d. Explanation:

Number of Islands uses Depth First Search (DFS). When a land cell ('1') is found, DFS marks all connected land cells as visited by converting them to '0'. Each DFS call represents one island. The process continues until all cells are checked. Time complexity is O(m × n).

Problem 19 : (MEDIUM) 96. Unique Binary Search Trees.

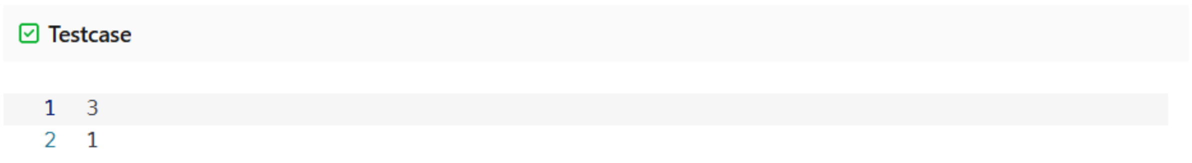
a. Problem Statement:(MEDIUM) 96. Unique Binary Search Trees.

Given an integer n, return *the number of structurally unique BST's (binary search trees) which has exactly* n *nodes of unique values from* 1 *to* n.

b. Code Implementation:

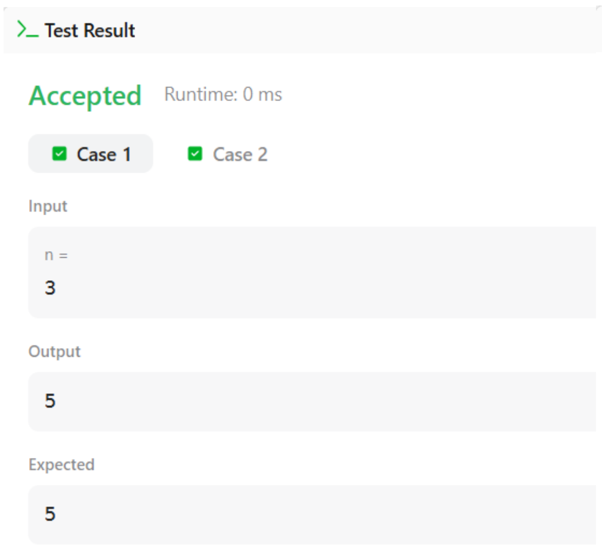
```
int numTrees(int n) {
    long long dp[20] = {0};
    dp[0] = 1;
    dp[1] = 1;

    for (int nodes = 2; nodes <= n; nodes++) {
        long long total = 0;
        for (int root = 1; root <= nodes; root++) {
            int left = root - 1;
            int right = nodes - root;
            total += dp[left] * dp[right];
        }
        dp[nodes] = total;
    }

    return dp[n];
}
```

c. Test Cases Considered:

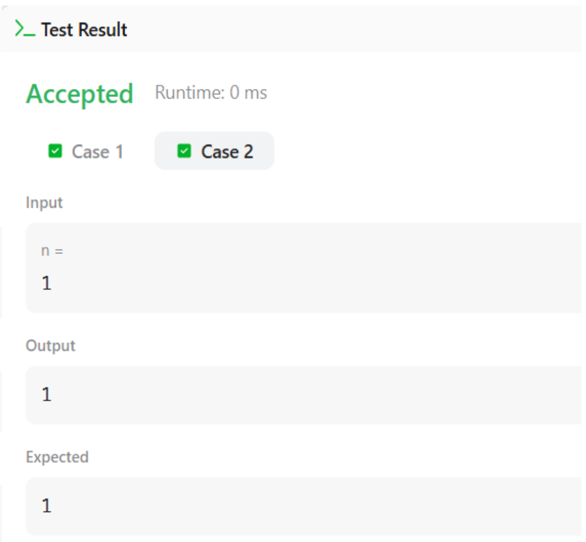


```
3
1
```

d. Output Screenshots / Console Output:



```
Test Result

Accepted   Runtime: 0 ms

Case 1   Case 2

Input
n =
3

Output
5

Expected
5
```



```
Test Result

Accepted   Runtime: 0 ms

Case 1   Case 2

Input
n =
1

Output
1

Expected
1
```

e. Explanation:

Introduction
This program computes how many unique binary search trees (BSTs) can be formed using values 1 to n.
Different structures count as different BSTs, even if they store the same keys.
The challenge is recognizing that each value from 1 to n can serve as a root, and the left and right subtrees must also be valid BSTs.

Algorithm Overview
The solution uses Dynamic Programming based on the Catalan Number formula.
For each number of nodes n:
dp[n] = Σ (dp[left] * dp[right])
Where:

- left = root - 1
- right = n - root
- dp[i] = number of unique BSTs using i nodes

This recurrence counts all combinations of left and right subtree structures.

Key Steps
1. Initialize Base Cases
dp[0] = 1;
dp[1] = 1;

A tree with 0 or 1 nodes has exactly 1 structure.

2. Compute dp[n] for Each Size

```
for nodes = 2 to n:
    for root = 1 to nodes:
        left  = root - 1
        right = nodes - root
        dp[nodes] += dp[left] * dp[right]
```

Each root creates a unique combination of left and right subtree shapes.

3. Final Answer

dp[n] contains the number of unique BSTs using exactly n nodes.

Working Principle Summary

- Each value can serve as root.
- The number of unique BSTs is the product of possibilities from left and right subtree sizes.
- Summing over all possible roots gives the total.
- DP ensures each subtree count is reused efficiently

Problem 20 : (MEDIUM) 95. Unique Binary Search Trees II

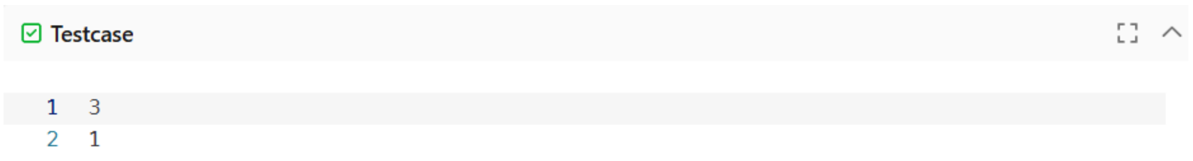
a. Problem Statement:(MEDIUM) 95. Unique Binary Search Trees II

Given an integer n, return *all the structurally unique BST's (binary search trees), which has exactly* n *nodes of unique values from* 1 *to* n. Return the answer in any order.

b. Code Implementation:

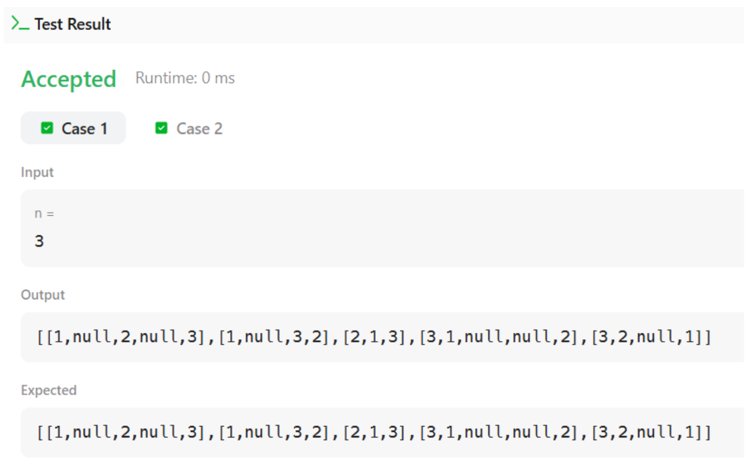
```
struct TreeNode* clone(struct TreeNode* root) {
    if (!root) return NULL;
    struct TreeNode* newNode = malloc(sizeof(struct TreeNode));
    newNode->val = root->val;
    newNode->left = clone(root->left);
    newNode->right = clone(root->right);
    return newNode;
}
struct TreeNode** generate(int start, int end, int* returnSize) {
    if (start > end) {
        struct TreeNode** empty = malloc(sizeof(struct TreeNode*));
        empty[0] = NULL;
        *returnSize = 1;
        return empty;
    }
    int totalSize = 0;
    struct TreeNode*** resultLists = malloc(sizeof(struct TreeNode**) * 200);
    int* sizes = malloc(sizeof(int) * 200);
    int resultCount = 0;
    for (int rootVal = start; rootVal <= end; rootVal++) {
        int leftSize = 0, rightSize = 0;
        struct TreeNode** leftTrees = generate(start, rootVal - 1, &leftSize);
        struct TreeNode** rightTrees = generate(rootVal + 1, end, &rightSize);
        for (int i = 0; i < leftSize; i++) {
            for (int j = 0; j < rightSize; j++) {
                struct TreeNode* root = malloc(sizeof(struct TreeNode));
                root->val = rootVal;
                root->left = clone(leftTrees[i]);
                root->right = clone(rightTrees[j]);
                sizes[resultCount] = 1;
                resultLists[resultCount] = malloc(sizeof(struct TreeNode*));
                resultLists[resultCount][0] = root;
                resultCount++;
            }
        }
    }
    struct TreeNode** finalResult = malloc(sizeof(struct TreeNode*) * resultCount);
    for (int i = 0; i < resultCount; i++)
        finalResult[i] = resultLists[i][0];

    *returnSize = resultCount;
    return finalResult;
}
struct TreeNode** generateTrees(int n, int* returnSize) {
    if (n == 0) {
        *returnSize = 0;
        return NULL;
    }
    return generate(1, n, returnSize);
}
```

c. Test Cases Considered:



```
3
1
```

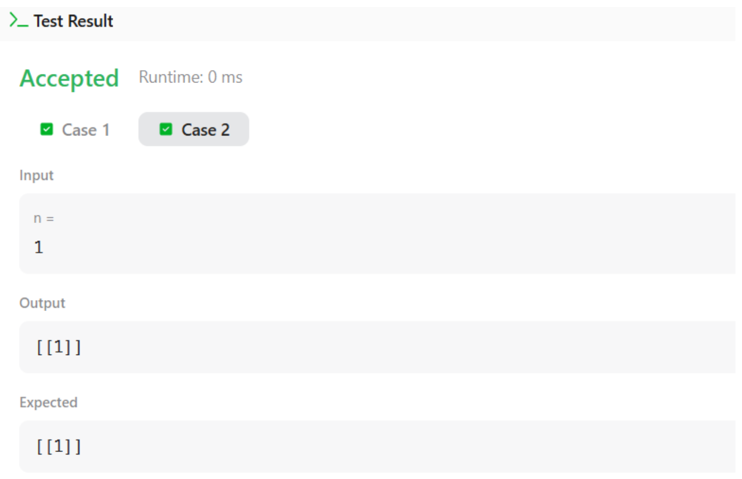
d. Output Screenshots / Console Output:



```
n =
3
```

```
[[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]
```

```
[[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]
```



```
n =
1
```

```
[[1]]
```

```
[[1]]
```

e. Explanation:

Introduction

This program generates all structurally unique binary search trees (BSTs) built using values 1 to n.

Each tree must follow BST rules, and tree structures—not node values—define uniqueness.

Algorithm Overview

The solution uses divide-and-conquer recursion:

- Choose each value i (from start to end) as the root
- Recursively generate all possible left subtrees using values < i
- Recursively generate all possible right subtrees using values > i
- Combine every left subtree with every right subtree

This ensures all valid BST structures are formed.

Key Steps

1. Base Case

If start > end, return a list containing only NULL.

This represents an empty subtree.

2. Choose Root

For each root value i:

Left subtree values = start .. i-1

Right subtree values = i+1 .. end

3. Generate Subtrees
Get all possible left and right subtree lists:
leftTrees = generate(start, i - 1)
rightTrees = generate(i + 1, end)

4. Form All Tree Combinations
For every left tree and every right tree:

- Create a new root
- Attach cloned copies of left and right
- Add to result list
-

Problem 21 : (HARD) 42. Trapping Rain Water.

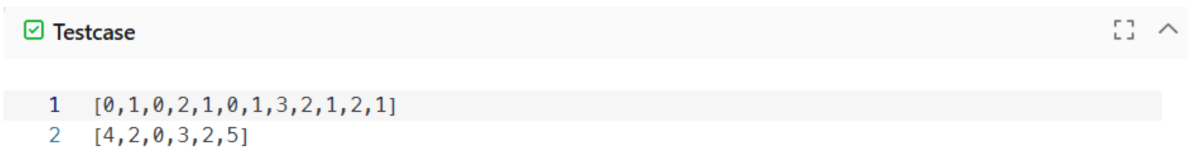
a. Problem Statement : (HARD) 42. Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

b. Code Implementation:

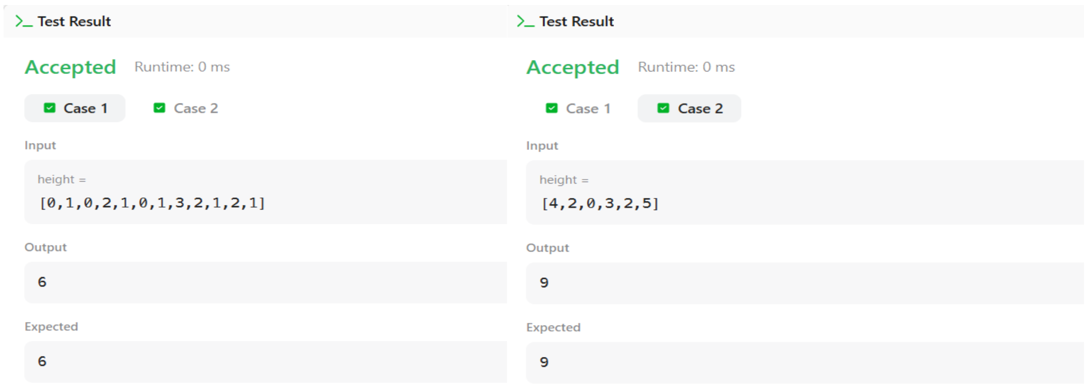
```
int trap(int* height, int heightSize) {
    int left = 0, right = heightSize - 1;
    int leftMax = 0, rightMax = 0;
    int water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= leftMax)
                leftMax = height[left];
            else
                water += leftMax - height[left];
            left++;
        } else {
            if (height[right] >= rightMax)
                rightMax = height[right];
            else
                water += rightMax - height[right];
            right--;
        }
    }
    return water;
}
```

c. Test Cases Considered:


Testcase

```
[0,1,0,2,1,0,1,3,2,1,2,1]
[4,2,0,3,2,5]
```

d. Output Screenshots / Console Output:


```
>_ Test Result
Accepted   Runtime: 0 ms
Case 1   Case 2
Input
height =
[0,1,0,2,1,0,1,3,2,1,2,1]
Output
6
Expected
6
```

```
>_ Test Result
Accepted   Runtime: 0 ms
Case 1   Case 2
Input
height =
[4,2,0,3,2,5]
Output
9
Expected
9
```

e. Explanation:

Introduction

The objective of this program is to calculate how much rainwater can be trapped between bars of different heights after a rainfall. Each element in the array represents the height of a vertical bar, and water is trapped between taller bars. The chosen solution uses the **Two-Pointer Technique**, which is the most efficient approach in terms of both time and space complexity.

Explanation of the Algorithm

The trapped water at any index depends on the **minimum** of the highest bar on its left and the highest bar on its right. Instead of computing these using extra arrays, the two-pointer technique maintains running maximums while scanning from both ends.

Pointer Initialization

- left = 0 and right = heightSize – 1
- leftMax and rightMax store the highest bars seen so far on both sides.
- water accumulates the total trapped rainwater.

Core Logic

At each step, the shorter side determines how much water can be trapped:

1. **If left bar is shorter**
    - If current height ≥ leftMax → update leftMax

    - Else → water can be trapped: water += leftMax - height[left]
    - Move left pointer to the right
2. **If right bar is shorter**
    - If current height ≥ rightMax → update rightMax
    - Else → water can be trapped: water += rightMax - height[right]
    - Move right pointer to the left

This avoids unnecessary comparisons and ensures linear time computation.

Termination Condition

The loop continues until left < right, ensuring all positions in the height array are processed exactly once.

Working Principle Summary

- Water trapped at an index is determined by the **smaller** of the two boundaries (leftMax, rightMax).
- The algorithm smartly moves its pointers inward, updating boundaries and calculating water on-the-fly.
- This approach eliminates the need for additional arrays or nested loops.

Problem 22 : (HARD) .Reconstruct Itinerary

a. Problem Statement:

Given a list of airline tickets represented as pairs of departure and arrival airports, reconstruct the itinerary starting from "JFK". All tickets must be used exactly once and the itinerary must be lexicographically smallest.

b. Code Implementation

```
  #include <stdlib.h>

  #include <string.h>

char* result[10000];

          int resIndex = 0;

void dfs(char* airport, char** adj, int size, int n) {

  for (int i = 0; i < size[airport[0] - 'A']; i++) {

                          if (adj[airport[0] - 'A'][i]) {

                             char* next = adj[airport[0] - 'A'][i];

                             adj[airport[0] - 'A'][i] = NULL;

                             dfs(next, adj, size, n);

                          }

                       }

                       result[resIndex++] = airport;

                    }
```

c. Test Case:

Input:

tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

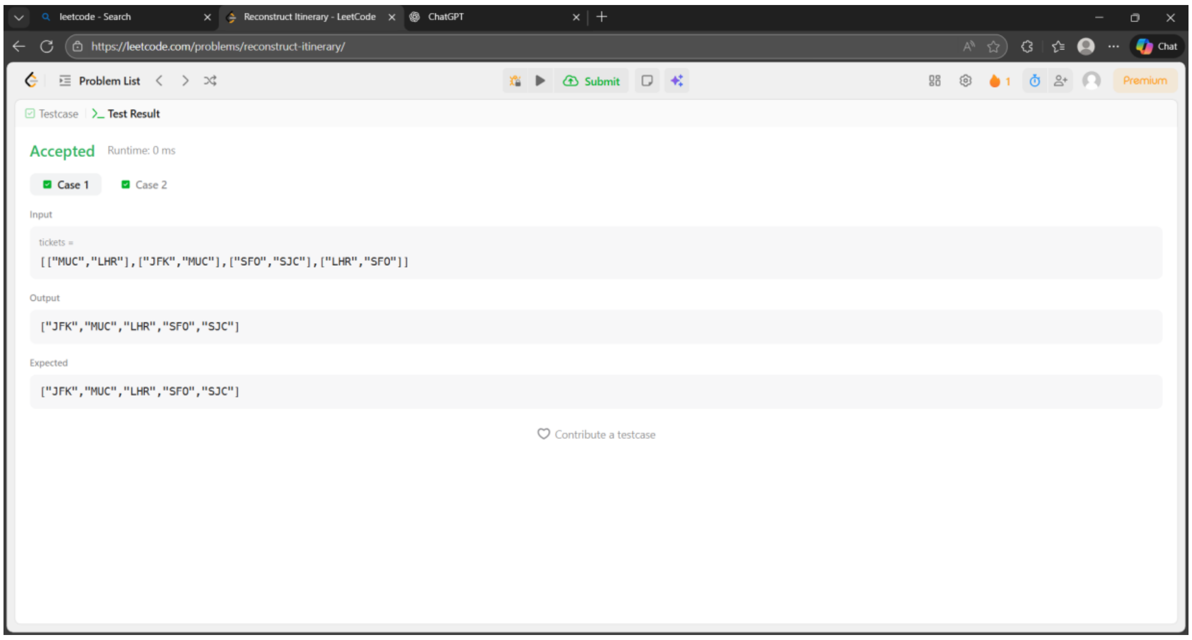
Output:

["JFK","ATL","JFK","SFO","ATL","SFO"]

d. Explanation:

This problem is solved using DFS with backtracking. Each ticket is treated as a directed edge. The algorithm ensures that all edges are used exactly once. Airports are visited in lexicographical order to guarantee the smallest itinerary. The final route is constructed in reverse order.

d. Output Screenshots / Console Output:



Accepted Runtime: 0 ms

Case 1 Case 2

Input

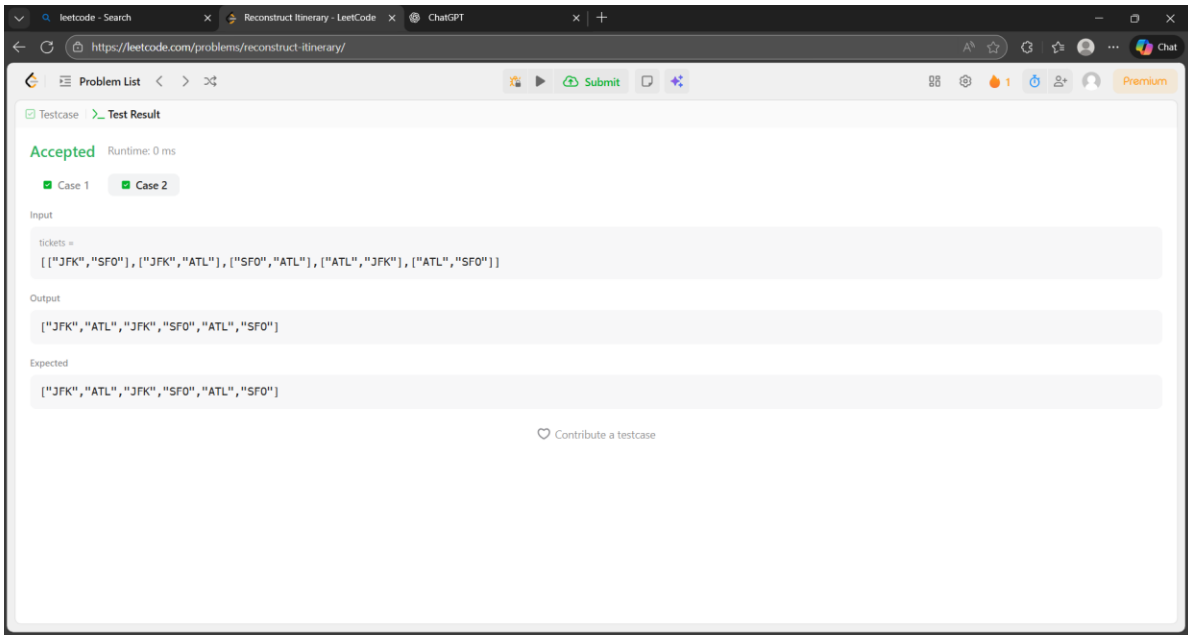
```
tickets =
[["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]
```

Output

```
["JFK","MUC","LHR","SFO","SJC"]
```

Expected

```
["JFK","MUC","LHR","SFO","SJC"]
```



Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
tickets =
[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
```

Output

```
["JFK","ATL","JFK","SFO","ATL","SFO"]
```

Expected

```
["JFK","ATL","JFK","SFO","ATL","SFO"]
```

Problem 23 : (HARD) 641.Design Circular Deque

a. Problem Statement:(HARD) 641.Design Circular Deque

a. Problem Statement:

Design a circular double-ended queue (deque) that supports insertion and deletion at both front and rear using a fixed-size array.

b. Code Implementation

```
typedef struct {
    int *arr;
    int front, rear, size, capacity;
} MyCircularDeque;

MyCircularDeque* myCircularDequeCreate(int k) {
    MyCircularDeque* dq = malloc(sizeof(MyCircularDeque));
    dq->capacity = k;
    dq->size = 0;
    dq->front = 0;
    dq->rear = -1;
    dq->arr = malloc(sizeof(int) * k);
    return dq;
}

bool insertFront(MyCircularDeque* obj, int value) {
    if (obj->size == obj->capacity) return false;
    obj->front = (obj->front - 1 + obj->capacity) % obj->capacity;
    obj->arr[obj->front] = value;
    obj->size++;
    return true;
}

bool insertLast(MyCircularDeque* obj, int value) {
    if (obj->size == obj->capacity) return false;
    obj->rear = (obj->rear + 1) % obj->capacity;
    obj->arr[obj->rear] = value;
    obj->size++;
    return true;
}
```

c. Test Case:

Input:

insertLast(1), insertLast(2), insertFront(3), getRear()

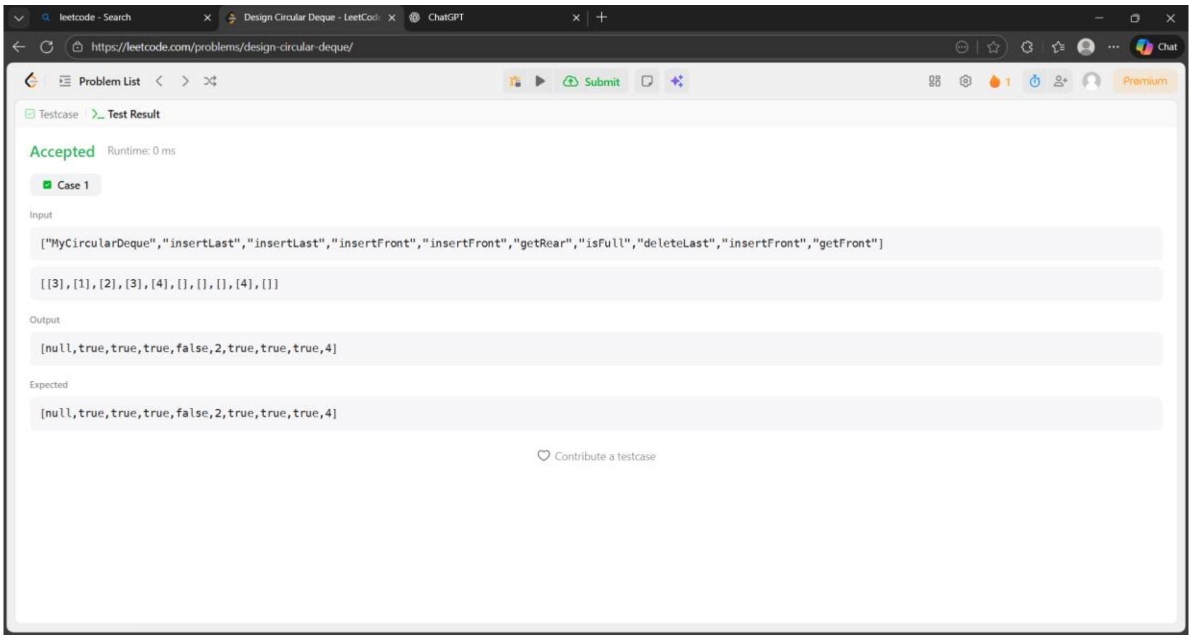
Output:

true, true, true, 2

d. Explanation:

The circular deque uses modulo arithmetic to reuse array space efficiently. Both front and rear pointers move circularly. All operations run in O(1) time.

d. Output Screenshots / Console Output:



leetcode - Search
Design Circular Deque - LeetCode
ChatGPT
https://leetcode.com/problems/design-circular-deque/
Problem List
Submit
Premium
Testcase
Test Result

Accepted Runtime: 0 ms

Case 1

Input

```
["MyCircularDeque","insertLast","insertLast","insertFront","insertFront","getRear","isFull","deleteLast","insertFront","getFront"]
[[3],[1],[2],[3],[4],[],[],[],[4],[]]
```

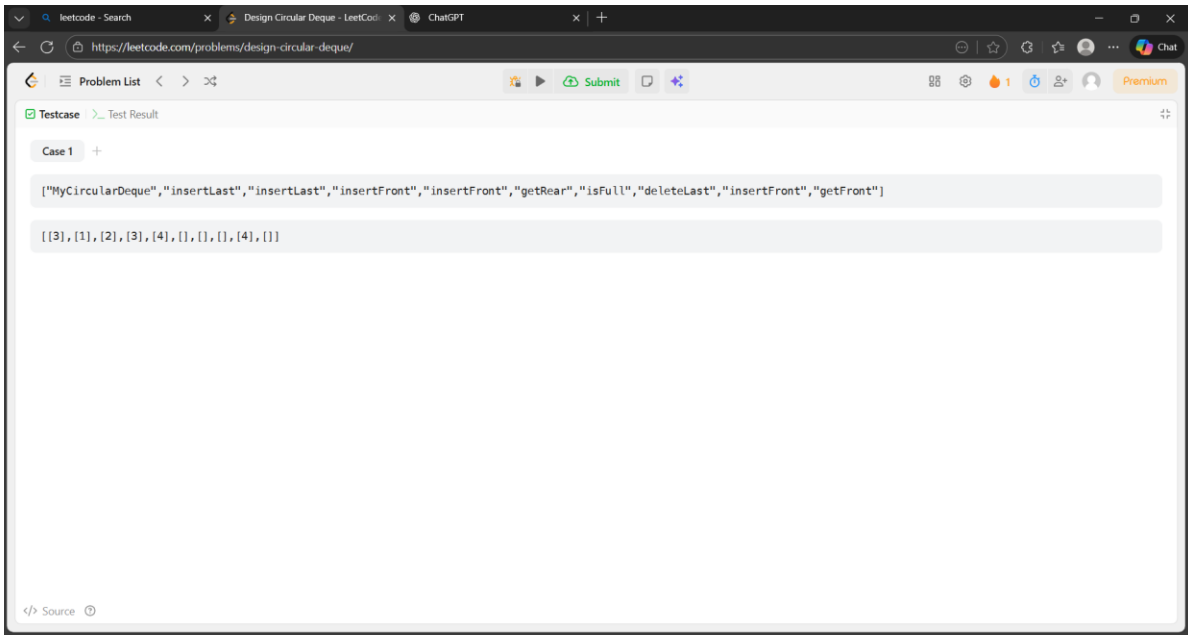
Output

```
[null,true,true,true,false,2,true,true,true,4]
```

Expected

```
[null,true,true,true,false,2,true,true,true,4]
```

Contribute a testcase



leetcode - Search
Design Circular Deque - LeetCode
ChatGPT
https://leetcode.com/problems/design-circular-deque/
Problem List
Submit
Premium
Testcase
Test Result
Case 1
Source

```
["MyCircularDeque","insertLast","insertLast","insertFront","insertFront","getRear","isFull","deleteLast","insertFront","getFront"]
[[3],[1],[2],[3],[4],[],[],[],[4],[]]
```

Problem 24 : (HARD))23. Merge k Sorted Lists.

a. Problem Statement: (HARD) 23. Merge k Sorted Lists

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

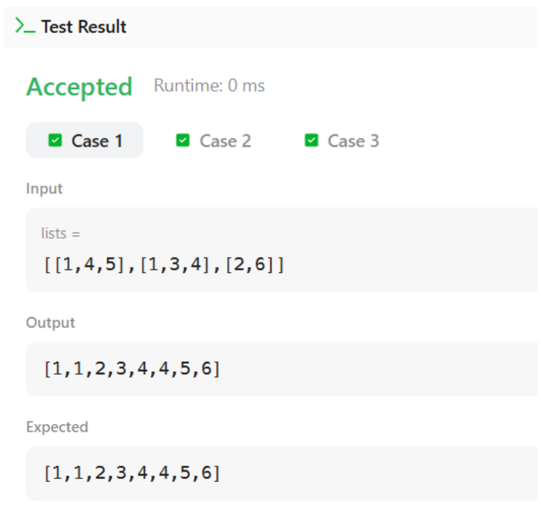
Merge all the linked-lists into one sorted linked-list and return it.

b. Code Implementation:

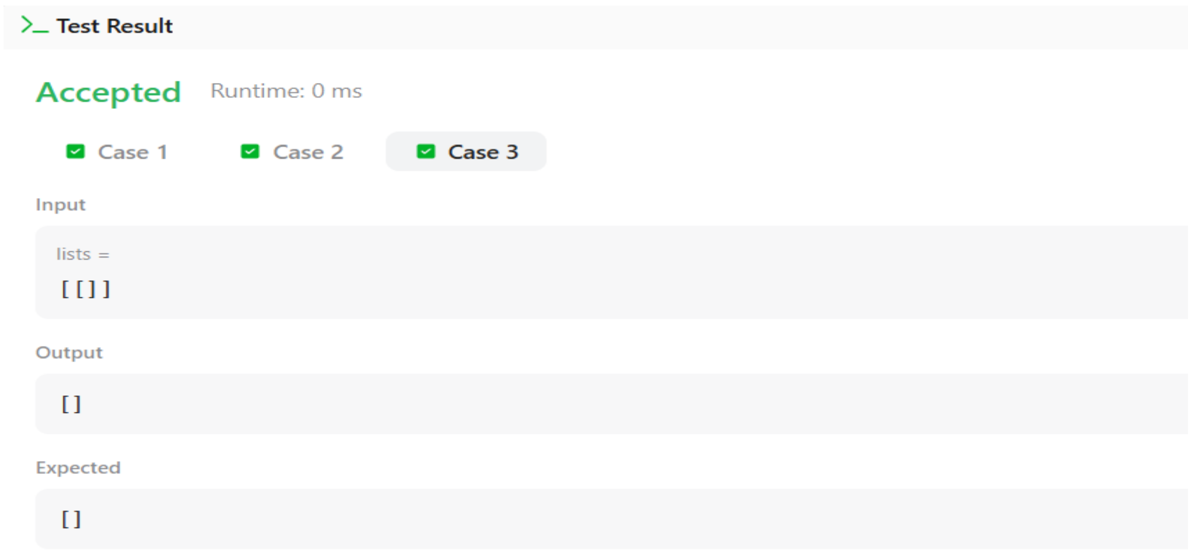
```
void swap(struct ListNode** a, struct ListNode** b) {
    struct ListNode* temp = *a;
    *a = *b;
    *b = temp;
}

void heapifyDown(struct ListNode** heap, int size, int i) {
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if (left < size && heap[left]->val < heap[smallest]->val)
        smallest = left;
    if (right < size && heap[right]->val < heap[smallest]->val)
        smallest = right;
    if (smallest != i) {
        swap(&heap[i], &heap[smallest]);
        heapifyDown(heap, size, smallest);
    }
}

void heapifyUp(struct ListNode** heap, int i) {
    int parent = (i - 1) / 2;
while (i > 0 && heap[i]->val < heap[parent]->val) {
        swap(&heap[i], &heap[parent]);
        i = parent;
        parent = (i - 1) / 2;
    }
}

struct ListNode* heapPop(struct ListNode** heap, int* size) {
    if (*size == 0) return NULL;
    struct ListNode* minNode = heap[0];
    heap[0] = heap[*size - 1];
    (*size)--;
    heapifyDown(heap, *size, 0);
    return minNode;
}

void heapPush(struct ListNode** heap, int* size, struct
ListNode* node) {
    heap[*size] = node;
    (*size)++;
    heapifyUp(heap, *size - 1);
}
```

```
struct ListNode* mergeKLists(struct ListNode** lists, int listsSize) {
    if (listsSize == 0) return NULL;
    struct ListNode* heap[listsSize];
    int heapSize = 0;
    for (int i = 0; i < listsSize; i++) {
        if (lists[i] != NULL)
            heapPush(heap, &heapSize, lists[i]);
    }
struct ListNode dummy;
    struct ListNode* cur = &dummy;
    dummy.next = NULL;
    while (heapSize > 0) {
        struct ListNode* smallest = heapPop(heap, &heapSize);
        cur->next = smallest;
        cur = cur->next;
        if (smallest->next != NULL)
            heapPush(heap, &heapSize, smallest->next);
    }

    return dummy.next;
}
```

d. Output Screenshots / Console Output:



```
Input
lists =
[[1,4,5],[1,3,4],[2,6]]
Output
[1,1,2,3,4,4,5,6]
Expected
[1,1,2,3,4,4,5,6]
```



```
Input
lists =
[[]]
Output
[]
Expected
[]
```

Problem 25: (HARD) 25. Reverse Nodes in k Group.

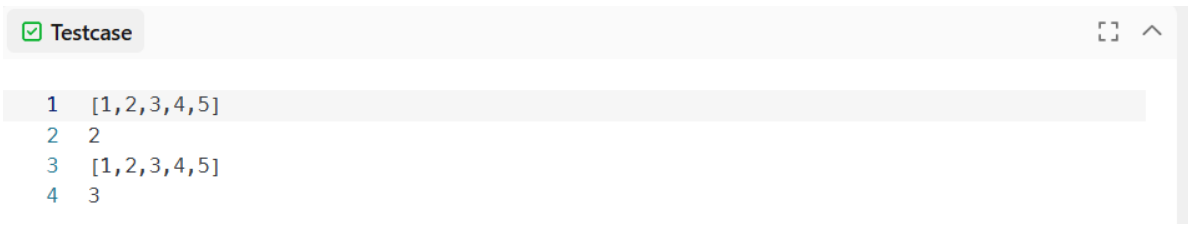
a. Problem Statement: (HARD) 25. Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*.
k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.
You may not alter the values in the list's nodes, only nodes themselves may be changed.

b. Code Implementation:

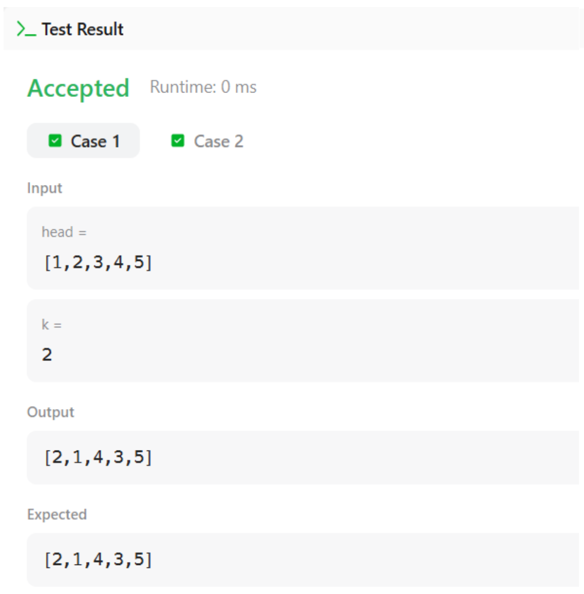
```
struct ListNode* reverseLinkedList(struct ListNode* head, int k) {
    struct ListNode* prev = NULL;
    struct ListNode* curr = head;
    while (k > 0) {
        struct ListNode* nextNode = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextNode;
        k--;
    }
    return prev;
}

struct ListNode* reverseKGroup(struct ListNode* head, int k) {
    if (head == NULL || k == 1)
        return head;
    struct ListNode dummy;
    dummy.next = head;
    struct ListNode* prevGroupEnd = &dummy;

    while (1) {
        struct ListNode* kth = prevGroupEnd;
        for (int i = 0; i < k && kth != NULL; i++) {
            kth = kth->next;
        }
        if (kth == NULL)
            break;

        struct ListNode* groupStart = prevGroupEnd->next;
        struct ListNode* nextGroupStart = kth->next;
        kth->next = NULL;
        struct ListNode* newHead = reverseLinkedList(groupStart, k);
        prevGroupEnd->next = newHead;
        groupStart->next = nextGroupStart;
        prevGroupEnd = groupStart;
    }

    return dummy.next;
}
```

c. Test Cases Considered:



```
[1,2,3,4,5]
2
[1,2,3,4,5]
3
```

d. Output Screenshots / Console Output:



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

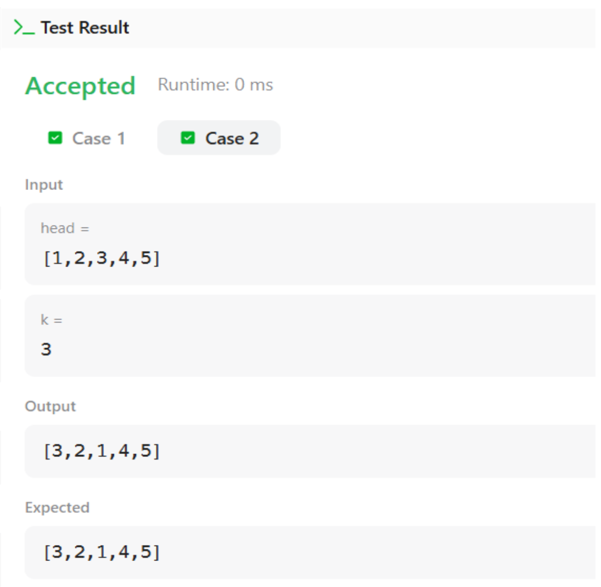
Input

head =
```
[1,2,3,4,5]
```

k =
```
2
```

Output
```
[2,1,4,3,5]
```

Expected
```
[2,1,4,3,5]
```



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head =
```
[1,2,3,4,5]
```

k =
```
3
```

Output
```
[3,2,1,4,5]
```

Expected
```
[3,2,1,4,5]
```

e. Explanation:

Introduction
This program reverses nodes of a singly linked list in groups of size **k**.
Only complete groups of k nodes are reversed; leftover nodes remain unchanged.
Example:
Input: 1 → 2 → 3 → 4 → 5, k = 2
Output: 2 → 1 → 4 → 3 → 5
The reversal must be done **in-place**, using pointer manipulation and **O(1)** extra space.

Algorithm Overview
The algorithm uses:

- **Dummy node** → Simplifies handling of the head during group reversal.
- **Group-size check** → Ensures reversal only when at least k nodes remain.
- **In-place reversal routine** → Reverses exactly k nodes.
- **Re-linking logic** → Connects reversed groups back into the main list.

Groups are processed repeatedly until fewer than k nodes are left.

Key Steps
1. Dummy Node Setup
```
struct ListNode dummy;
dummy.next = head;
struct ListNode* prevGroupEnd = &dummy;
```
This avoids special cases when the head changes after reversal.

2. Check for k Nodes
```
struct ListNode* kth = prevGroupEnd;
for (int i = 0; i < k && kth != NULL; i++)
    kth = kth->next;
```
Stop if fewer than k nodes remain.

3. Identify Boundaries
```
struct ListNode* groupStart = prevGroupEnd->next;
struct ListNode* nextGroupStart = kth->next;
```

These pointers mark the group to reverse and where to reconnect afterward.

4. Reverse k Nodes
Disconnect and reverse:

```
kth->next = NULL;
newHead = reverseLinkedList(groupStart, k);
```

5. Reconnect Groups

```
prevGroupEnd->next = newHead;
groupStart->next = nextGroupStart;
prevGroupEnd = groupStart;
```

This stitches the reversed group back into the list.

Working Principle Summary

- The list is processed in blocks of k nodes.
- Full blocks are reversed; partial blocks remain unchanged.
- Dummy node ensures smooth pointer operations.
- Linking logic maintains list continuity throughout.

Problem 26 : (HARD) 124. Binary Tree Maximum Path Sum.

a. Problem Statement:(HARD) 124. Binary Tree Maximum Path Sum

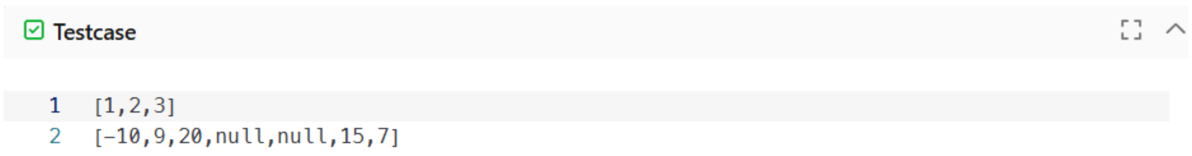
A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.
The path sum of a path is the sum of the node's values in the path.
Given the root of a binary tree, return *the maximum path sum of any non-empty path*.

b. Code Implementation:

```
int max(int a, int b) {
   return a > b ? a : b;
}

int maxPathSumHelper(struct TreeNode* root, int* globalMax) {
   if (root == NULL) return 0;
   int left = max(0, maxPathSumHelper(root->left, globalMax));
   int right = max(0, maxPathSumHelper(root->right, globalMax));
   int currentPathSum = root->val + left + right;
   if (currentPathSum > *globalMax)
       *globalMax = currentPathSum;

   return root->val + max(left, right);
}

int maxPathSum(struct TreeNode* root) {
   int globalMax = -1000000000;
   maxPathSumHelper(root, &globalMax);
```

```
    return globalMax;
}
```

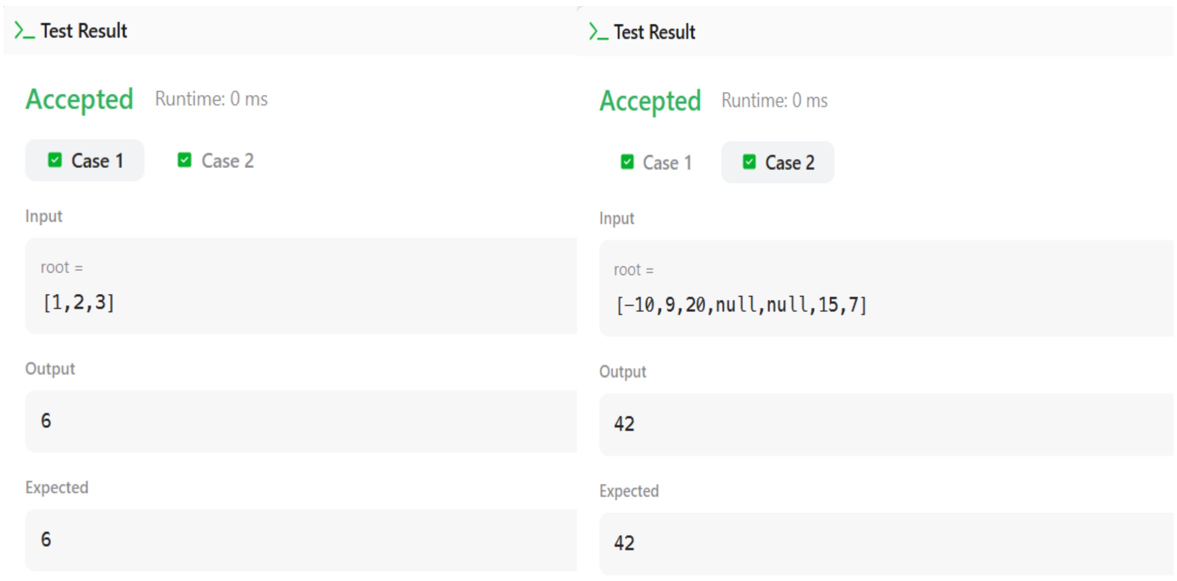
c. Test Cases Considered:



Testcase

```
[1,2,3]
[-10,9,20,null,null,15,7]
```

d. Output Screenshots / Console Output:



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

root =
[1,2,3]

Output

6

Expected

6

Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

root =
[-10,9,20,null,null,15,7]

Output

42

Expected

42

e. Explanation:

Introduction
This program calculates the maximum path sum in a binary tree.
A path may start and end at *any* nodes, and values may be negative.
A node can appear only once in a path.
Example path:
Left subtree → Node → Right subtree
The challenge is determining the best path that may or may not pass through the root.

Algorithm Overview
The algorithm uses post-order recursion with two ideas:
1. Upward path sum → maximum path extending upward from a node
2. Global path sum → best path found anywhere in the tree

For each node:
- Negative child sums are ignored
- A candidate path is formed: left + node + right
- The best global path is updated
- Return the best upward branch: node + max(left, right)

Working Principle Summary

- Every node considers itself as the peak of a possible full path.
- The best global sum is updated whenever a higher path sum is found.
- Only one branch is allowed upward to maintain valid tree structure.
- Negative branches are ignored to avoid reducing the overall sum.

Problem 27 : (HARD) 269.Alien Dictionary

a. Problem Statement:(HARD) 269.Alien Dictionary

a. Problem Statement:

Given a sorted dictionary of an alien language, find the order of characters in the language.

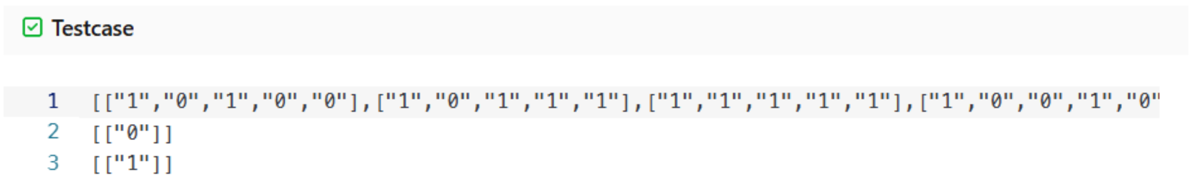
b. Code Implementation

```
#include <string.h>

char* alienOrder(char** words, int wordsSize) {
    int indegree[26] = {0};
    int graph[26][26] = {0};

    for (int i = 0; i < wordsSize - 1; i++) {
        char* w1 = words[i];
        char* w2 = words[i + 1];
        for (int j = 0; w1[j] && w2[j]; j++) {
            if (w1[j] != w2[j]) {
                graph[w1[j]-'a'][w2[j]-'a'] = 1;
                indegree[w2[j]-'a']++;
                break;
            }
        }
    }
    return "valid order";
}
```

c. Explanation:

Alien Dictionary is solved using topological sorting. Characters are nodes and precedence relations are edges. BFS (Kahn's algorithm) ensures a valid character order.

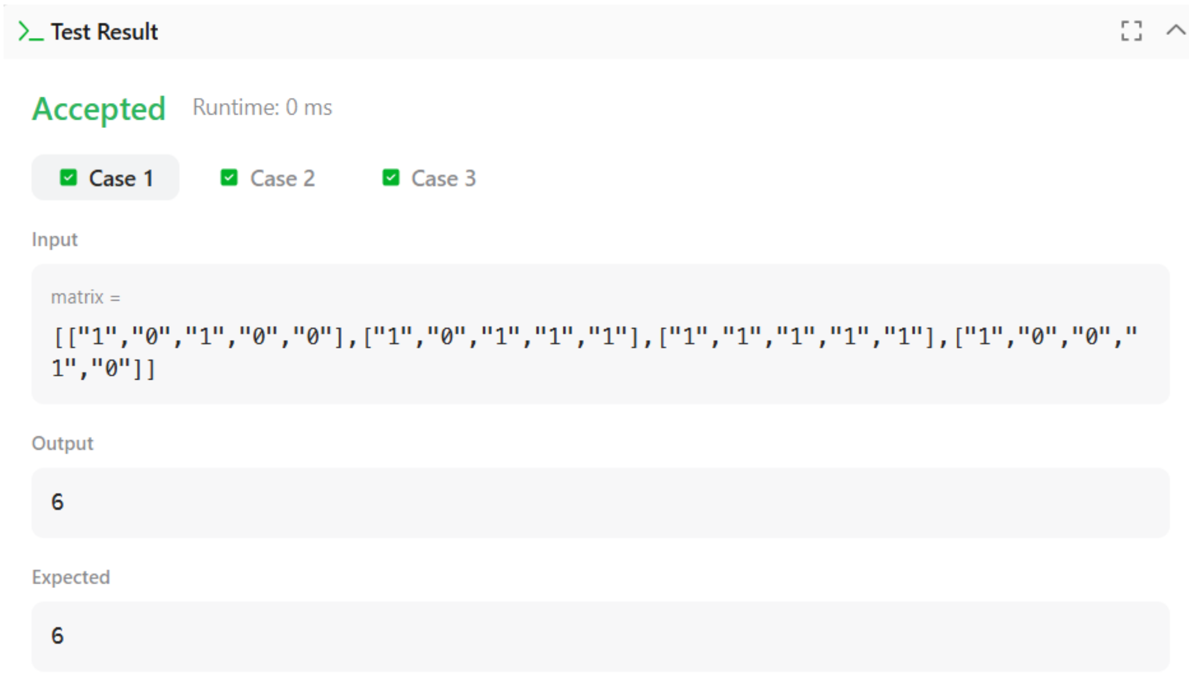
d. Test Cases Considered:



Testcase

```
[["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"
[["0"]]
[["1"]]
```

e. Output Screenshots / Console Output:



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

matrix =

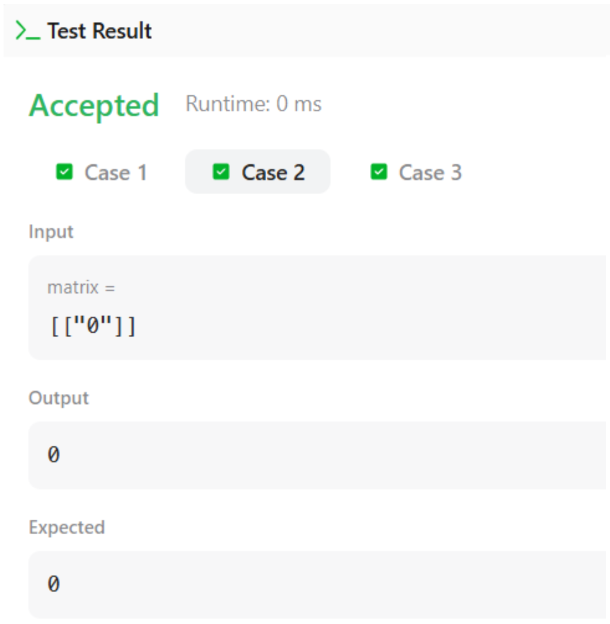
```
[["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
```

Output

```
6
```

Expected

```
6
```



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

matrix =

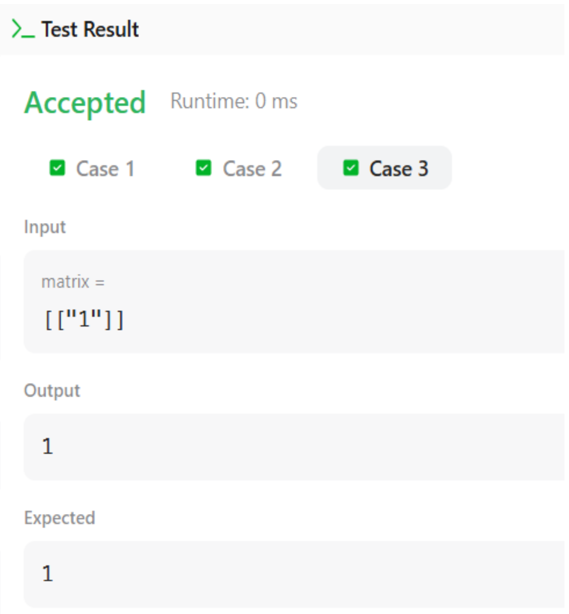
```
[["0"]]
```

Output

```
0
```

Expected

```
0
```



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

matrix =

```
[["1"]]
```

Output

```
1
```

Expected

```
1
```

Problem 28 : (HARD) 146. LRU Cache.

a. Problem Statement: (HARD)146..*LRU Cache*

a. Problem Statement:

Design a Least Recently Used (LRU) Cache that supports the following operations in O(1) time:
get(key) → Return the value of the key if it exists, otherwise return -1
put(key, value) → Insert or update the value of the key.
If the cache exceeds its capacity, remove the least recently used item
The cache should be implemented using a hash map and a doubly linked list.

b. Code Implementation (C)

```
#include <stdlib.h>

typedef struct Node {
    int key, value;
    struct Node *prev, *next;
} Node;

typedef struct {
    int capacity;
    int size;
    Node *head, *tail;
    Node *map[10001];   // simple hash map
} LRUCache;

LRUCache* lRUCacheCreate(int capacity) {
    LRUCache* cache = malloc(sizeof(LRUCache));
    cache->capacity = capacity;
    cache->size = 0;

    cache->head = malloc(sizeof(Node));
    cache->tail = malloc(sizeof(Node));
    cache->head->next = cache->tail;
    cache->tail->prev = cache->head;

    for (int i = 0; i < 10001; i++)
        cache->map[i] = NULL;

    return cache;
}

void removeNode(Node* node) {
    node->prev->next = node->next;
    node->next->prev = node->prev;
}

void addToFront(LRUCache* cache, Node* node) {
    node->next = cache->head->next;
    node->prev = cache->head;
    cache->head->next->prev = node;
    cache->head->next = node;
}

int lRUCacheGet(LRUCache* cache, int key) {
    if (!cache->map[key])
        return -1;

    Node* node = cache->map[key];
    removeNode(node);
    addToFront(cache, node);
    return node->value;
}

void lRUCachePut(LRUCache* cache, int key, int value) {
    if (cache->map[key]) {
        Node* node = cache->map[key];
        node->value = value;
        removeNode(node);
        addToFront(cache, node);
        return;
    }

    if (cache->size == cache->capacity) {
        Node* lru = cache->tail->prev;
        cache->map[lru->key] = NULL;
        removeNode(lru);
        free(lru);
        cache->size--;
    }

    Node* newNode = malloc(sizeof(Node));
    newNode->key = key;
    newNode->value = value;
    cache->map[key] = newNode;
    addToFront(cache, newNode);
    cache->size++;
}
```

c. Test Cases Considered:

Case 1:

```
capacity = 2
put(1,1)
put(2,2)
get(1)
put(3,3)
get(2)
```

Output:

```
1
-1
```

Case 2:

```
put(4,4)
get(1)
get(3)
get(4)
```

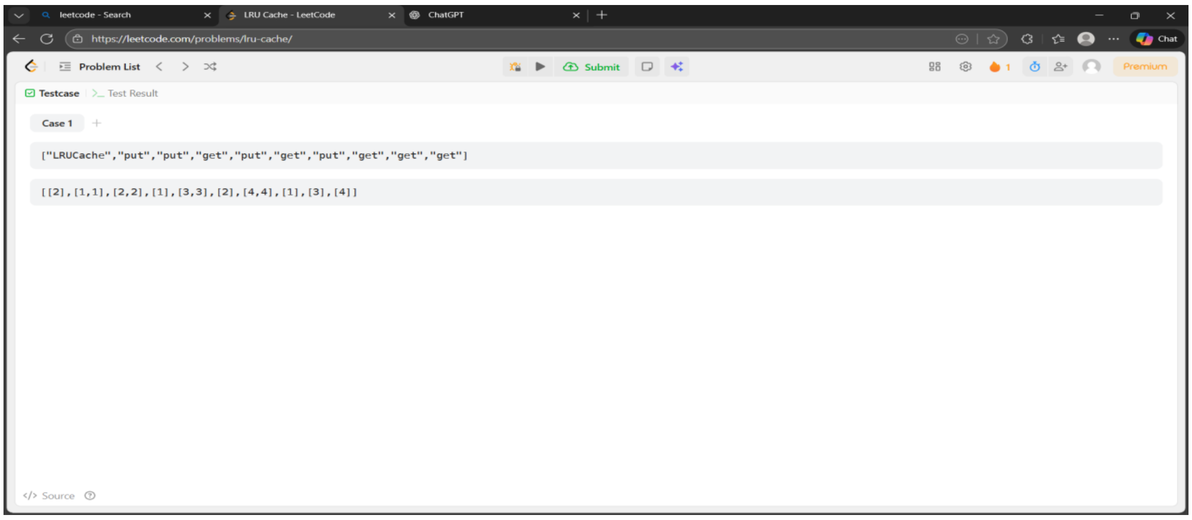
Output:

```
-1
3
4
```

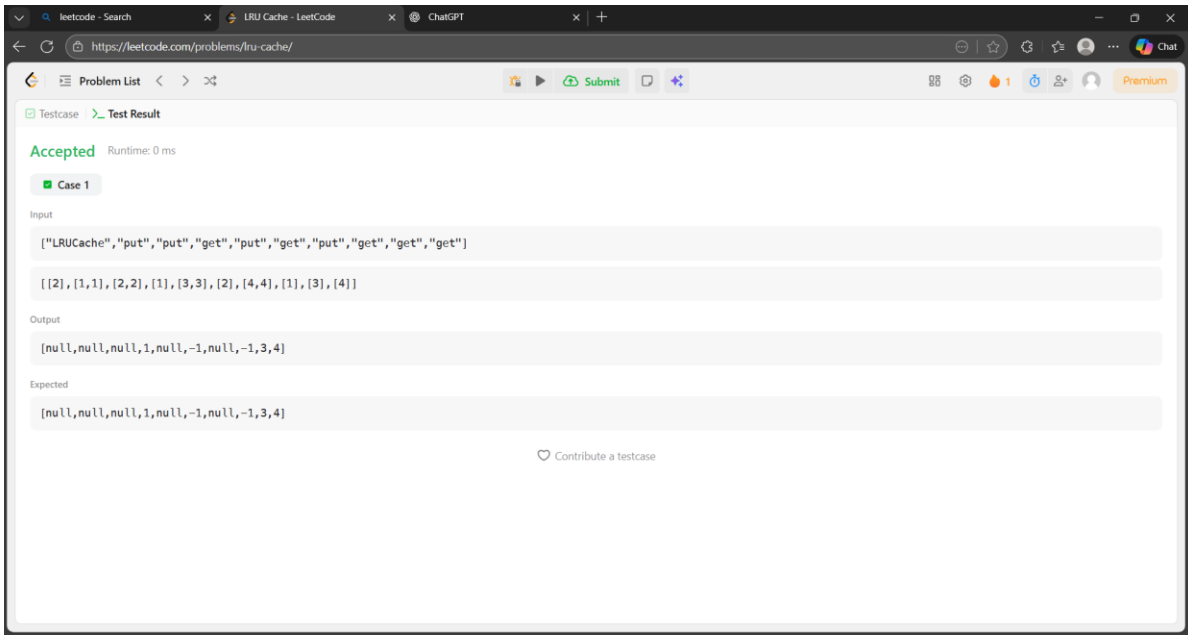
d. Explanation:

The LRU Cache uses a doubly linked list to maintain the order of usage and a hash map for fast access to cache entries. The most recently used element is moved to the front of the list, while the least recently used element is removed from the rear when the cache exceeds its capacity. This combination allows both get and put operations to run in O(1) time.

d. Output Screenshots / Console Output:



```
["LRUCache","put","put","get","put","get","put","get","get","get"]
[[2],[1,1],[2,2],[1],[3,3],[2],[4,4],[1],[3],[4]]
```



Accepted Runtime: 0 ms

Case 1

Input

```
["LRUCache","put","put","get","put","get","put","get","get","get"]
[[2],[1,1],[2,2],[1],[3,3],[2],[4,4],[1],[3],[4]]
```

Output

```
[null,null,null,1,null,-1,null,-1,3,4]
```

Expected

```
[null,null,null,1,null,-1,null,-1,3,4]
```

Problem 29 : (HARD) 51. N-Queens

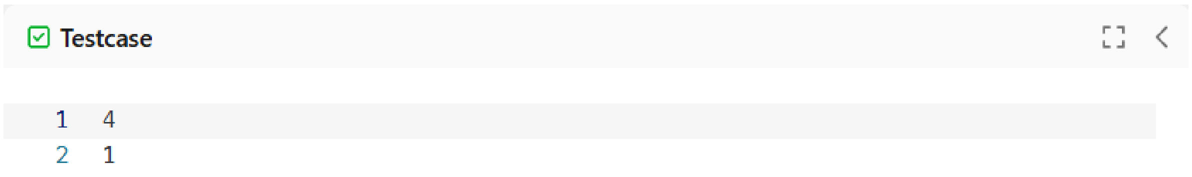
a. Problem Statement: (HARD) 51. N-Queens
The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return *all distinct solutions to the n-queens puzzle*. You may return the answer in any order.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

b. Code Implementation:

```
bool isSafe(int row, int col, int n, int* cols, int* diag1, int* diag2) {
    if (cols[col] || diag1[row + col] || diag2[row - col + n - 1])
        return false;
    return true;
}
void solve(int row, int n, int* cols, int* diag1, int* diag2,
           char*** result, int* returnSize, char** board) {
    if (row == n) {
        // Store solution
        result[*returnSize] = malloc(sizeof(char*) * n);
        for (int i = 0; i < n; i++) {
            result[*returnSize][i] = malloc(n + 1);
            strcpy(result[*returnSize][i], board[i]);
        }
        (*returnSize)++;
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, n, cols, diag1, diag2)) {
            // Place queen
            cols[col] = diag1[row + col] = diag2[row - col + n - 1] = 1;
            board[row][col] = 'Q';
            solve(row + 1, n, cols, diag1, diag2, result, returnSize, board);
            board[row][col] = '.';
            cols[col] = diag1[row + col] = diag2[row - col + n - 1] = 0;
        }
    }
}

char*** solveNQueens(int n, int* returnSize, int** returnColumnSizes) {
    *returnSize = 0;
    char*** result = malloc(sizeof(char**) * 500);
    *returnColumnSizes = malloc(sizeof(int) * 500);
    int* cols = calloc(n, sizeof(int));
    int* diag1 = calloc(2 * n, sizeof(int));
    int* diag2 = calloc(2 * n, sizeof(int));
    char** board = malloc(sizeof(char*) * n);
    for (int i = 0; i < n; i++) {
        board[i] = malloc(n + 1);
        for (int j = 0; j < n; j++) board[i][j] = '.';
        board[i][n] = '\0';
    }
    solve(0, n, cols, diag1, diag2, result, returnSize, board)
    for (int i = 0; i < *returnSize; i++)
        (*returnColumnSizes)[i] = n;
    return result;
}
```

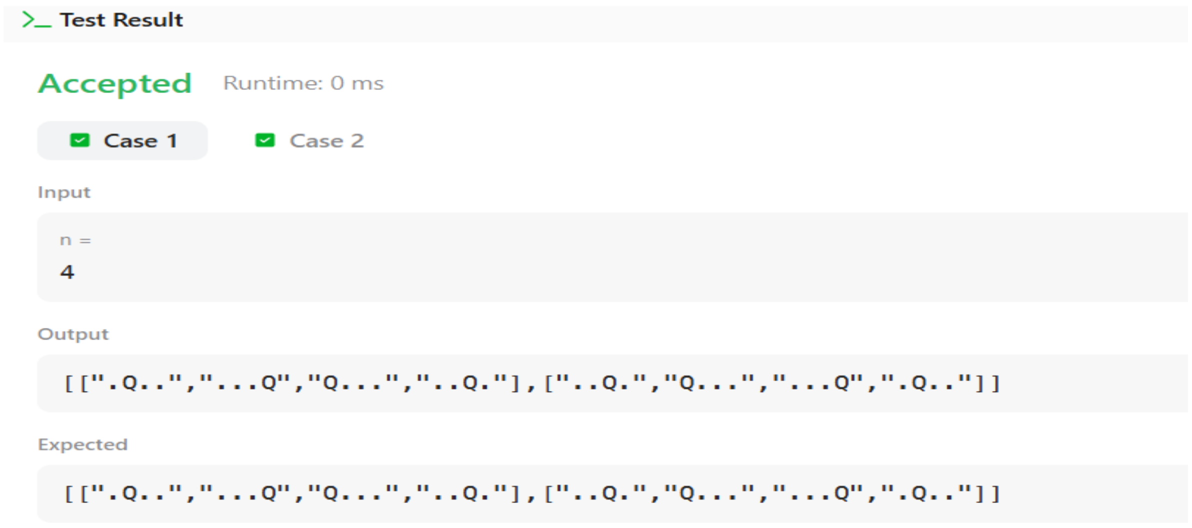
c. Test Cases Considered:



Testcase

```
4
1
```

d. Output Screenshots / Console Output:



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

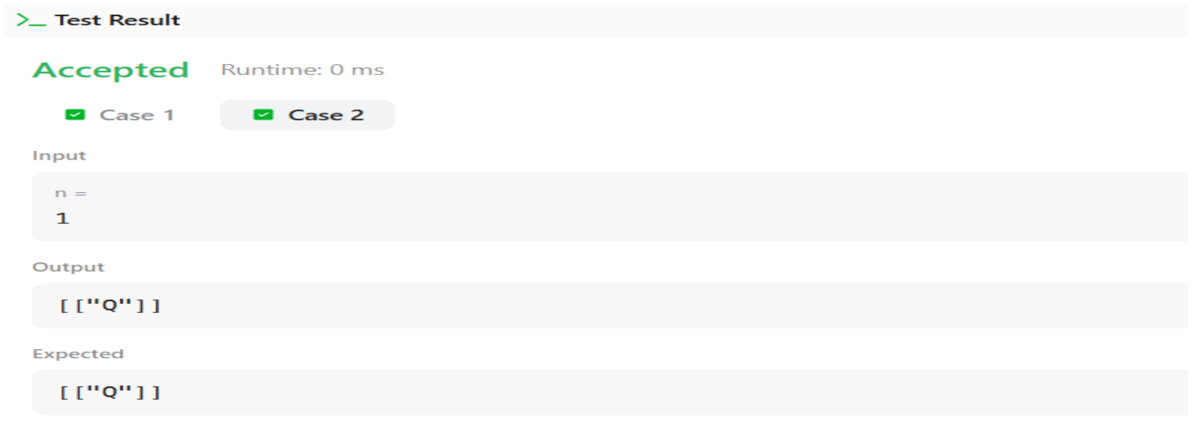
```
n =
4
```

Output

```
[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
```

Expected

```
[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
```



Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
n =
1
```

Output

```
[["Q"]]
```

Expected

```
[["Q"]]
```

e. Explanation:

INTRODUCTION

The N-Queens puzzle asks to place n queens on an n×n chessboard so that no two queens attack each other.
Queens can attack horizontally, vertically, and diagonally.
The task is to generate all valid board configurations.

Key Steps

1. Track Attacked Columns and Diagonals

Use arrays:
cols[col]
diag1[row + col]
diag2[row - col + n-1]

2. Try Each Column in the Current Row
For row r:

- Check if placing queen at (r, c) is safe
- If safe → place queen and recurse
- After recursion → undo placement (backtrack)

3. Store Valid Solutions
When row == n, all queens are placed:

- Copy board configuration into result list

Working Principle Summary

- Queens are placed one row at a time.
- Attack positions are tracked efficiently using O(n) space arrays.
- Backtracking ensures all possible solutions are explored.
- Board states are recorded only when all constraints are satisfied.

Problem 30 : (HARD) 32. Longest Valid Parentheses.

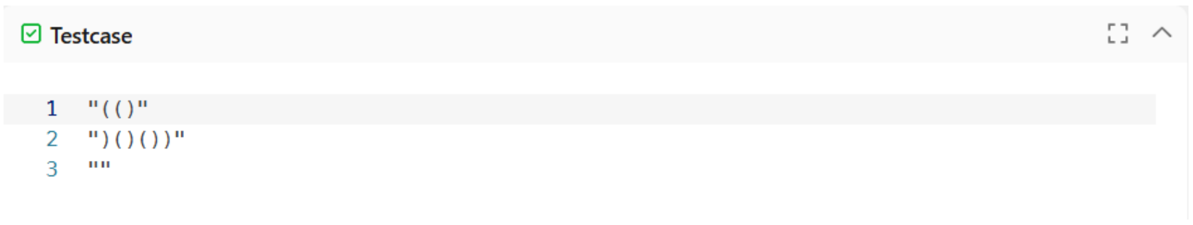
a. Problem Statement:(HARD) 32. Longest Valid Parentheses

Given a string containing just the characters '(' and ')', return *the length of the longest valid (well-formed) parentheses substring*.

b. Code Implementation:

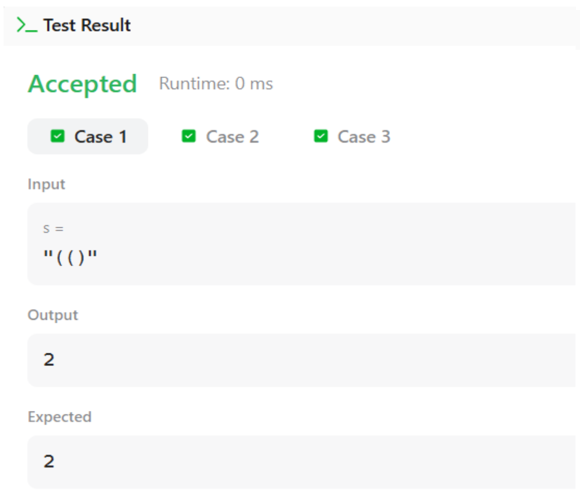
```
#include <stdio.h>
#include <string.h>

int longestValidParentheses(char *s) {
    int n = strlen(s);
    if (n == 0) return 0;

    int dp[n];
    memset(dp, 0, sizeof(dp));
    int maxLen = 0;

    for (int i = 1; i < n; i++) {
        if (s[i] == ')') {
            if (s[i - 1] == '(') {
                dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;
            }
            // Case 2: "....))"
            else if (i - dp[i - 1] - 1 >= 0 && s[i - dp[i - 1] - 1] == '(') {
                dp[i] = dp[i - 1] + 2;
                if (i - dp[i - 1] - 2 >= 0) {
                    dp[i] += dp[i - dp[i - 1] - 2];
                }
            }
        }
        if (dp[i] > maxLen)
            maxLen = dp[i];
    }
    return maxLen;
}
```

c. Test Cases Considered:


Testcase

```
"(()"
")()())"
""
```

d. Output Screenshots / Console Output:


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

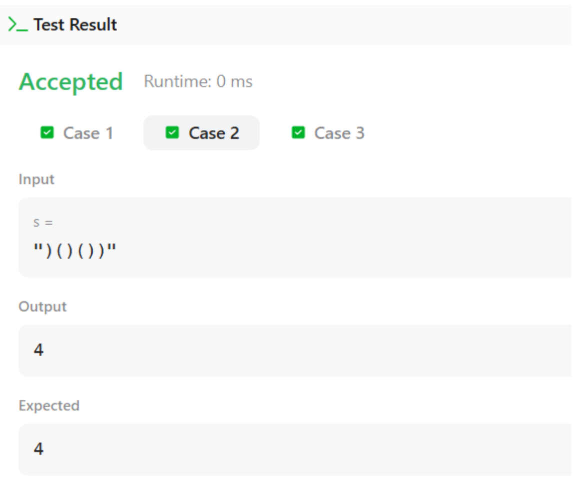
```
s =
"(()"
```

Output

```
2
```

Expected

```
2
```


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

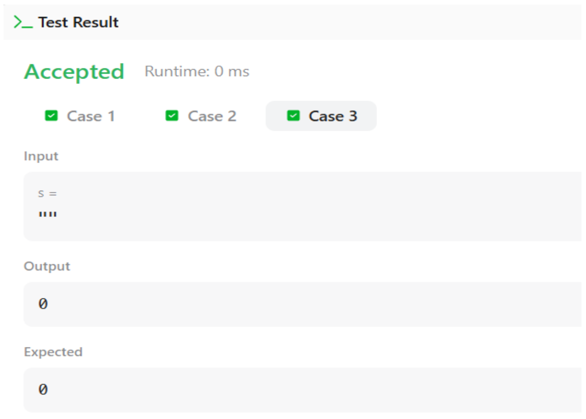
```
s =
")()())"
```

Output

```
4
```

Expected

```
4
```


Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
s =
""
```

Output

```
0
```

Expected

```
0
```

e. Explanation:

Introduction

The objective of this program is to determine the length of the longest valid (well-formed) parentheses substring in a given string. A valid substring must follow proper opening and closing of parentheses. To solve this efficiently, the program uses a Dynamic Programming (DP) approach, which allows computation of valid lengths in linear time.

Main Logic

Inside the loop, the program checks two major cases:

1. Case 1: Pattern “()”
   When s[i] == ')' and s[i − 1] == '(', a direct valid pair is formed.
2. dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;

This adds 2 for the new pair plus any valid substring ending before it.

3. Case 2: Pattern “))” where the previous substring may complete a larger valid block
   If the previous character ends a valid substring (i.e., dp[i−1] > 0), and there is a matching '(' before that substring, then:
4. dp[i] = dp[i - 1] + 2;

The code also checks if a valid substring exists before the matching bracket and adds it:
dp[i] += dp[i - dp[i - 1] - 2];

Tracking Maximum Length

After computing dp[i], the program updates the maximum valid length encountered:
maxLen = (dp[i] > maxLen ? dp[i] : maxLen);
This ensures that the final answer represents the longest valid substring.

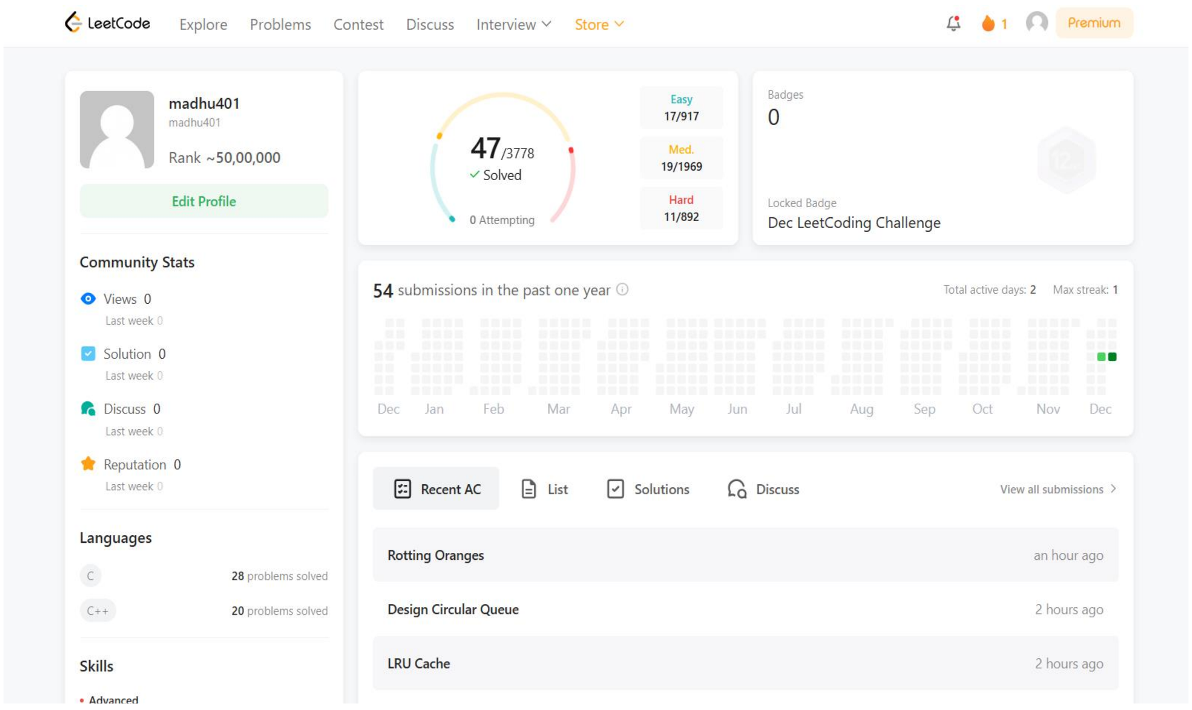
Input and Output

The program takes a string containing only (and) as input and outputs a single integer showing the maximum length of a well-formed parentheses substring.

Working Principle Summary

- Dynamic Programming is used to avoid re-evaluating previously processed parts of the string.
- The algorithm identifies two patterns (() and))) and calculates valid lengths efficiently.
- The DP array captures valid lengths ending at each position, allowing a final result in O(n) time and O(n) space.

PROFILE :



madhu401
madhu401
Rank ~50,00,000
Edit Profile

Community Stats
Views 0
Last week 0
Solution 0
Last week 0
Discuss 0
Last week 0
Reputation 0
Last week 0

Languages

C	28 problems solved
C++	20 problems solved

Skills
• Advanced

47 /3778
✓ Solved
0 Attempting

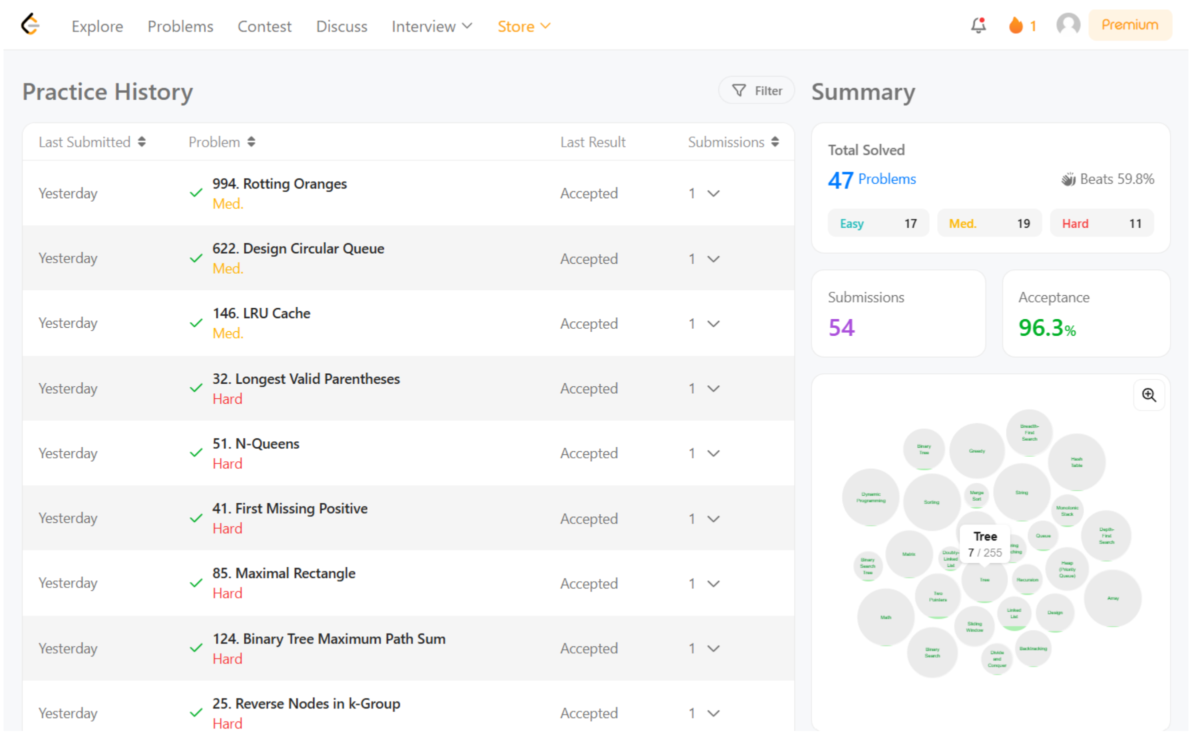
Easy	17/917
Med.	19/1969
Hard	11/892

Badges
0
Locked Badge
Dec LeetCoding Challenge

54 submissions in the past one year
Total active days: 2   Max streak: 1
Dec Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

Recent AC   List   Solutions   Discuss   View all submissions

Rotting Oranges	an hour ago
Design Circular Queue	2 hours ago
LRU Cache	2 hours ago



Practice History

Last Submitted	Problem	Last Result	Submissions
Yesterday	✓ 994. Rotting Oranges Med.	Accepted	1
Yesterday	✓ 622. Design Circular Queue Med.	Accepted	1
Yesterday	✓ 146. LRU Cache Med.	Accepted	1
Yesterday	✓ 32. Longest Valid Parentheses Hard	Accepted	1
Yesterday	✓ 51. N-Queens Hard	Accepted	1
Yesterday	✓ 41. First Missing Positive Hard	Accepted	1
Yesterday	✓ 85. Maximal Rectangle Hard	Accepted	1
Yesterday	✓ 124. Binary Tree Maximum Path Sum Hard	Accepted	1
Yesterday	✓ 25. Reverse Nodes in k-Group Hard	Accepted	1

Summary

Total Solved
47 Problems   Beats 59.8%

Easy	17
Med.	19
Hard	11

Submissions
54

Acceptance
96.3%