VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELAGAVI - 590 018, KARNATAKA

A Report on

“Data Structure and Applications “

AAT

(BCS304)

in

Artificial Intelligence and Machine Learning

By

Mr. Elias Hatim Omar Al Nadary **USN: 1BY23AI048**

Dr. Shruthi S
Associate Professor
Department of AI&ML, BMSIT&M.

ಬಿ.ಎಂ.ಎಸ್. ತಾಂತ್ರಿಕ ಮತ್ತು ವ್ಯವಸ್ಥಾಪನಾ ಮಹಾವಿದ್ಯಾಲಯ
BMS Institute of Technology and Management
(An Autonomous Institution Affiliated to VTU, Belagavi)
Avalahalli, Doddaballapur Main Road, Bengaluru – 560119
2024-2025

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELAGAVI – 590 018, KARNATAKA

ಬಿ.ಎಂ.ಎಸ್. ತಾಂತ್ರಿಕ ಮತ್ತು ವ್ಯವಸ್ಥಾಪನಾ ಮಹಾವಿದ್ಯಾಲಯ
BMS Institute of Technology and Management
(An Autonomous Institution Affiliated to VTU, Belagavi)
Avalahalli, Doddaballapur Main Road, Bengaluru – 560119

This is to certify that the AAT **"Data structure and Applications"** is the work carried out by **Mr.Elias Hatim Omar Al-Nadary (1BY23AI048)** of Data Structure and Applications Theory (BCS304) of the BMSIT&M during the year 2024-25. The report has been approved as it satisfies the academic requirements in respect AAT work for the B.E Degree.

____

Signature of the Course Cordinator

Marks Distribution

1.

2.

3

Total

`

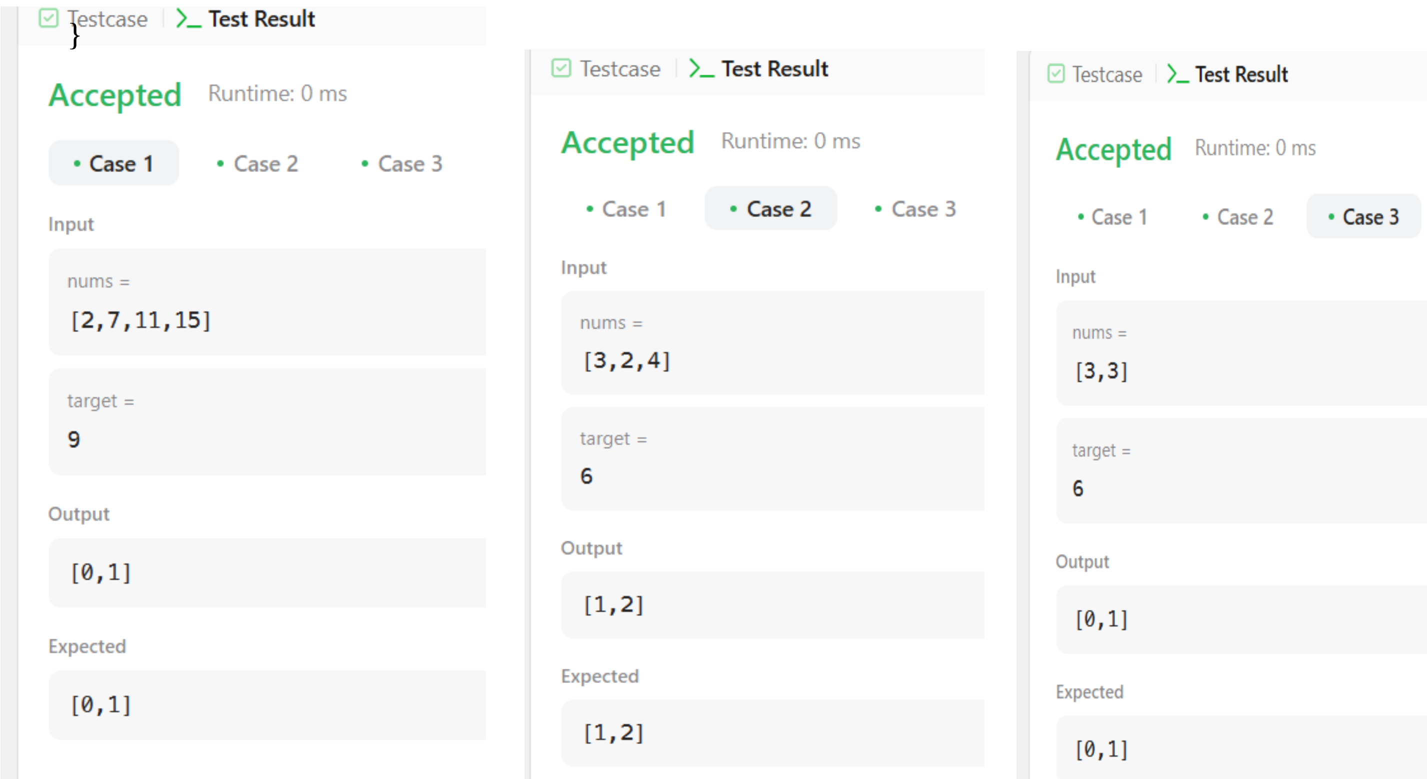
Programs

Array

Program 1: Two Sum (Easy level) Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*. You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

CODE:

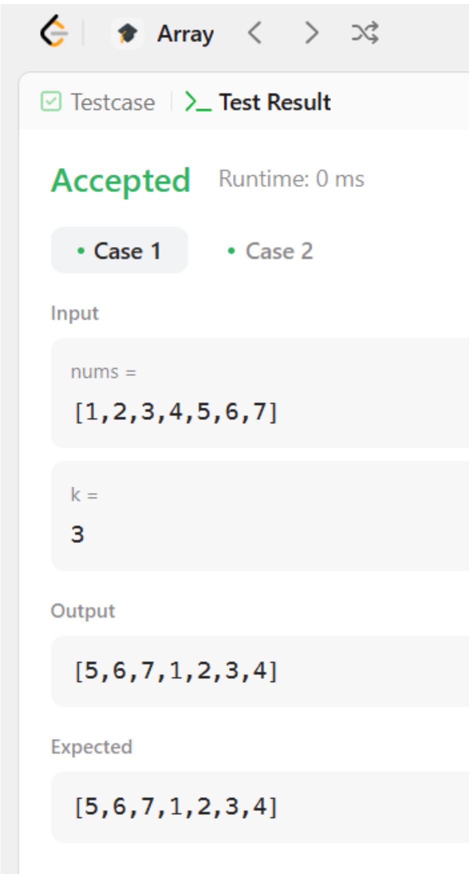
```
int* twoSum(int* nums, int numsSize, int target, int* returnSize) {

// Allocating memory for the result array int* result =
(int*)malloc(2 * sizeof(int));

// Searching for the two indices

for (int i = 0; i < numsSize - 1; i++) {
for (int j = i + 1; j < numsSize; j++) { if (nums[i] +
nums[j] == target) {

result[0] = i; result[1] = j;

*returnSize = 2; // Set the size of the result array return result;

}

}

}


// If no solution found, return NULL

*returnSize = 0; return NULL;
}
```

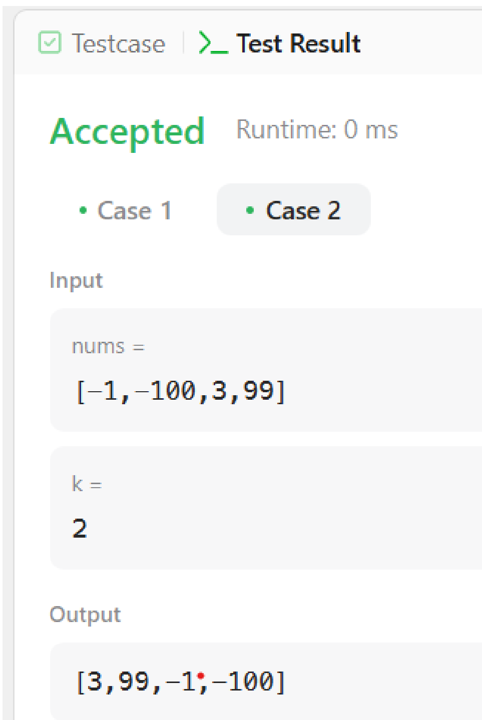


Program 2: Container with most water (medium) You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).Find two lines that together with the x-axis form a container, such that the container contains the most water.Return the maximum amount of water a container can store.

CODE:

```
#include <stdio.h>

    // Function to calculate the maximum area of water
    int maxArea(int* height, int heightSize) {
        int left = 0;           // Pointer at the beginning
        int right = heightSize - 1; // Pointer at the end
        int maxArea = 0;        // To store the maximum area

        while (left < right) {

            // Calculate the area between the current left and right pointers
            int width = right - left;
            int currentArea = width * (height[left] < height[right] ? height[left] : height[right]);


            // Update maxArea if the current area is larger
            if (currentArea > maxArea) {
                maxArea = currentArea;
            }


            // Move the pointer pointing to the shorter line
            if (height[left] <
            height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return maxArea;
    }
```

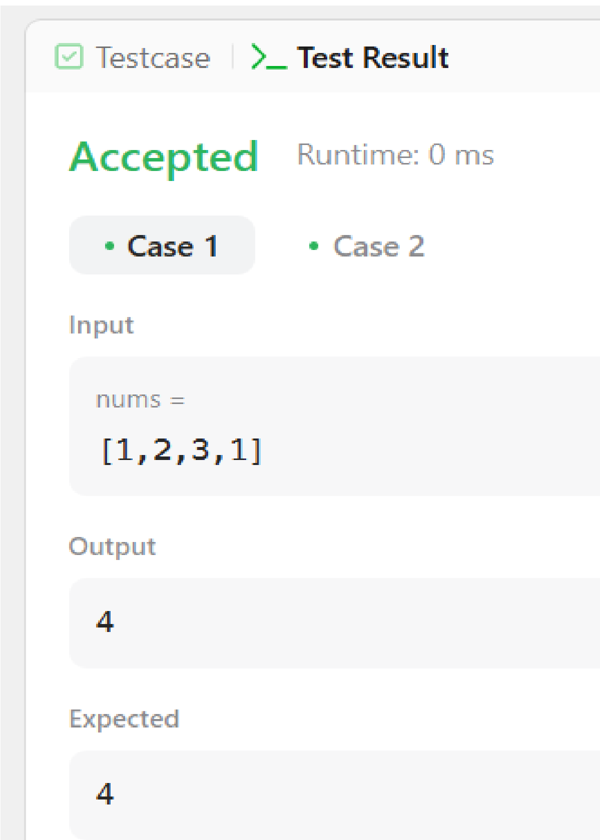


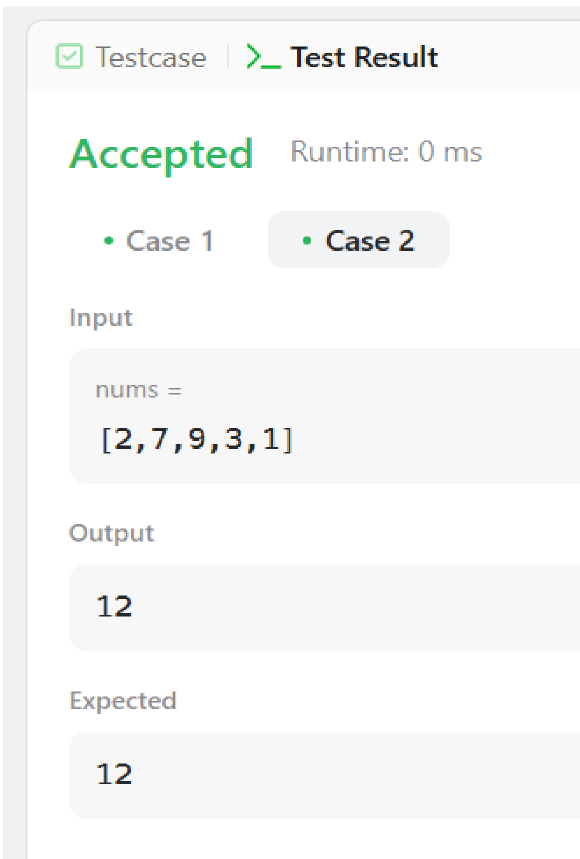


Program 3 : Rotate Array (medium) Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

CODE:

```
void reverse(int* nums, int start, int end) {
    while (start < end) {

        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;

        end--;
    }

}


void rotate(int* nums, int numsSize, int k) {
   k = k % numsSize;

    reverse(nums, 0, numsSize - 1);

    reverse(nums, 0, k - 1);
    reverse(nums, k, numsSize - 1);

}
```





Program 4: House robber (medium) You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

CODE:

```
int rob(int* nums, int numsSize) {
    if (numsSize == 0) return 0;

    if (numsSize == 1) return nums[0];

    int prev2 = 0; // dp[i-2], initially 0
    int prev1 = 0; // dp[i-1], initially 0
    int curr = 0;  // dp[i]


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

        curr = (prev1 > nums[i] + prev2) ? prev1 : (nums[i] + prev2);
        prev2 = prev1;
        prev1 = curr;

    }

    return curr;

}
```

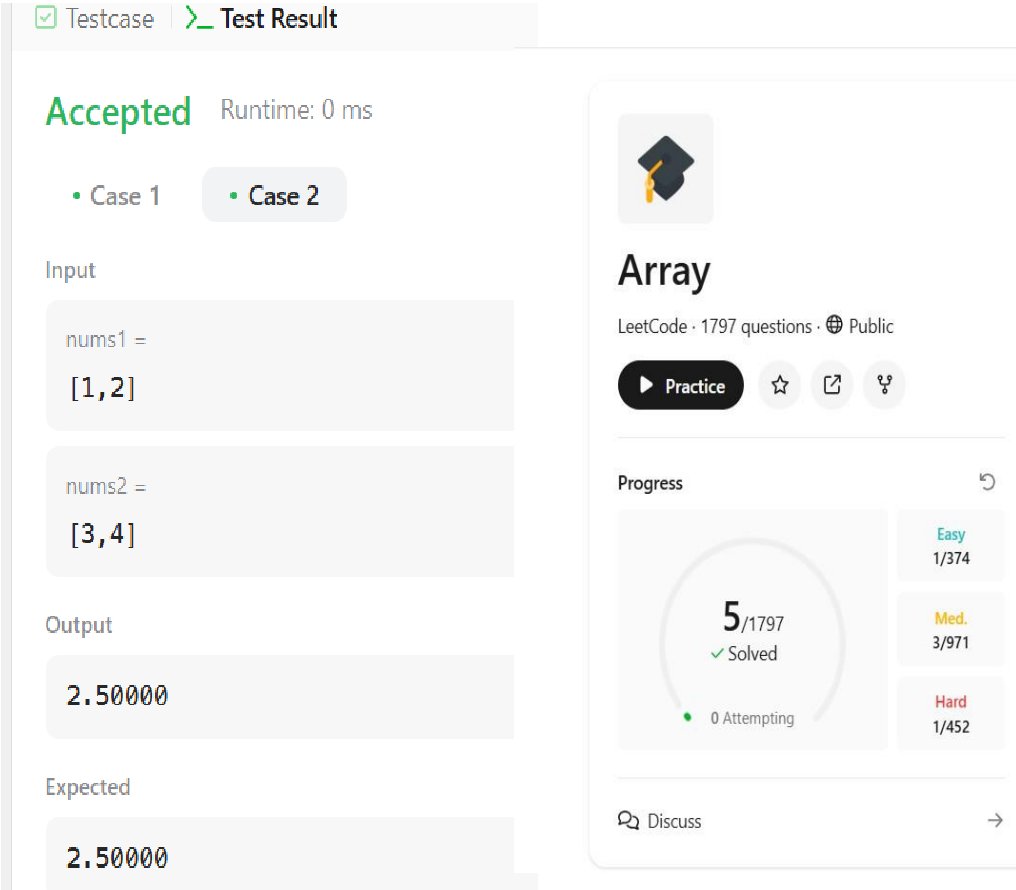
`

Program 5: Median of Two Sorted Arrays Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.The overall run time complexity should be O(log (m+n)).

CODE:

```
double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int nums2Size) {
    if (nums1Size > nums2Size) {
        // Ensuring nums1 is the smaller array
        return findMedianSortedArrays(nums2, nums2Size, nums1, nums1Size);
    }


    int l = 0, r = nums1Size;
    int mid = (nums1Size + nums2Size + 1) / 2;
    while (l <= r) {
        int partition1 = (l + r) / 2;
        int partition2 = mid - partition1;
        int maxLeft1 = (partition1 == 0) ? INT_MIN : nums1[partition1 - 1];
        int minRight1 = (partition1 == nums1Size) ? INT_MAX : nums1[partition1];
        int maxLeft2 = (partition2 == 0) ? INT_MIN : nums2[partition2 - 1];
        int minRight2 = (partition2 == nums2Size) ? INT_MAX : nums2[partition2];
        if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
            // Found the correct partition
            if ((nums1Size + nums2Size) % 2 == 0) {
                return (fmax(maxLeft1, maxLeft2) + fmin(minRight1, minRight2)) / 2.0;
            } else {
                return fmax(maxLeft1, maxLeft2);
            }
        } else if (maxLeft1 > minRight2) {
            // Move partition1 to the left
            r = partition1 - 1;
        } else {
            // Move partition1 to the right
            l = partition1 + 1;
        }
    }
    return 0.0; }
```


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums1 =
[1,3]
nums2 =
[2]
Output
2.00000
Expected
2.00000


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums1 =
[1,2]
nums2 =
[3,4]
Output
2.50000
Expected
2.50000
Array
LeetCode · 1797 questions · Public
Practice
Progress
5/1797
Solved
0 Attempting
Easy
1/374
Med.
3/971
Hard
1/452
Discuss

`

Stacks

Program 1: Baseball Game (Easy) You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.You are given a list of strings operations, where operations[i] is the ith operation you must apply to the record and is one of the following:

-An integer x.

-Record a new score of x.

-Record a new score that is the sum of the previous two scores.

-Record a new score that is the double of the previous score.

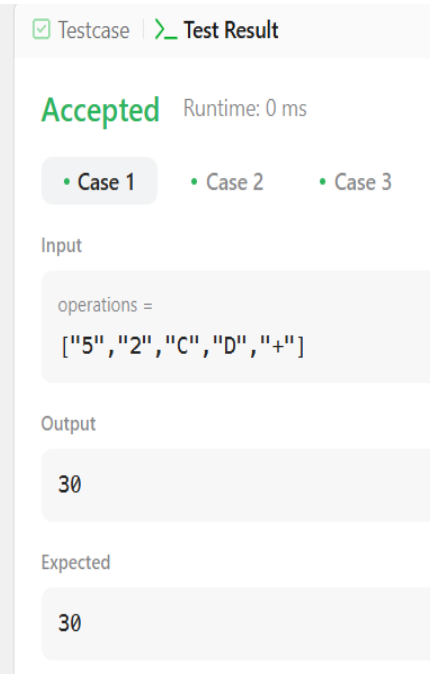
-Invalidate the previous score, removing it from the record.

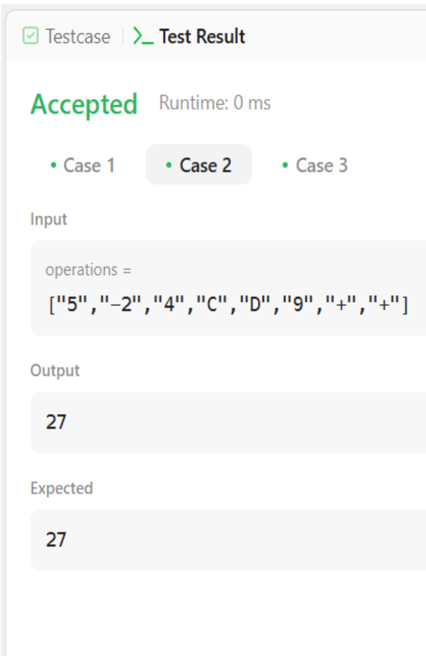
-Return the sum of all the scores on the record after applying all the operations.

CODE:

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

#define MAX_SIZE 1000
 int calPoints(char** ops, int opsSize) {

int stack[MAX_SIZE];
    int top = -1;  // Stack pointer
    for (int i = 0; i < opsSize; i++) {
        char* op = ops[i];

        if (strcmp(op, "+") == 0) {
            if (top >= 1) {  // Ensure at least two elements are present
                int newtop = stack[top] + stack[top - 1];

                stack[++top] = newtop;
            }
        } else if (strcmp(op, "C") == 0) {
            if (top >= 0) {  // Ensure there is at least one element to pop
                top--;  // Pop the last element

            }
        } else if (strcmp(op, "D") == 0) {
            if (top >= 0) {  // Ensure there is at least one element to double
                stack[++top] = 2 * stack[top];  // Double the last element
```

```
            }
        } else {
            stack[++top] = atoi(op);  // Convert string to integer and push
    }}

    // total score int ans = 0;

    for (int i = 0; i <= top; i++) {
        ans += stack[i];

    }
    return ans;
}
```





`

Program 2 : Min Stack (medium) Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

CODE:

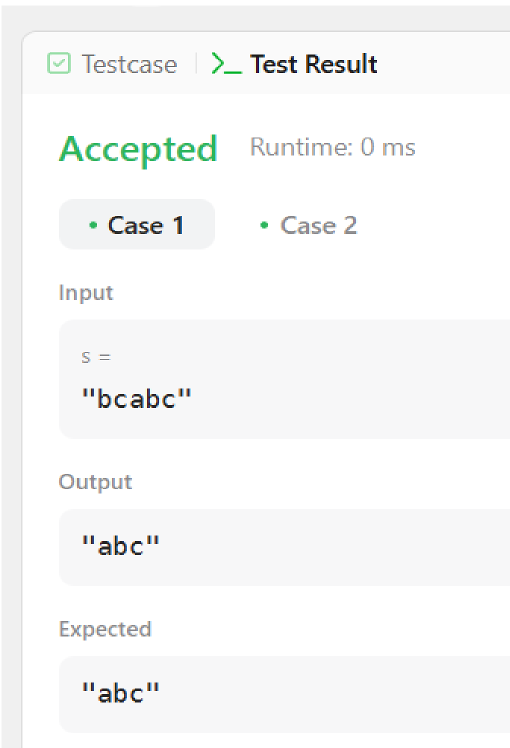
```
#include <stdlib.h>
#include <limits.h>

// Define the structure for MinStack
typedef struct {

    int *stack;
    int *minStack;
    int top;

    int minTop;
    int capacity;

} MinStack;


MinStack* minStackCreate() {
    MinStack* obj = (MinStack*)malloc(sizeof(MinStack));
    obj->capacity = 10000;  // Initialize with a default capacity
    obj->stack = (int*)malloc(sizeof(int) * obj->capacity);

    obj->minStack = (int*)malloc(sizeof(int) * obj->capacity);
    obj->top = -1;
    obj->minTop = -1;
    return obj;

}
void minStackPush(MinStack* obj, int val) {
    // Push onto the main stack
    obj->stack[++(obj->top)] = val;
    // Push onto the min stack
    if (obj->minTop == -1 || val <= obj->minStack[obj->minTop]) {
        obj->minStack[++(obj->minTop)] = val;
    }}
void minStackPop(MinStack* obj) {
    if (obj->top == -1) return;  // Stack is empty
 // Check if the element being popped is the minimum
    if (obj->stack[obj->top] == obj->minStack[obj->minTop]) {
        obj->minTop--;  // Pop from the min stack

    }
```

```
   obj->top--;  // Pop from the main stack
}
int minStackTop(MinStack* obj) {
    return obj->stack[obj->top];

}


int minStackGetMin(MinStack* obj) {
    return obj->minStack[obj->minTop];

}
void minStackFree(MinStack* obj) {
    free(obj->stack);

    free(obj->minStack);
    free(obj);

}
```

Program 3: Remove Duplicate Letters (medium) Given a string s, remove duplicate letters so that every letter appears once and only once.

CODE:

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


#define MAX_LEN 10004


char* removeDuplicateLetters(char* s) {
   int last[26] = {0}, inStack[26] = {0};
   char stack[MAX_LEN];
   int top = -1;

   for (int i = 0; s[i]; i++) last[s[i] - 'a'] = i;

   for (int i = 0; s[i]; i++) {
      char c = s[i];
      if (inStack[c - 'a']) continue;


      while (top >= 0 && stack[top] > c && last[stack[top] - 'a'] > i) {
         inStack[stack[top--] - 'a'] = 0;
      }


      stack[++top] = c;
      inStack[c - 'a'] = 1;
   }


   stack[top + 1] = '\0';
   return strdup(stack);
}
```



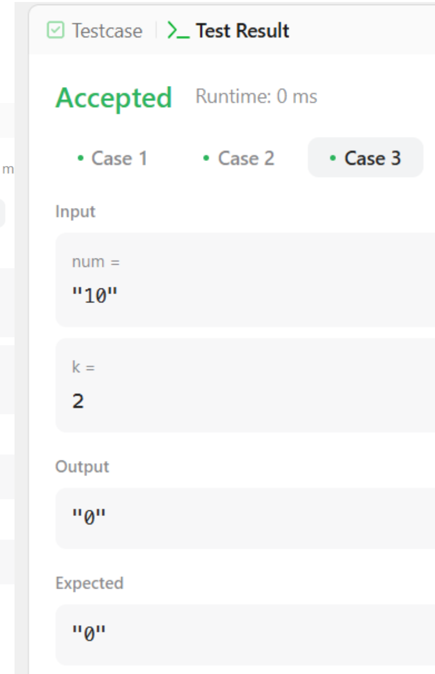
`

Program 4 : Remove K digits (medium) Given string num representing a non-negative integer num, and an integer k, return the smallest possible integer after removing k digits from num.

CODE:

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

char* removeKdigits(char* num, int k) {
    int n = strlen(num);

    int newLength = n - k;

    if (k >= n) {

        char* result = (char*)malloc(2 * sizeof(char));
        result[0] = '0';

        result[1] = '\0';
        return result;

    }

    char* stack = (char*)malloc((n + 1) * sizeof(char));
    int top = -1;

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

        while (top >= 0 && stack[top] > num[i] && k > 0) {
            top--;

            k--;

        }

        stack[++top] = num[i];

    }

    top -= k;

    stack[top + 1] = '\0';

    int start = 0;

    while (stack[start] == '0' && start <= top) {
        start++;

    }

    if (start > top) {

        char* result = (char*)malloc(2 * sizeof(char));
        result[0] = '0';

        result[1] = '\0';
        free(stack);
        return result;

    }
```

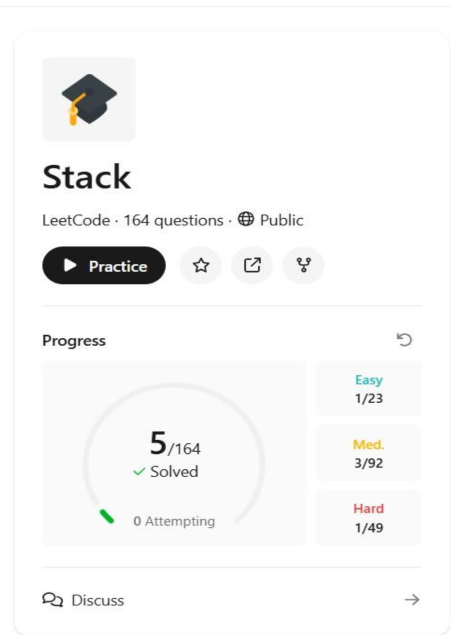
```
    char* result = (char*)malloc((top - start + 2) * sizeof(char));
    strcpy(result, stack + start);

    free(stack);
    return result;

}
```

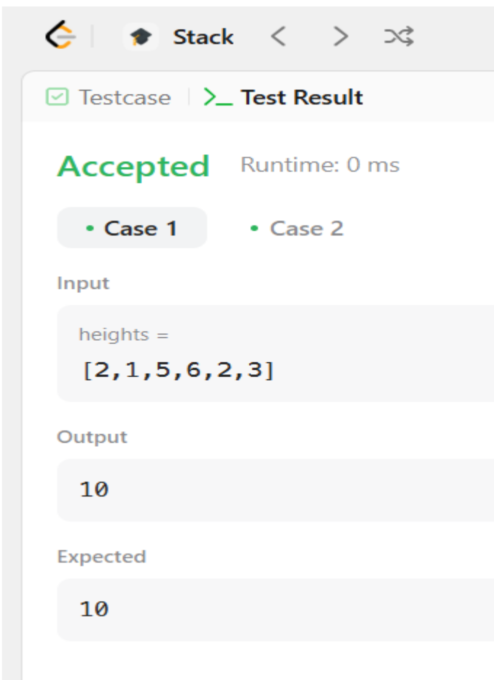


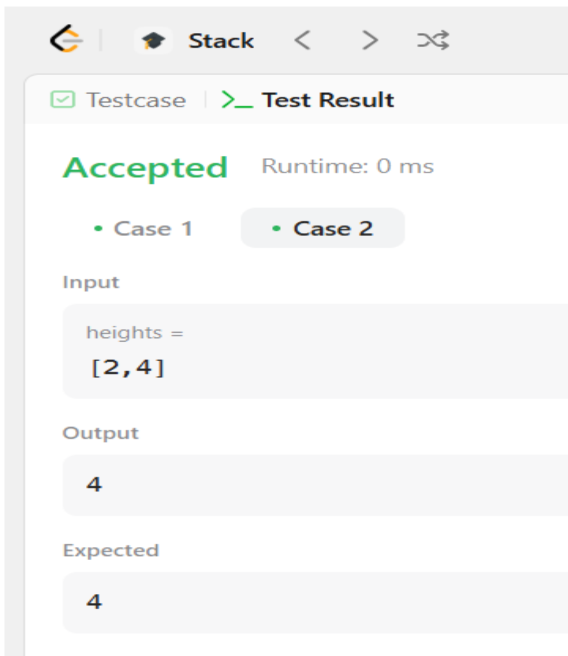
Program 5: Largest Rectangle in Histogram (hard) Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

CODE:

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

int largestRectangleArea(int* heights, int heightsSize) {
    int* stack = (int*)malloc((heightsSize + 1) * sizeof(int));
    int top = -1;

    int maxArea = 0;

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

        int currentHeight = (i == heightsSize) ? 0 : heights[i];

        while (top >= 0 && currentHeight < heights[stack[top]]) {
            int height = heights[stack[top--]];

            int width = (top == -1) ? i : (i - stack[top] - 1);

            maxArea = (height * width > maxArea) ? height * width : maxArea;

        }

        stack[++top] = i;

    }

    free(stack);
    return maxArea;

}
```







`

Queue

Program 1: Implement Queue using stacks (medium)

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).

CODE:

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

// Define the stack structure
typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

// Define the queue structure using two stacks
typedef struct {
    Stack *stack1;
    Stack *stack2;
} MyQueue;

Stack *createStack(int capacity) {
    Stack *stack = (Stack *)malloc(sizeof(Stack));
    stack->data = (int *)malloc(capacity * sizeof(int));
    stack->top = -1;
    stack->capacity = capacity;
    return stack;
}


// function to push an element onto a stack
void stackPush(Stack *stack, int x) {
    if (stack->top == stack->capacity - 1) return; // Stack overflow
    stack->data[++stack->top] = x;
}


//  function to pop an element from a stack
int stackPop(Stack *stack) {
    if (stack->top == -1) return -1; // Stack underflow
    return stack->data[stack->top--];
```

`

```
}

//  function to peek the top element of a stack
int stackPeek(Stack *stack) {

    if (stack->top == -1) return -1; // Stack is empty
    return stack->data[stack->top];

}

//  function to check if a stack is empty
bool stackIsEmpty(Stack *stack) {

    return stack->top == -1;

}

// Create a queue

MyQueue* myQueueCreate() {

    MyQueue *queue = (MyQueue *)malloc(sizeof(MyQueue));
    queue->stack1 = createStack(100); // Adjust capacity as needed
    queue->stack2 = createStack(100);

    return queue;

}

// Push an element to the back of the queue
void myQueuePush(MyQueue* obj, int x) {

    stackPush(obj->stack1, x);

}

// Pop the element from the front of the queue
int myQueuePop(MyQueue* obj) {

    if (stackIsEmpty(obj->stack2)) {

        while (!stackIsEmpty(obj->stack1)) {
            stackPush(obj->stack2, stackPop(obj->stack1));

        }

    }

    return stackPop(obj->stack2);

}
```

```
// Get the front element of the queue
int myQueuePeek(MyQueue* obj) {
if (stackIsEmpty(obj->stack2)) {

while (!stackIsEmpty(obj->stack1)) {
            stackPush(obj->stack2, stackPop(obj->stack1));

        }

    }

    return stackPeek(obj->stack2);

}


// Check if the queue is empty

bool myQueueEmpty(MyQueue* obj) {

    return stackIsEmpty(obj->stack1) && stackIsEmpty(obj->stack2);

}


// Free the queue

void myQueueFree(MyQueue* obj) {
    free(obj->stack1->data);

    free(obj->stack1);
    free(obj->stack2->data);
    free(obj->stack2);
    free(obj);

}
```

`

Program 2: Design your implementation of the circular queue.

CODE:

```
typedef struct {
    int* data;      // Array to store queue elements
    int size;       // Maximum size of the queue
    int front;      // Index of the front element
    int rear;       // Index of the last element
} MyCircularQueue;

// Function declarations
MyCircularQueue* myCircularQueueCreate(int k);
bool myCircularQueueEnQueue(MyCircularQueue* obj, int value);
bool myCircularQueueDeQueue(MyCircularQueue* obj);
int myCircularQueueFront(MyCircularQueue* obj);
int myCircularQueueRear(MyCircularQueue* obj);
bool myCircularQueueIsEmpty(MyCircularQueue* obj);
bool myCircularQueueIsFull(MyCircularQueue* obj);
void myCircularQueueFree(MyCircularQueue* obj);
// Function definitions
MyCircularQueue* myCircularQueueCreate(int k) {
    MyCircularQueue* queue = (MyCircularQueue*)malloc(sizeof(MyCircularQueue));
    queue->data = (int*)malloc(sizeof(int) * k);
    queue->size = k;
    queue->front = -1;
    queue->rear = -1;
    return queue;
}
bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
    if (myCircularQueueIsFull(obj)) {
        return false;
    }
    if (myCircularQueueIsEmpty(obj)) {
        obj->front = 0;
    }
    obj->rear = (obj->rear + 1) % obj->size;
    obj->data[obj->rear] = value;
    return true;
}
bool myCircularQueueDeQueue(MyCircularQueue* obj) {
    if (myCircularQueueIsEmpty(obj)) {
```

`

```
        return false;
    }
    if (obj->front == obj->rear) {
        obj->front = -1;
        obj->rear = -1;
    } else {
        obj->front = (obj->front + 1) % obj->size;
    }
    return true;
}
int myCircularQueueFront(MyCircularQueue* obj) {
    if (myCircularQueueIsEmpty(obj)) {
        return -1;
    }
    return obj->data[obj->front];
}
int myCircularQueueRear(MyCircularQueue* obj) {
    if (myCircularQueueIsEmpty(obj)) {
        return -1;
    }
    return obj->data[obj->rear];
}
bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
    return obj->front == -1;
}
bool myCircularQueueIsFull(MyCircularQueue* obj) {
    return (obj->rear + 1) % obj->size == obj->front;
}


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

`

Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1

Input

```
["MyCircularQueue","enQueue","enQueue","enQueue","enQueue","Rear","isFull","deQueue","enQueue","Rear"]
```

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

Output

```
[null,true,true,true,false,3,true,true,true,4]
```

Expected

```
[null,true,true,true,false,3,true,true,true,4]
```

`

Program 3: Design your implementation of the circular double-ended queue (deque).

CODE:

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

typedef struct {
    int *data;

    int front;
    int rear;

    int capacity;
    int size;

MyCircularDeque* myCircularDequeCreate(int k) {

    MyCircularDeque* obj = (MyCircularDeque*)malloc(sizeof(MyCircularDeque));
    obj->data = (int*)malloc(k * sizeof(int));

    obj->front = 0;

    obj->rear = -1;
    obj->capacity = k;
    obj->size = 0;
    return obj;

}

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

    obj->size++;

    if (obj->rear == -1) obj->rear = obj->front;
    return true;

}

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

    obj->rear = (obj->rear + 1) % obj->capacity;
    obj->data[obj->rear] = value;

    obj->size++;

    if (obj->front == -1) obj->front = obj->rear;
    return true;

}

bool myCircularDequeDeleteFront(MyCircularDeque* obj) {
    if (obj->size == 0) return false;

    if (obj->size == 1) {
    obj->front = 0;

obj->rear = -1;
```

`

```
} else {
        obj->front = (obj->front + 1) % obj->capacity;
    }
    obj->size--;
    return true;
}
bool myCircularDequeDeleteLast(MyCircularDeque* obj) {
    if (obj->size == 0) return false;
    if (obj->size == 1) {
    obj->rear = -1;
obj->front = 0;
} else {
obj->rear = (obj->rear - 1 + obj->capacity) % obj->capacity;
    }
    obj->size--;
    return true;
}
int myCircularDequeGetFront(MyCircularDeque* obj) {
    return obj->size == 0 ? -1 : obj->data[obj->front];
}
int myCircularDequeGetRear(MyCircularDeque* obj) {
    return obj->size == 0 ? -1 : obj->data[obj->rear];
}
bool myCircularDequeIsEmpty(MyCircularDeque* obj) {
    return obj->size == 0;
}
bool myCircularDequeIsFull(MyCircularDeque* obj) {
    return obj->size == obj->capacity;
}
void myCircularDequeFree(MyCircularDeque* obj) {
    free(obj->data);
    free(obj);
```

`

}

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]
```

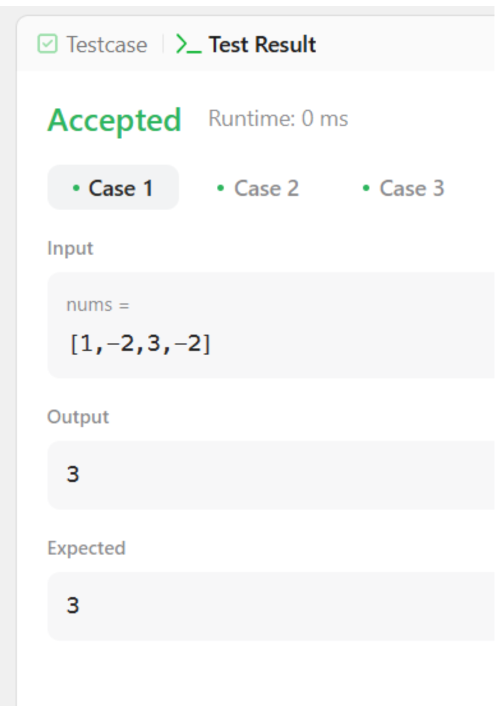
Expected

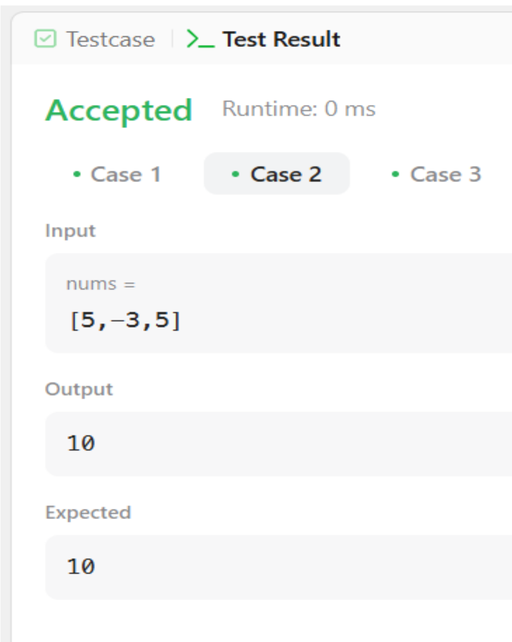
```
[null,true,true,true,false,2,true,true,true,4]
```

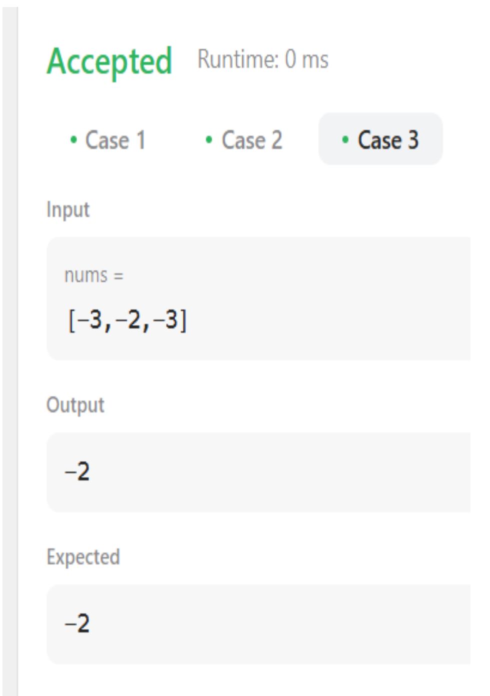
Program 4 : Maximum Sum Circular Subarray (medium),Given a circular integer array nums of length n, return the maximum possible sum of a non-empty subarray of nums.

CODE:

```
int maxSubarraySumCircular(int* nums, int numsSize) {
    int kadane(int* nums, int numsSize) {

        int maxSum = nums[0], currentSum = nums[0];
        for (int i = 1; i < numsSize; i++) {

            currentSum = currentSum > 0 ? currentSum + nums[i] : nums[i];
            maxSum = max(maxSum, currentSum);

        }

        return maxSum;

    }


    int maxNormal = kadane(nums, numsSize);
    int totalSum = 0;

    for (int i = 0; i < numsSize; i++) {
        totalSum += nums[i];

        nums[i] = -nums[i];

    }

    int minSubarraySum = kadane(nums, numsSize);
    int maxCircular = totalSum + minSubarraySum;


    if (maxCircular == 0) return maxNormal;
    return max(maxNormal, maxCircular);

}
```





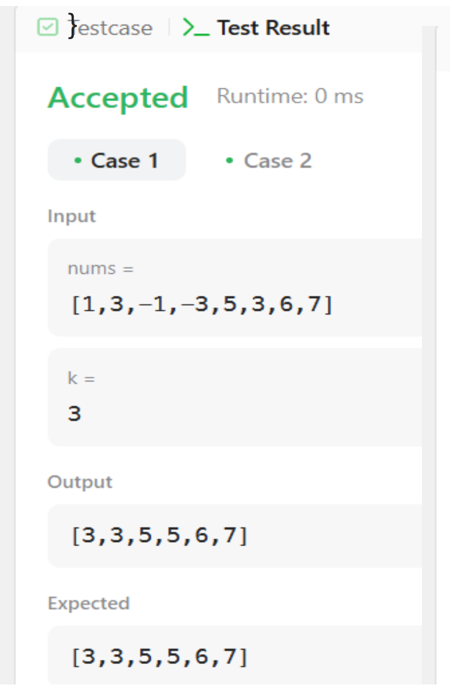


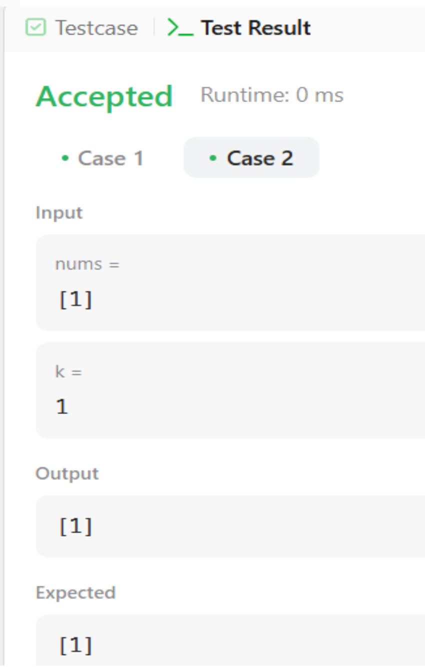
Program 5: Sliding Windows Maximum (hard)

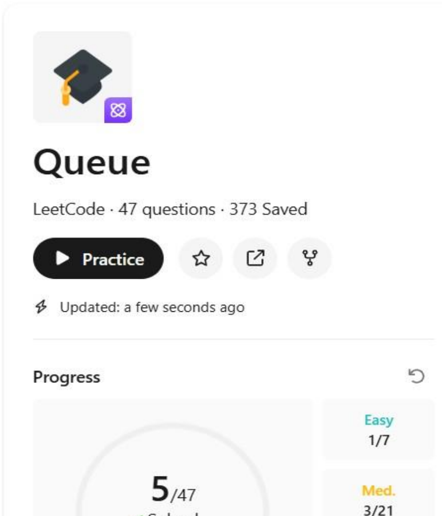
You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.Return the max sliding window.

CODE:

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

int* maxSlidingWindow(int* nums, int numsSize, int k, int* returnSize) {
    int* result = (int*)malloc((numsSize - k + 1) * sizeof(int));

    int* deque = (int*)malloc(numsSize * sizeof(int));
    int front = 0, rear = 0, idx = 0;


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

        while (front < rear && nums[deque[rear - 1]] <= nums[i])
            rear--;

        deque[rear++] = i;


        if (deque[front] <= i - k)
            front++;


        if (i >= k - 1)

            result[idx++] = nums[deque[front]];

    }


    *returnSize = numsSize - k + 1;
    free(deque);

    return result;
}
```





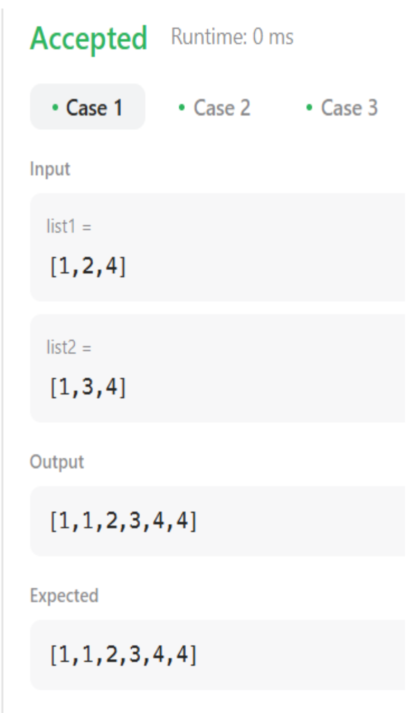


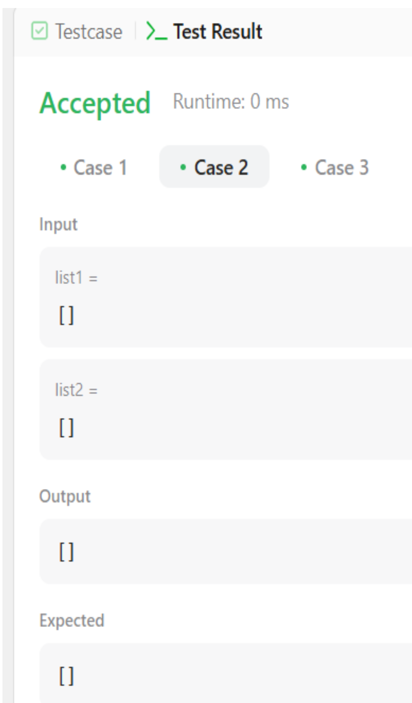
Linked List

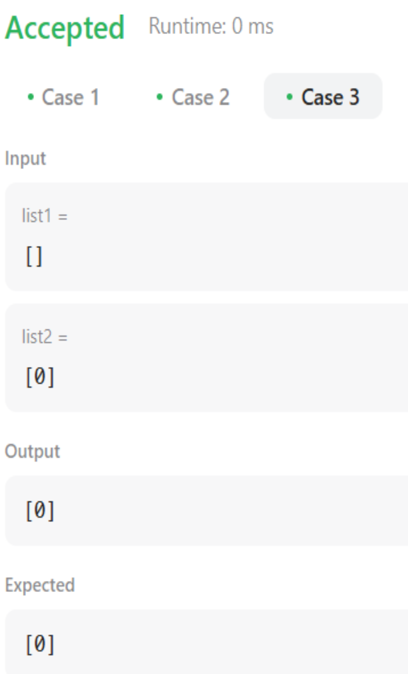
Program 1 : Merge Two Sorted List (Easy): You are given the heads of two sorted linked lists list1 and list2.Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

CODE:

```
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2) {
    struct ListNode head, *p = &head;

    while (l1 && l2) {

        if (l1->val <= l2->val) {
            p->next = l1;

            l1 = l1->next;

        } else {

            p->next = l2;
            l2 = l2->next;

        }

        p = p->next;

    }

    p->next = l1 ? l1 : l2;
    return head.next;

}
```



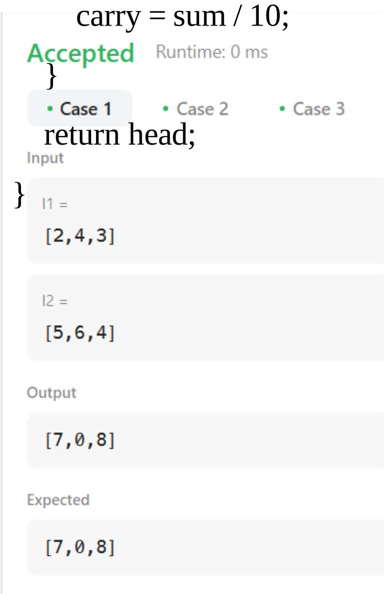


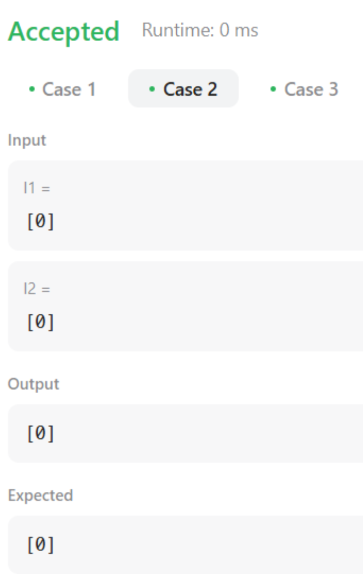


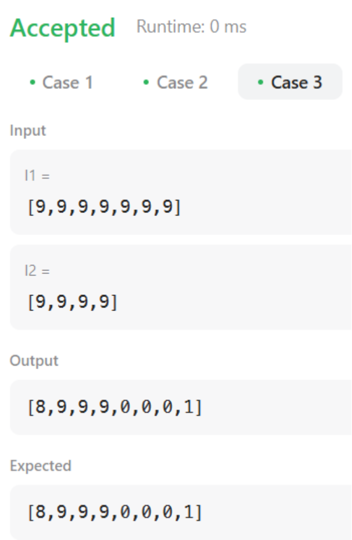
Program 2 : Add Two Numbers(medium),You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.You may assume the two numbers do not contain any leading zero, except the number 0 itself.

CODE:

```
struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2) {
    struct ListNode *head = NULL, *p = NULL;

    int carry = 0;


    while (l1 || l2 || carry) {
        int sum = carry;

        if (l1) {

            sum += l1->val;
            l1 = l1->next;

        }

        if (l2) {

            sum += l2->val;
            l2 = l2->next;

        }

        struct ListNode* new_node = (struct ListNode*)malloc(sizeof(struct ListNode));
        new_node->val = sum % 10;

        new_node->next = NULL;
        if (!head) {

            head = new_node;

        } else {

            p->next = new_node;

        }

        p = new_node;
        carry = sum / 10;
    }
    return head;
}
```



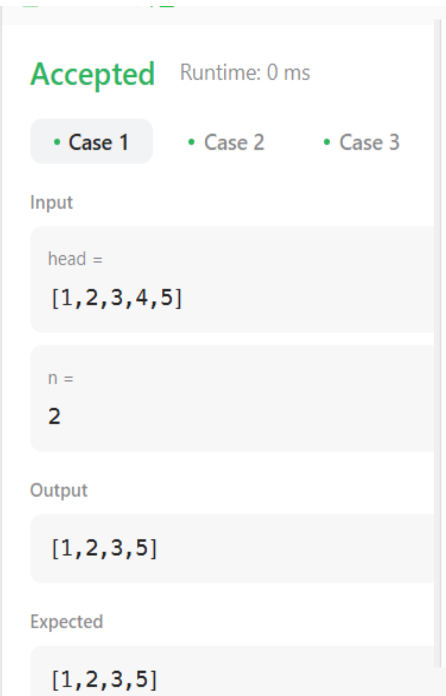


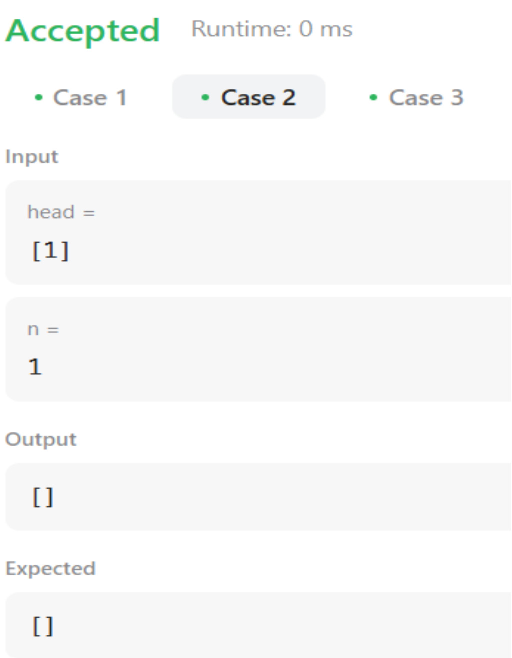


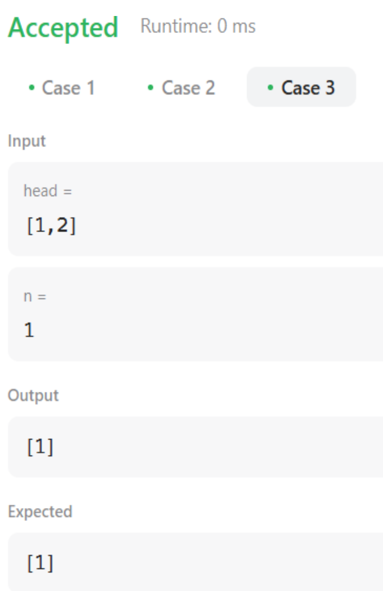
Program 3 : Remove Nth Node from End of List(medium) Given the head of a linked list, remove the nth node from the end of the list and return its head.

CODE:

```
struct ListNode* removeNthFromEnd(struct ListNode* head, int n) {
    struct ListNode *fast = head, *slow = head;

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

    }

    if (!fast) {

        struct ListNode* temp = head;
        head = head->next;
        free(temp);

        return head;

    }

    while (fast->next) {
        fast = fast->next;
        slow = slow->next;

    }

    struct ListNode* temp = slow->next;
    slow->next = slow->next->next;
    free(temp);

    return head;

}
```



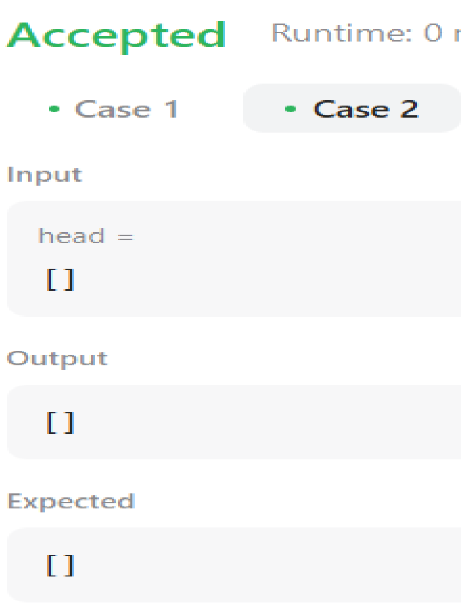


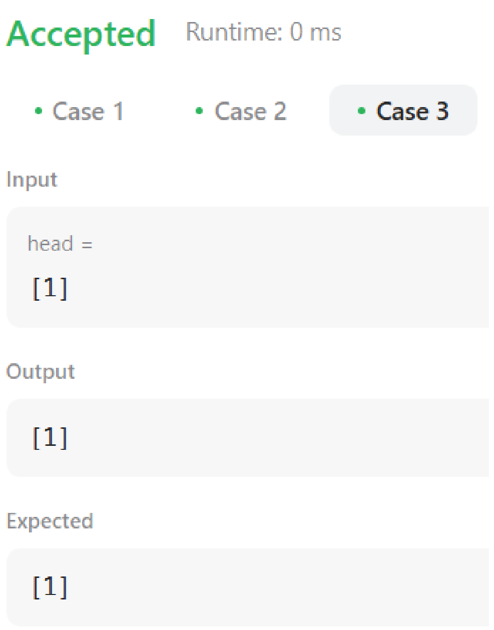


Program 4:Swap Nodes in Pairs (medium) Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

CODE:

```
struct ListNode* swapPairs(struct ListNode* head) {
    if (!head || !head->next) return head;

    struct ListNode *new_head = head->next;
    struct ListNode *prev = NULL;

    while (head && head->next) {
        struct ListNode *first = head;

        struct ListNode *second = head->next;
        first->next = second->next;

        second->next = first;
        if (prev) {
            prev->next = second;

        }

        prev = first;

        head = first->next;

    }

    return new_head; }
```





`

Program 5: Reverse Nodes in K Group (Hard) 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.

CODE:

```
struct ListNode* reverseKGroup(struct ListNode* head, int k) {

    struct ListNode *dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;

    struct ListNode *prev_group_end = dummy;


    while (1) {

        struct ListNode *kth_node = prev_group_end;
        for (int i = 0; i < k; i++) {

            kth_node = kth_node->next;

            if (!kth_node) return dummy->next;

        }

        struct ListNode *group_start = prev_group_end->next;
        struct ListNode *group_end = kth_node;

        struct ListNode *next_group_start = group_end->next;


        // Reverse the k nodes

        struct ListNode *prev = next_group_start, *curr = group_start;
        while (curr != next_group_start) {

            struct ListNode *next_node = curr->next;
            curr->next = prev;

            prev = curr;

            curr = next_node;

        }


        // Connect the reversed group to the previous and next groups
        prev_group_end->next = group_end;

        group_start->next = next_group_start;


        prev_group_end = group_start;

    }

}
```

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]

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]

Linked List

LeetCode · 79 questions · 1407 Saved

Practice

Updated: a few seconds ago

Progress

7/79

✓ Solved

0 Attempting

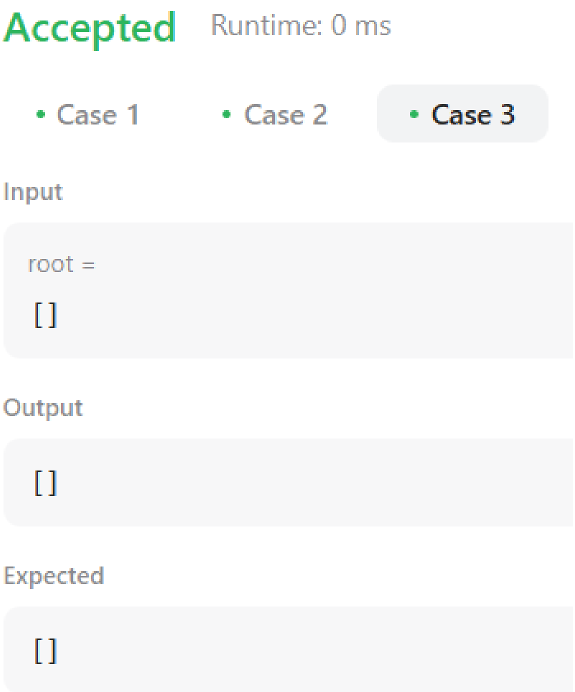
Easy	1/15
Med.	5/57
Hard	1/7

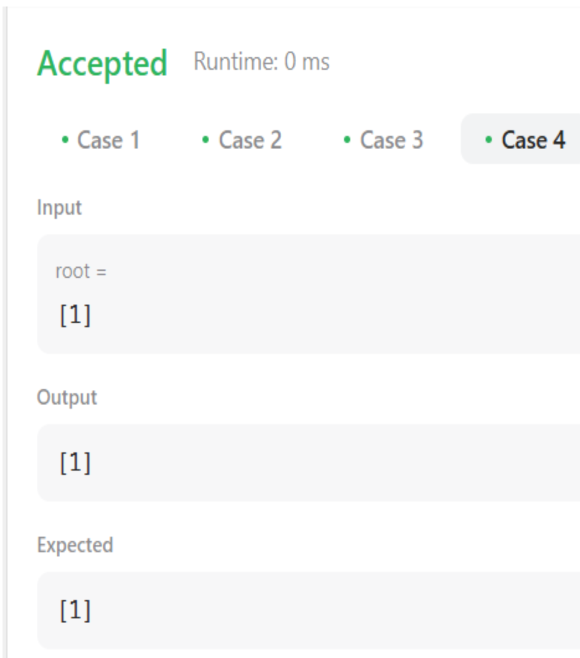
Trees

Program 1: Binary Tree Inorder Traversal (Easy) Given the root of a binary tree, return the inorder traversal of its nodes' values.

CODE:

```
void inorderTraversalHelper(struct TreeNode* root, int* returnSize, int* result) {
    if (!root) return;

    inorderTraversalHelper(root->left, returnSize, result);
    result[(*returnSize)++] = root->val;
    inorderTraversalHelper(root->right, returnSize, result);

}

int* inorderTraversal(struct TreeNode* root, int* returnSize) { int* result = (int*)malloc(1000 * sizeof(int));

    *returnSize = 0;

    inorderTraversalHelper(root, returnSize, result);
    return result;

}
```

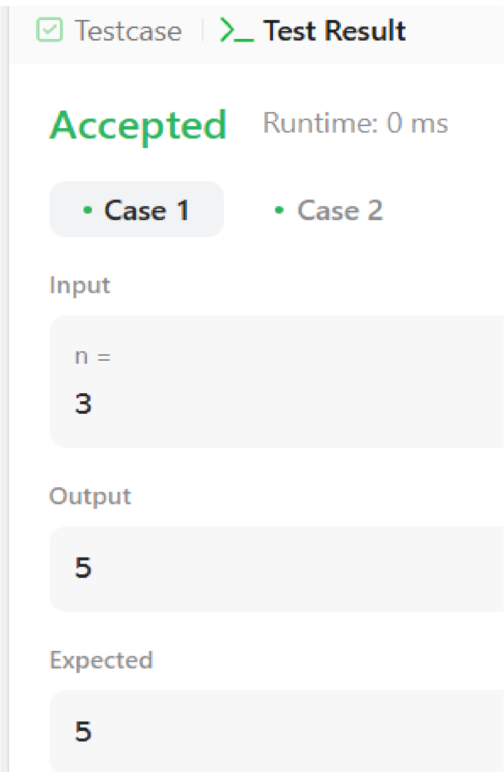


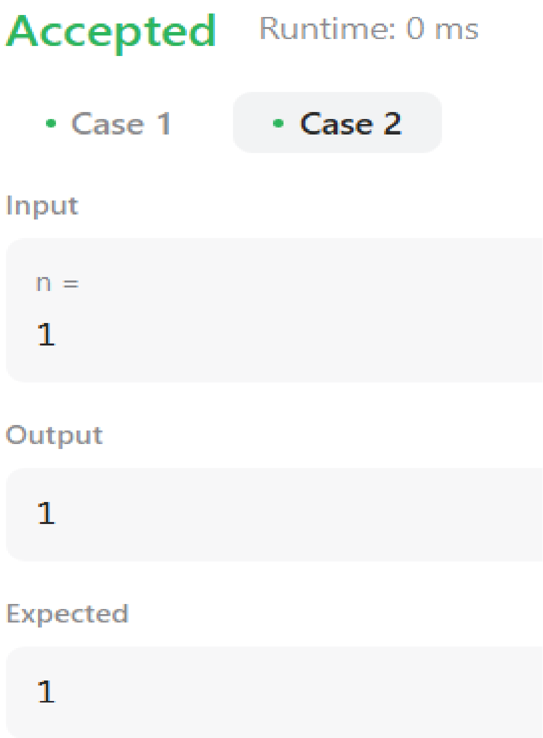


Program 2 : Unique Search Binary Trees (medium)Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.

CODE:

```
int numTrees(int n) {
    int dp[n + 1];
    dp[0] = dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = 0;

        for (int j = 1; j <= i; j++) {
            dp[i] += dp[j - 1] * dp[i - j];

        }}

    return dp[n];

}
```

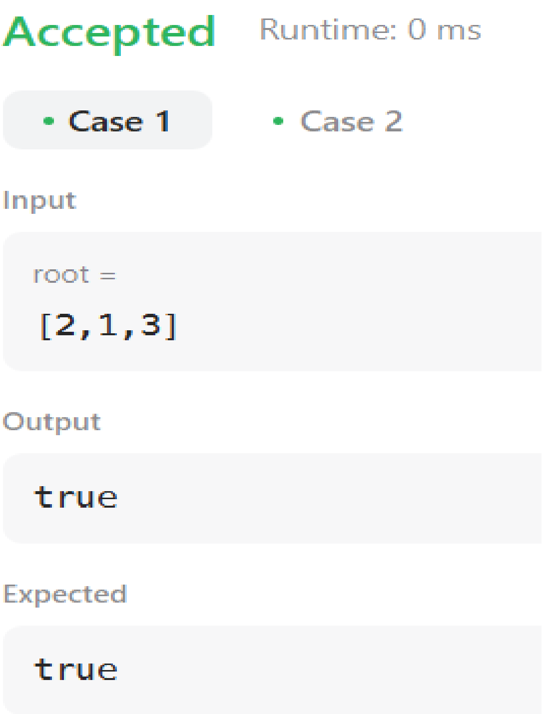


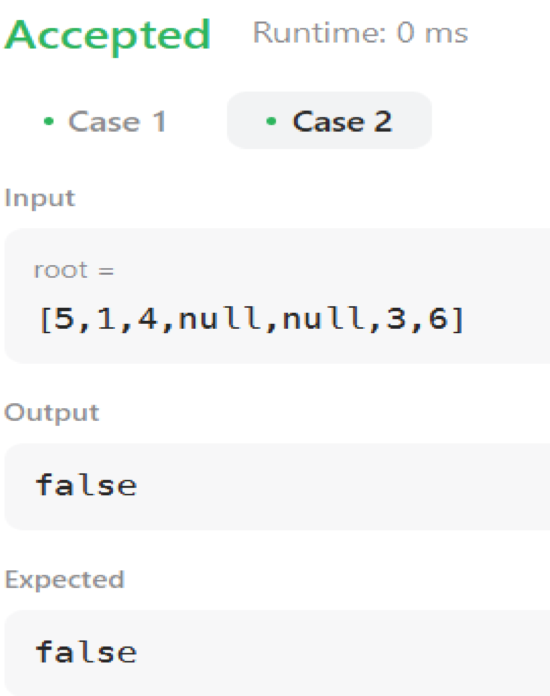


Program 3 : Validate Binary Search Tree (medium)Given the root of a binary tree, determine if it is a valid binary search tree (BST).

CODE:

```
int isValidBSTHelper(struct TreeNode* root, long long minVal, long long maxVal) {
    if (!root) return 1;

    if (root->val <= minVal || root->val >= maxVal) return 0;
    return isValidBSTHelper(root->left, minVal, root->val) &&

           isValidBSTHelper(root->right, root->val, maxVal);

}


int isValidBST(struct TreeNode* root) {

    return isValidBSTHelper(root, LONG_MIN, LONG_MAX);

}
```

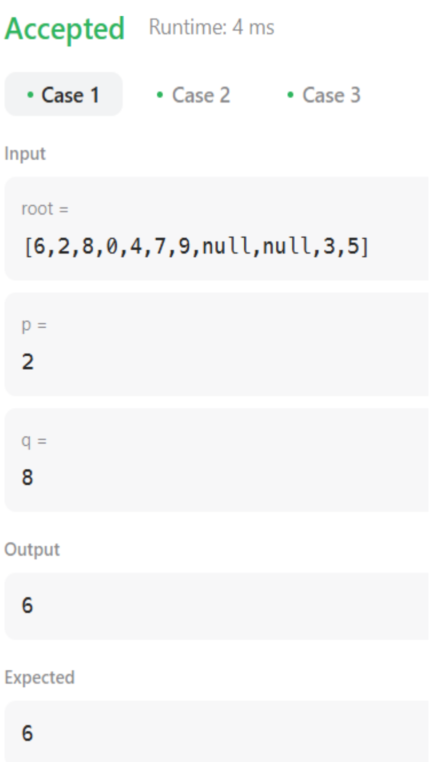


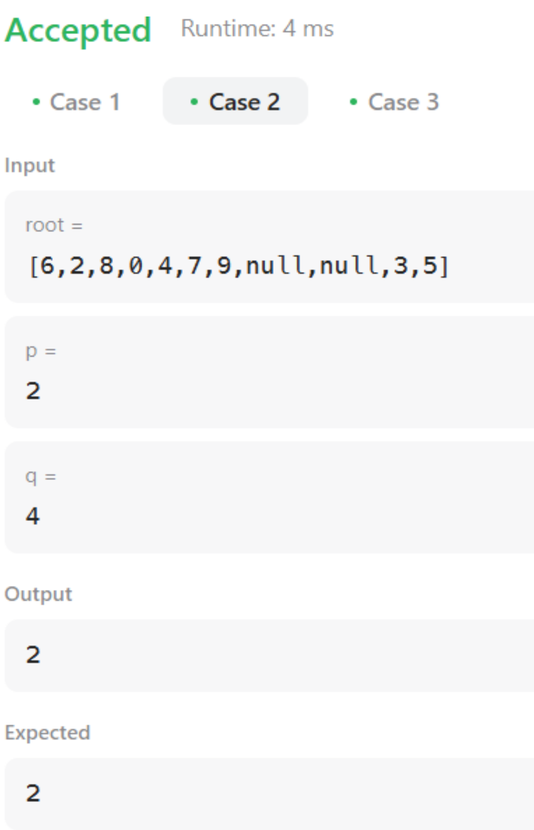


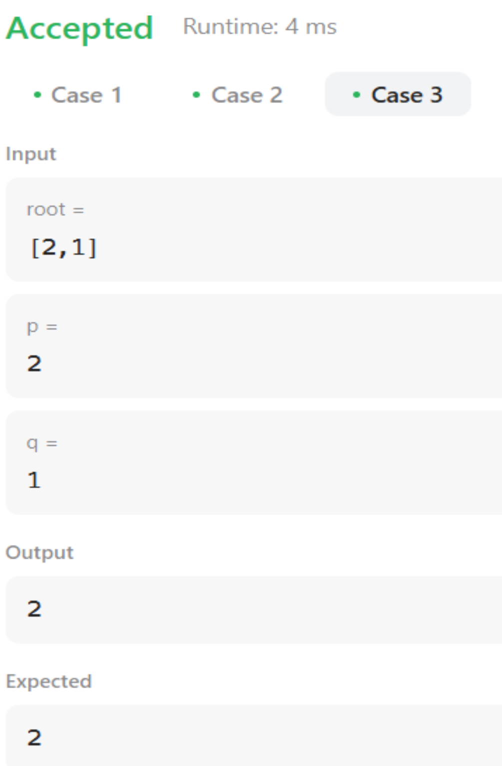
Program 4 Lowest Common Ancestor of a Binary Tree (medium) Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

CODE:

```
struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct
TreeNode* q) {

    while (root != NULL) {

        if (p->val < root->val && q->val < root->val) {
            root = root->left;

        } else if (p->val > root->val && q->val > root->val) {
            root = root->right;

        } else {

            return root;

        }

    }

    return NULL;

}
```







`

Program 5 Frog Position after T seconds (Hard)

Given an undirected tree consisting of n vertices numbered from 1 to n. A frog starts jumping from vertex 1. In one second, the frog jumps from its current vertex to another unvisited vertex if they are directly connected. The frog can not jump back to a visited vertex. In case the frog can jump to several vertices, it jumps randomly to one of them with the same probability. Otherwise, when the frog can not jump to any unvisited vertex, it jumps forever on the same vertex.

The edges of the undirected tree are given in the array edges, where edges[i] = [ai, bi] means that exists an edge connecting the vertices ai and bi.

Return the probability that after t seconds the frog is on the vertex target. Answers within 10-5 of the actual answer will be accepted.

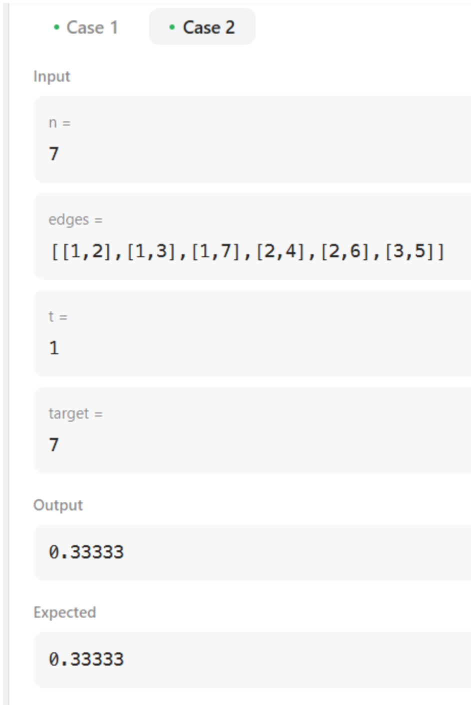
CODE:

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


typedef struct {
    int *vertices;
    int size;

} Graph;


Graph* createGraph(int n) {

    Graph *graph = (Graph *)malloc(n * sizeof(Graph));
    for (int i = 0; i < n; i++) {

        graph[i].vertices = (int *)malloc(n * sizeof(int));
        graph[i].size = 0;

    }

    return graph;

}


void addEdge(Graph *graph, int u, int v) {
    graph[u].vertices[graph[u].size++] = v;
    graph[v].vertices[graph[v].size++] = u;

}


double dfs(int node, int parent, int t, int target, Graph *graph, int *visited) {
    if (t == 0) {

            return node == target ? 1.0 : 0.0;

    }
```

`

```
    visited[node] = 1;
    double probability = 0.0;
    int unvisitedCount = 0;


        for (int i = 0; i < graph[node].size; i++) {
        int neighbor = graph[node].vertices[i];

        if (!visited[neighbor]) {
            unvisitedCount++;

        }

    }

       if (unvisitedCount == 0) {
        visited[node] = 0;

        return 0.0;

    }

        for (int i = 0; i < graph[node].size; i++) {
        int neighbor = graph[node].vertices[i];

        if (!visited[neighbor]) {

            probability += dfs(neighbor, node, t - 1, target, graph, visited) / unvisitedCount;

        }

    }

    visited[node] = 0;
    return probability;

}

double frogPosition(int n, int** edges, int edgesSize, int* edgesColSize, int t, int target) {
    Graph *graph = createGraph(n);

    for (int i = 0; i < edgesSize; i++) {
        int u = edges[i][0] - 1;

        int v = edges[i][1] - 1;
        addEdge(graph, u, v);

    }

    int *visited = (int *)calloc(n, sizeof(int));
```

```
    // Start DFS from node 0 (vertex 1 in the problem statement) with t seconds left
    double result = dfs(0, -1, t, target - 1, graph, visited);


    free(visited);

    for (int i = 0; i < n; i++) {
        free(graph[i].vertices);

    }

    free(graph);


    return result;

}
```



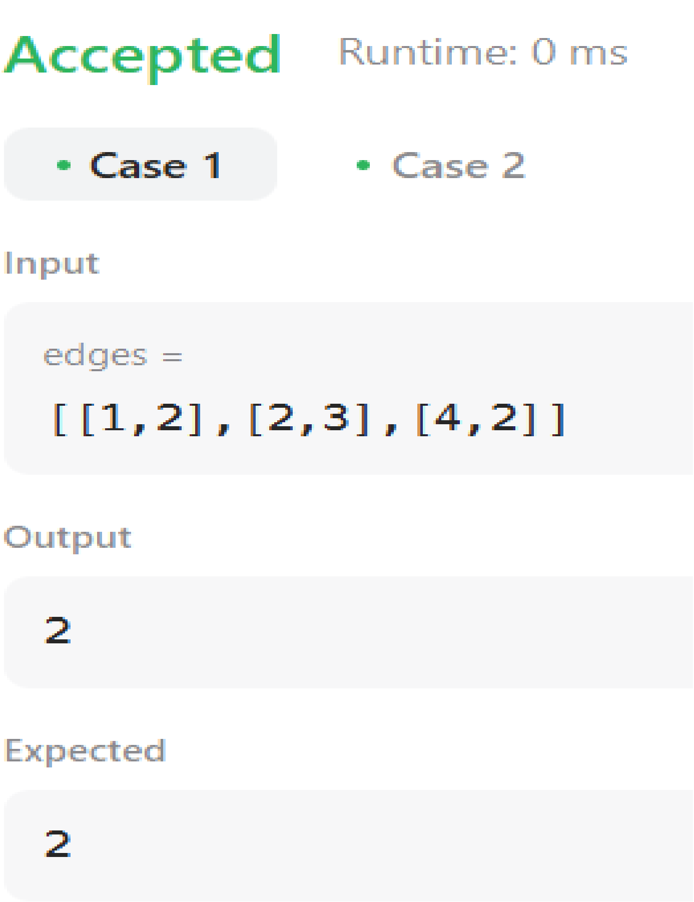
Graphs

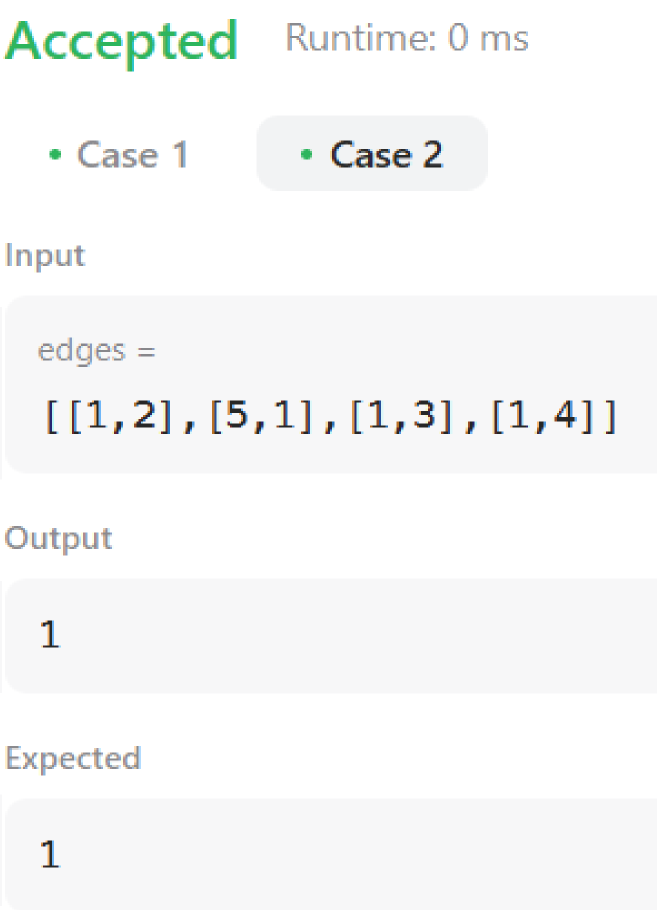
Program 1 Find Centre of Star Graph

There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node. You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.

CODE:

```
int findCenter(int** edges, int edgesSize, int* edgesColSize) {
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    } else {
        return edges[0][1];
    }
}
```





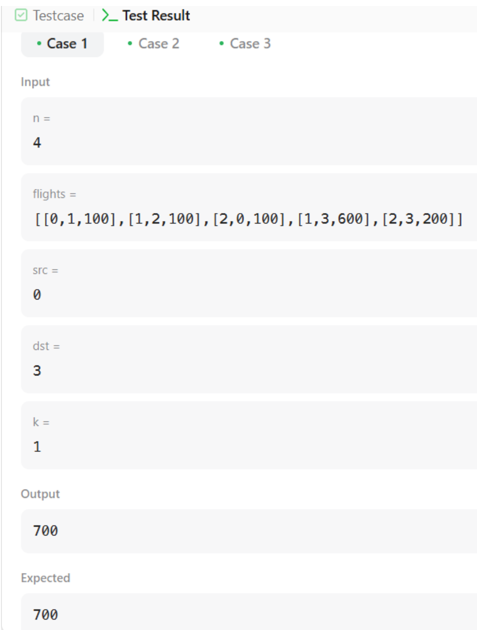
`

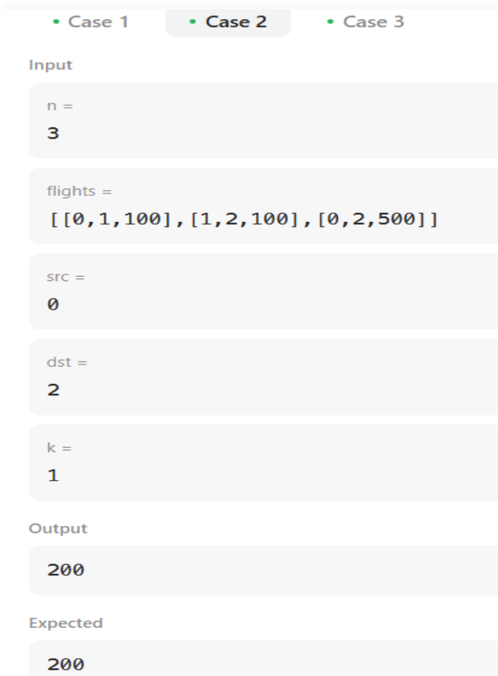
Program 2 Cheapest Flights within K stops (Medium) There are n cities connected by some number of flights. You are given an array flights where flights[i] = [fromi, toi, pricei] indicates that there is a flight from city fromi to city toi with cost price i. You are also given three integers src, dst, and k, return the cheapest price from src to dst with at most k stops. If there is no such route, return -1.

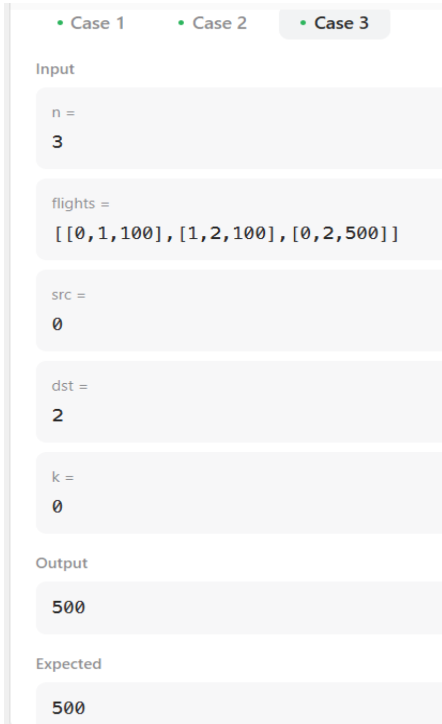
CODE:

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


int findCheapestPrice(int n, int** flights, int flightsSize, int* flightsColSize, int src, int dst, int k) {
    int* costs = (int*)malloc(n * sizeof(int));

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


    for (int i = 0; i < n; i++) {
        costs[i] = INT_MAX;

    }

    costs[src] = 0;


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

            tempCosts[j] = costs[j];

        }


        for (int j = 0; j < flightsSize; j++) {
            int from = flights[j][0];

            int to = flights[j][1];
            int price = flights[j][2];


            if (costs[from] != INT_MAX && costs[from] + price < tempCosts[to]) {
                tempCosts[to] = costs[from] + price;

            }

        }
```

```
        for (int j = 0; j < n; j++) {
            costs[j] = tempCosts[j];
        }
    }

    int result = costs[dst] == INT_MAX ? -1 : costs[dst];
    free(costs);
    free(tempCosts);
    return result;
}
```





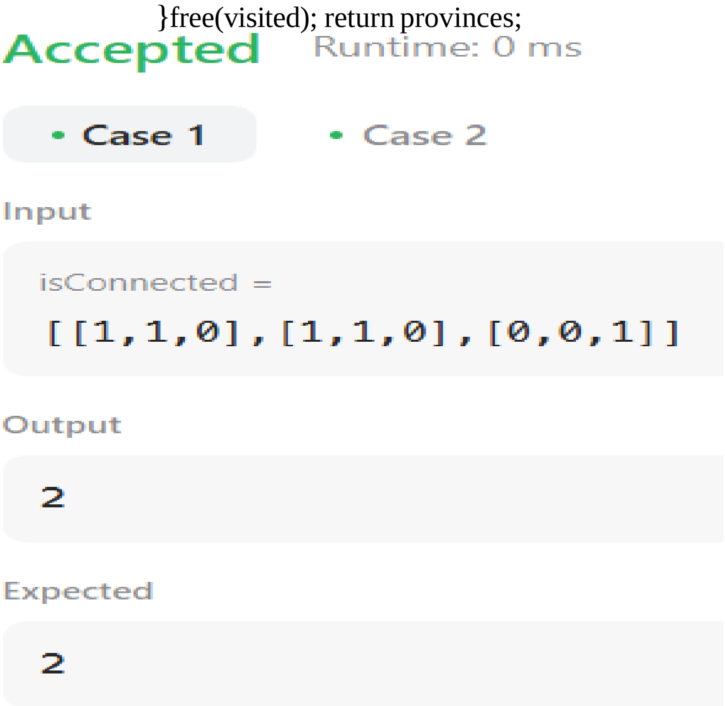


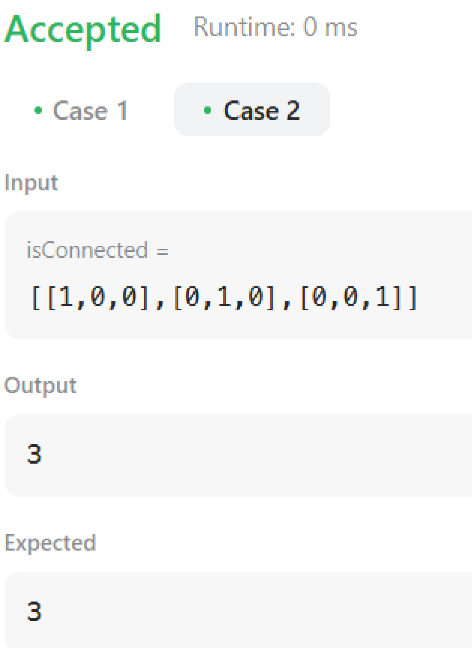
Program 3 Number of Provinces (medium)

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city.A province is a group of directly or indirectly connected cities and no other cities outside of the group.You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.Return the total number of provinces

CODE:

```
 #include <stdbool.h>

void dfs(int** isConnected, int isConnectedSize, int* visited, int city) {
    visited[city] = 1;

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

        if (isConnected[city][i] == 1 && !visited[i]) {
            dfs(isConnected, isConnectedSize, visited, i);

        }

    }

}


int findCircleNum(int** isConnected, int isConnectedSize, int* isConnectedColSize) {
    int* visited = (int*)calloc(isConnectedSize, sizeof(int));
    int provinces = 0;

    for (int i = 0; i < isConnectedSize; i++) {
        if (!visited[i]) {

            dfs(isConnected, isConnectedSize, visited, i);
            provinces++;

        }free(visited); return provinces;
```





`

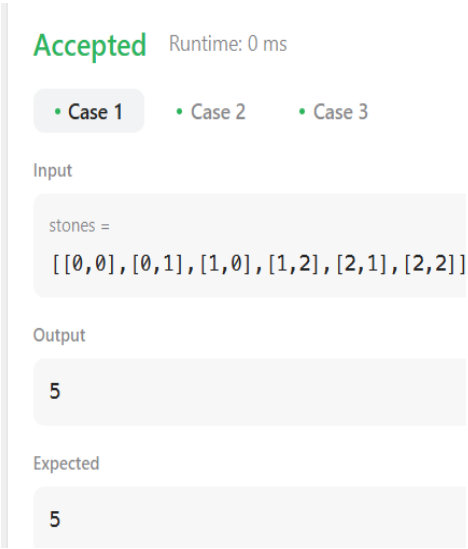
Program 4 Most stones removed with same row or column (medium)

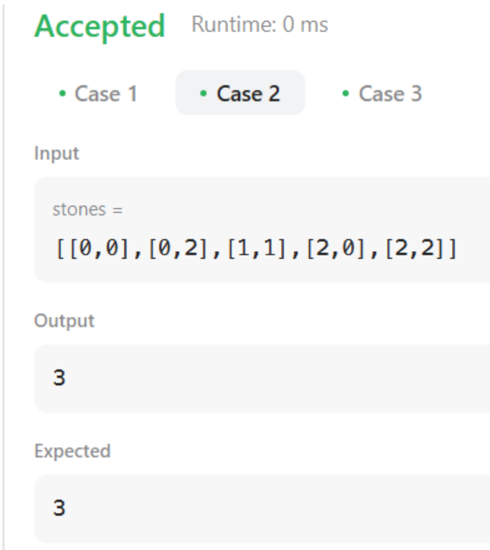
On a 2D plane, we place n stones at some integer coordinate points. Each coordinate point may have at most one stone. A stone can be removed if it shares either the same row or the same column as another stone that has not been removed.Given an array stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return the largest possible number of stones that can be removed.

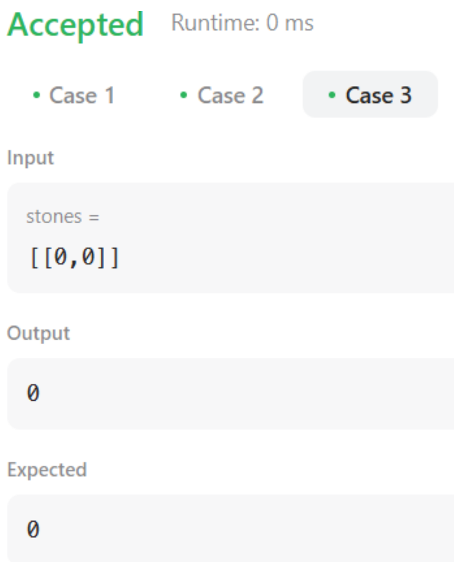
CODE:

```
#include <stdlib.h>

void dfs(int node, int** graph, int* visited, int graphSize) {
    visited[node] = 1;

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

        if (graph[node][i] && !visited[i]) {
            dfs(i, graph, visited, graphSize);

        }}

}

int removeStones(int** stones, int stonesSize, int* stonesColSize) { int** graph = (int**)malloc(stonesSize * sizeof(int*));

    for (int i = 0; i < stonesSize; i++) {
        graph[i] = (int*)calloc(stonesSize, sizeof(int));

    }

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

        for (int j = i + 1; j < stonesSize; j++) {

            if (stones[i][0] == stones[j][0] || stones[i][1] == stones[j][1]) {
                graph[i][j] = 1;

                graph[j][i] = 1;

            }}}

    int* visited = (int*)calloc(stonesSize, sizeof(int));
    int numOfConnectedComponents = 0;

    for (int i = 0; i < stonesSize; i++) {
        if (!visited[i]) {

            dfs(i, graph, visited, stonesSize);
            numOfConnectedComponents++;

        }}

    for (int i = 0; i < stonesSize; i++) {
        free(graph[i]);

    }free(graph);

    free(visited);

    return stonesSize - numOfConnectedComponents;
```

```
}
```


Accepted Runtime: 0 ms
• Case 1 • Case 2 • Case 3
Input
stones =
[[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
Output
5
Expected
5


Accepted Runtime: 0 ms
• Case 1 • Case 2 • Case 3
Input
stones =
[[0,0],[0,2],[1,1],[2,0],[2,2]]
Output
3
Expected
3


Accepted Runtime: 0 ms
• Case 1 • Case 2 • Case 3
Input
stones =
[[0,0]]
Output
0
Expected
0

`

Program 5 Minimize Malware Spread (Hard)

You are given a network of n nodes represented as an n x n adjacency matrix graph, where the ith node is directly connected to the jth node if graph[i][j] == 1.Some nodes initial are initially infected by malware. Whenever two nodes are directly connected, and at least one of those two nodes is infected by malware, both nodes will be infected by malware. This spread of malware will continue until no more nodes can be infected in this manner.Suppose M(initial) is the final number of nodes infected with malware in the entire network after the spread of malware stops. We will remove exactly one node from initial.Return the node that, if removed, would minimize M(initial). If multiple nodes could be removed to minimize M(initial), return such a node with the smallest index.Note that if a node was removed from the initial list of infected nodes, it might still be infected later due to the malware spread.

CODE:

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

void dfs(int** graph, int graphSize, int node, int* visited) {
    visited[node] = 1;

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

        if (graph[node][i] == 1 && !visited[i]) {
            dfs(graph, graphSize, i, visited);

        }}

}

int countInfected(int** graph, int graphSize, int* initial, int initialSize, int removeIdx) {
    int* visited = (int*)calloc(graphSize, sizeof(int));

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

        if (i != removeIdx && !visited[initial[i]]) {
            dfs(graph, graphSize, initial[i], visited);

        }}

    int count = 0;

    for (int i = 0; i < graphSize; i++) {
        if (visited[i]) count++;

    }free(visited);

     return count;

}

int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);

}

int minMalwareSpread(int** graph, int graphSize, int* graphColSize, int* initial, int initialSize) {
    qsort(initial, initialSize, sizeof(int), compare);

    int minSpread = graphSize + 1;
    int bestNode = initial[0];
```

```
for (int i = 0; i < initialSize; i++) {
    int spread = countInfected(graph, graphSize, initial, initialSize, i);
    if (spread < minSpread || (spread == minSpread && initial[i] < bestNode)) {
        minSpread = spread;
        bestNode = initial[i];
    }
}
return bestNode;
}
```