NAME:NEER AWASTHI

CLASS:A4 B2 20

# Task-1 submission

Problem:Array DS

Task is to reverse an array

Function:

int\* reverseArray(int a\_count, int\* a, int\* result\_count) {

\*result\_count = a\_count;

int\* reversed = malloc(a\_count \* sizeof(int));

if (reversed == NULL) {

\*result\_count = 0;

return NULL;

}

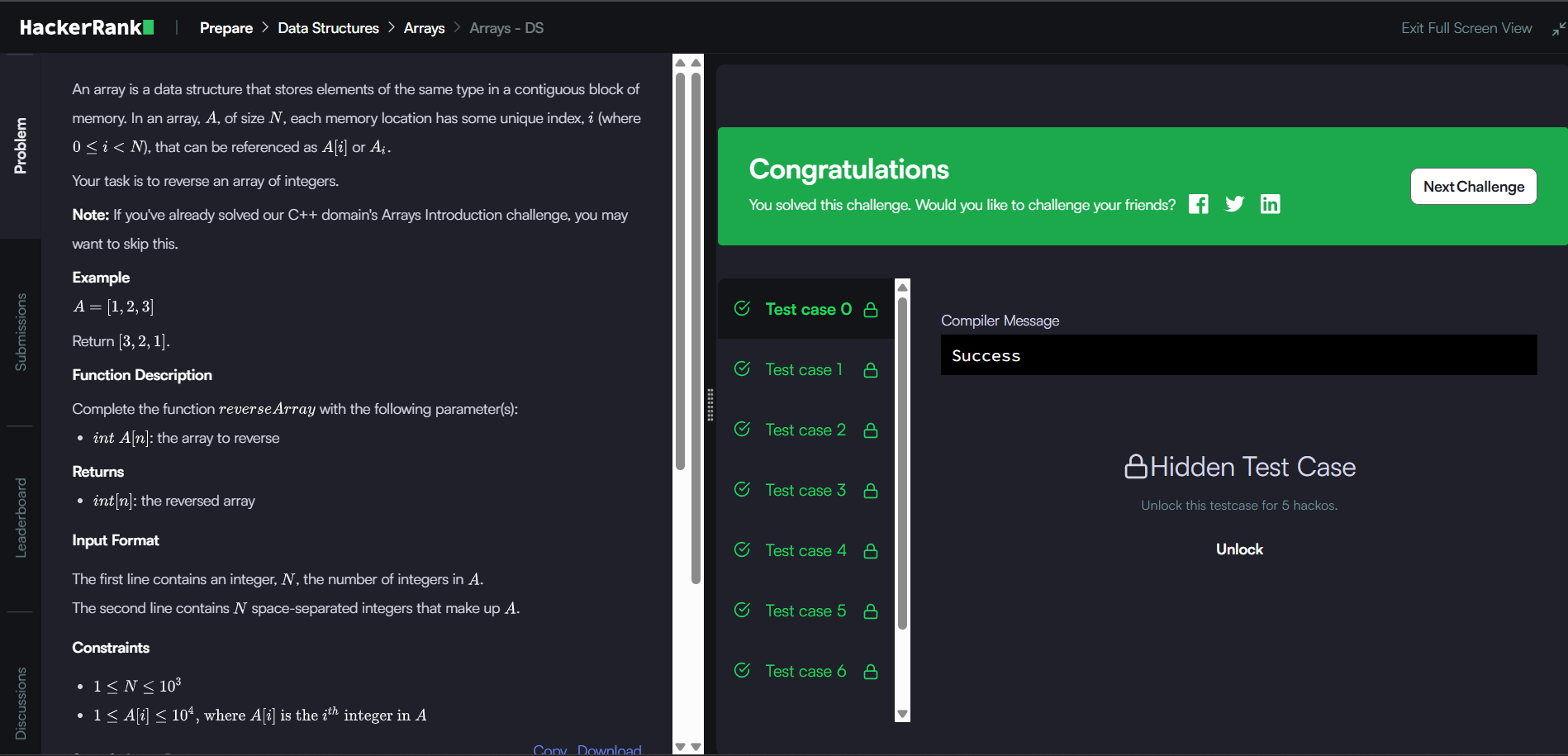
for (int i = 0; i < a\_count; i++) {

reversed[i] = a[a\_count - 1 - i];

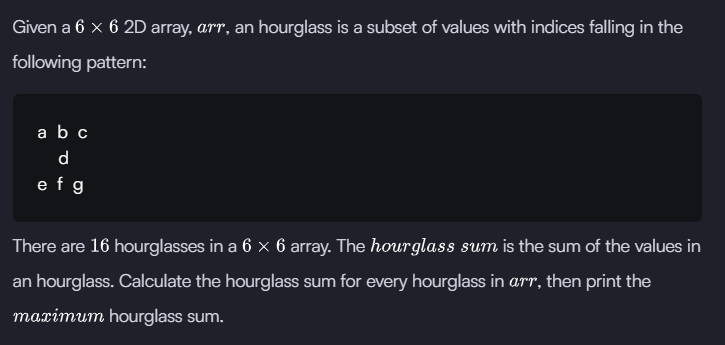
}

return reversed;

}



Problem:2 D Array DS



Function:

int hourglassSum(int arr\_rows, int arr\_columns, int\*\* arr) {

int max\_sum = INT\_MIN; // Initialize to smallest possible integer

for (int i = 0; i <= arr\_rows - 3; i++) {

for (int j = 0; j <= arr\_columns - 3; j++) {

int current\_sum =

arr[i][j] + arr[i][j+1] + arr[i][j+2] +

arr[i+1][j+1] +

arr[i+2][j] + arr[i+2][j+1] + arr[i+2][j+2];

if (current\_sum > max\_sum) {

max\_sum = current\_sum;

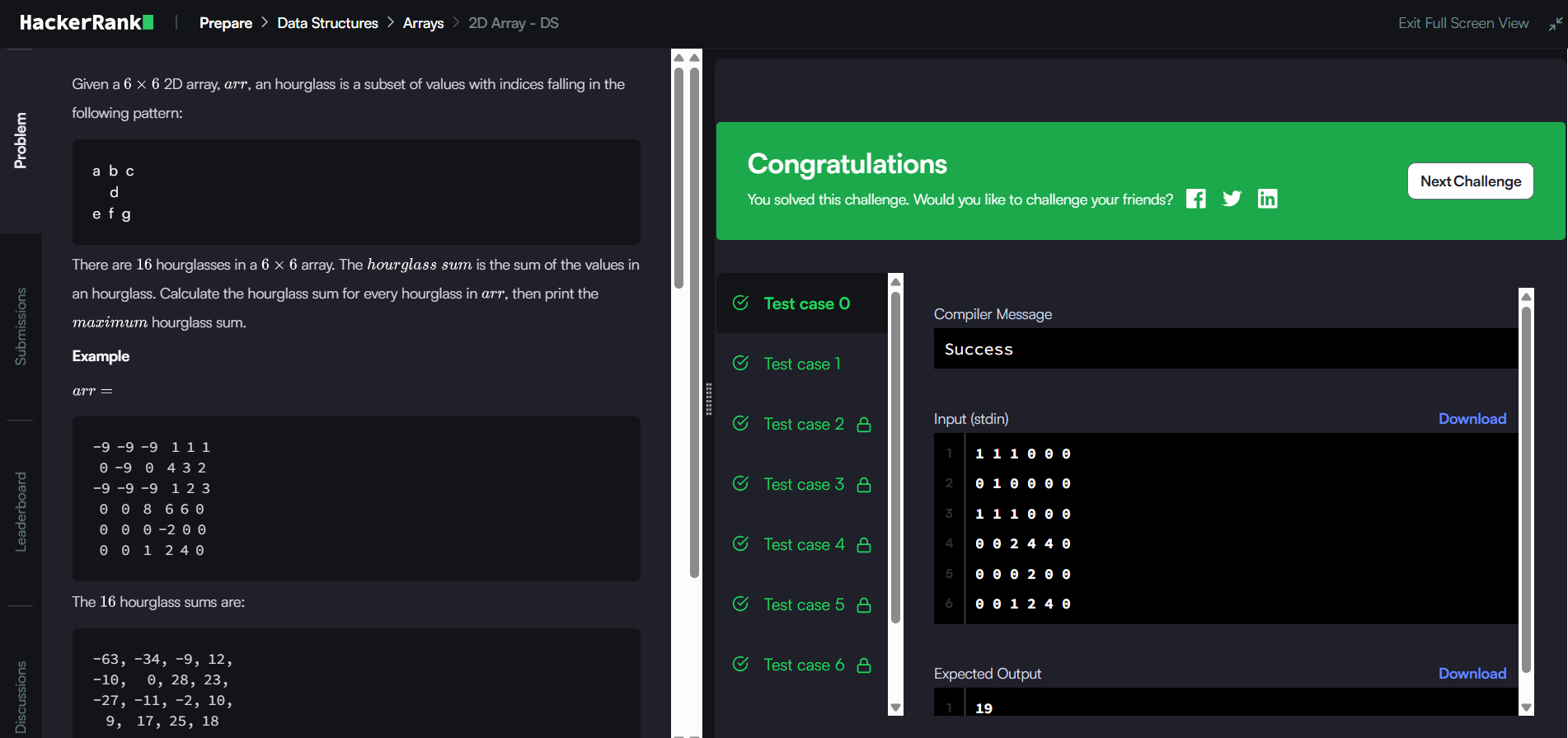
}

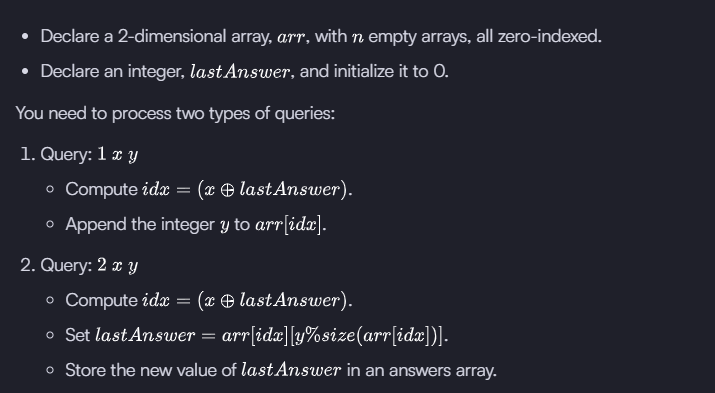
}

}

return max\_sum;

}



Problem:Dynamic Array  


Function:

int\* dynamicArray(int n, int queries\_rows, int queries\_columns, int\*\* queries, int\* result\_count) {

// Initialize n empty arrays

int\*\* seqList = malloc(n \* sizeof(int\*));

int\* seqSize = malloc(n \* sizeof(int)); // Store current sizes

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

seqList[i] = NULL;

seqSize[i] = 0;

}

int\* answers = malloc(queries\_rows \* sizeof(int));

int ansCount = 0;

int lastAnswer = 0;

for (int i = 0; i < queries\_rows; i++) {

int qType = queries[i][0];

int x = queries[i][1];

int y = queries[i][2];

int idx = (x ^ lastAnswer) % n;

if (qType == 1) {

// Append y to seqList[idx]

seqList[idx] = realloc(seqList[idx], (seqSize[idx] + 1) \* sizeof(int));

seqList[idx][seqSize[idx]] = y;

seqSize[idx]++;

} else if (qType == 2) {

// Get value at index y % size

int value = seqList[idx][y % seqSize[idx]];

lastAnswer = value;

answers[ansCount++] = lastAnswer;

}

}

\*result\_count = ansCount;

// Free the allocated sequences (optional)

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

free(seqList[i]);

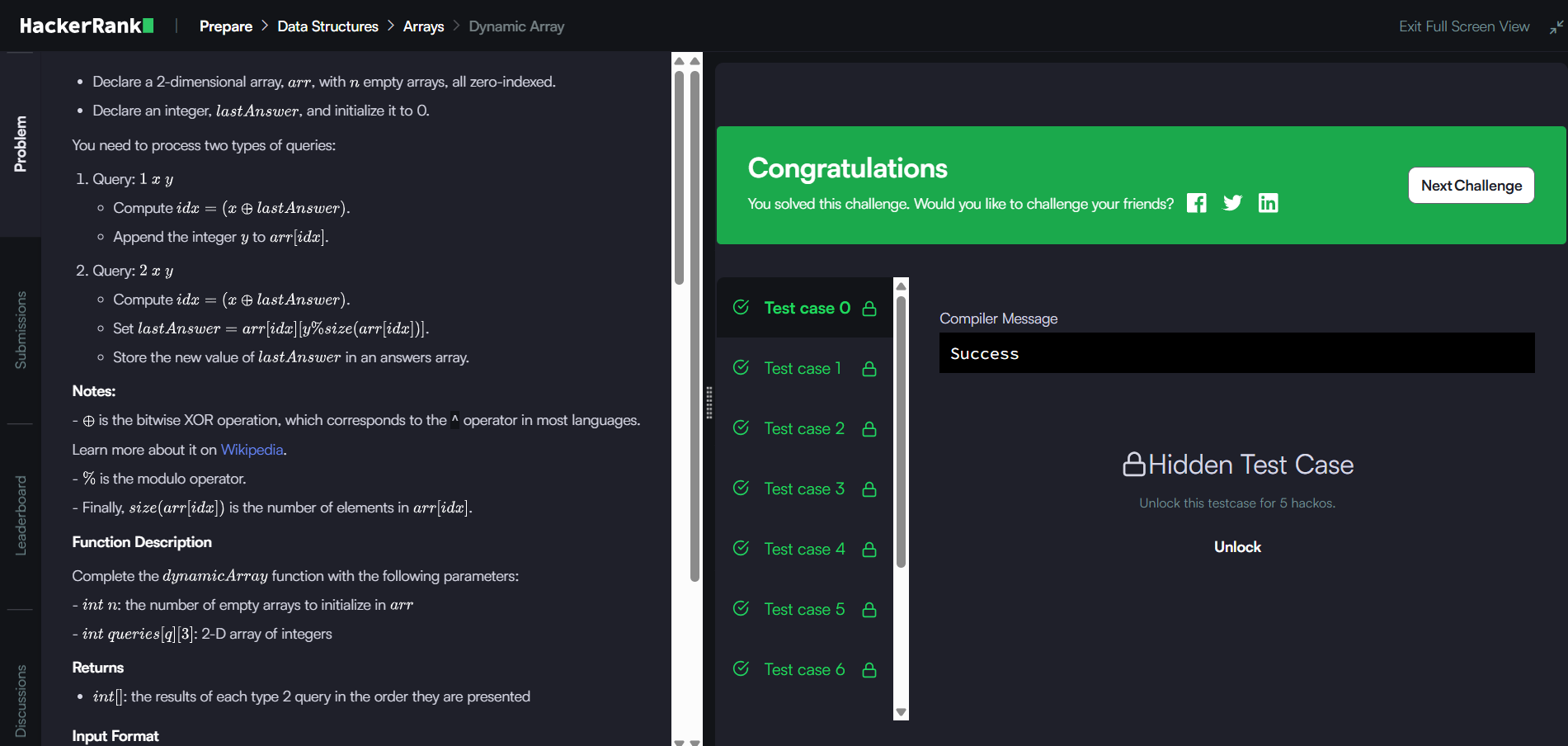
}

free(seqList);

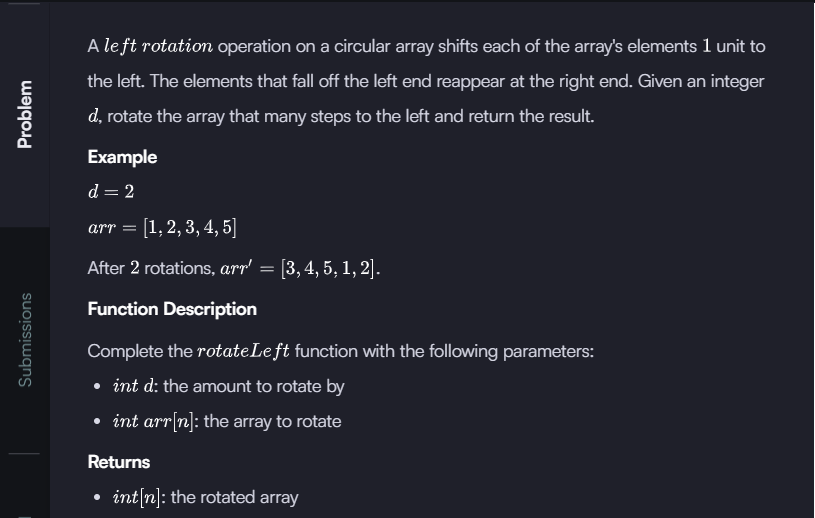
free(seqSize);

return answers;

}



Problem:Left Rotation



Function:

int\* rotateLeft(int d, int arr\_count, int\* arr, int\* result\_count) {

\*result\_count = arr\_count;

int\* rotated = malloc(arr\_count \* sizeof(int));

if (rotated == NULL) {

\*result\_count = 0;

return NULL;

}

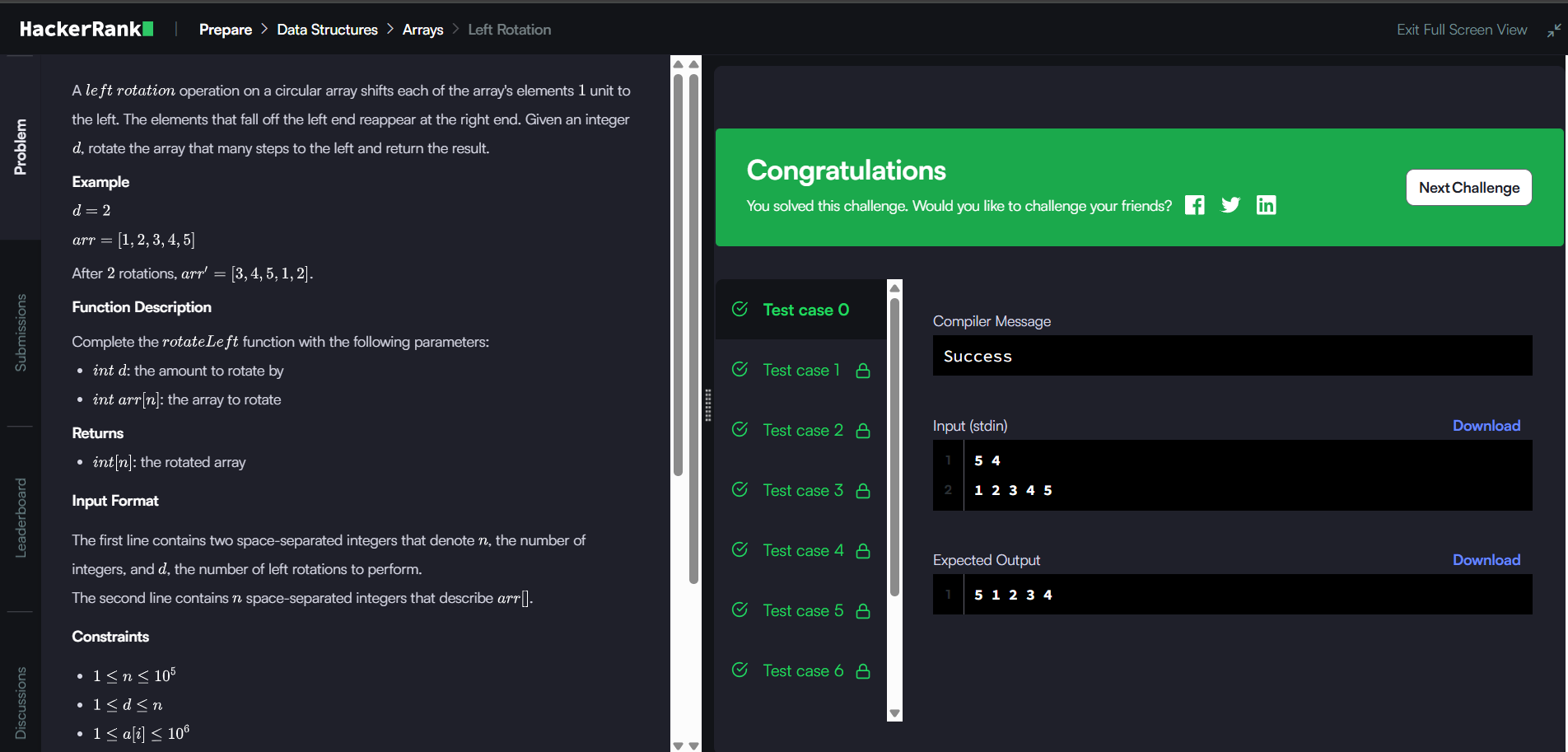
for (int i = 0; i < arr\_count; i++) {

rotated[i] = arr[(i + d) % arr\_count];

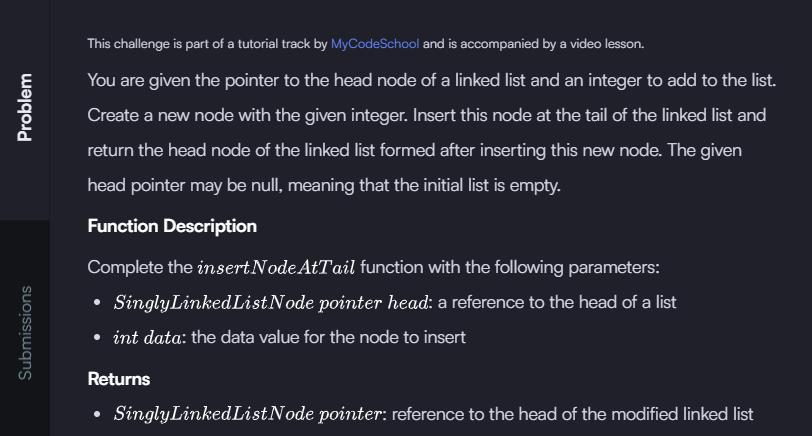
}

return rotated;

}



Problem:Insert a Node at the Tail of a Linked List



Function:

SinglyLinkedListNode\* insertNodeAtTail(SinglyLinkedListNode\* head, int data) {

SinglyLinkedListNode\* new\_node = malloc(sizeof(SinglyLinkedListNode));

new\_node->data = data;

new\_node->next = NULL;

// If the list is empty, new node becomes the head

if (head == NULL) {

return new\_node;

}

// Otherwise, traverse to the end of the list

SinglyLinkedListNode\* current = head;

while (current->next != NULL) {

current = current->next;

}

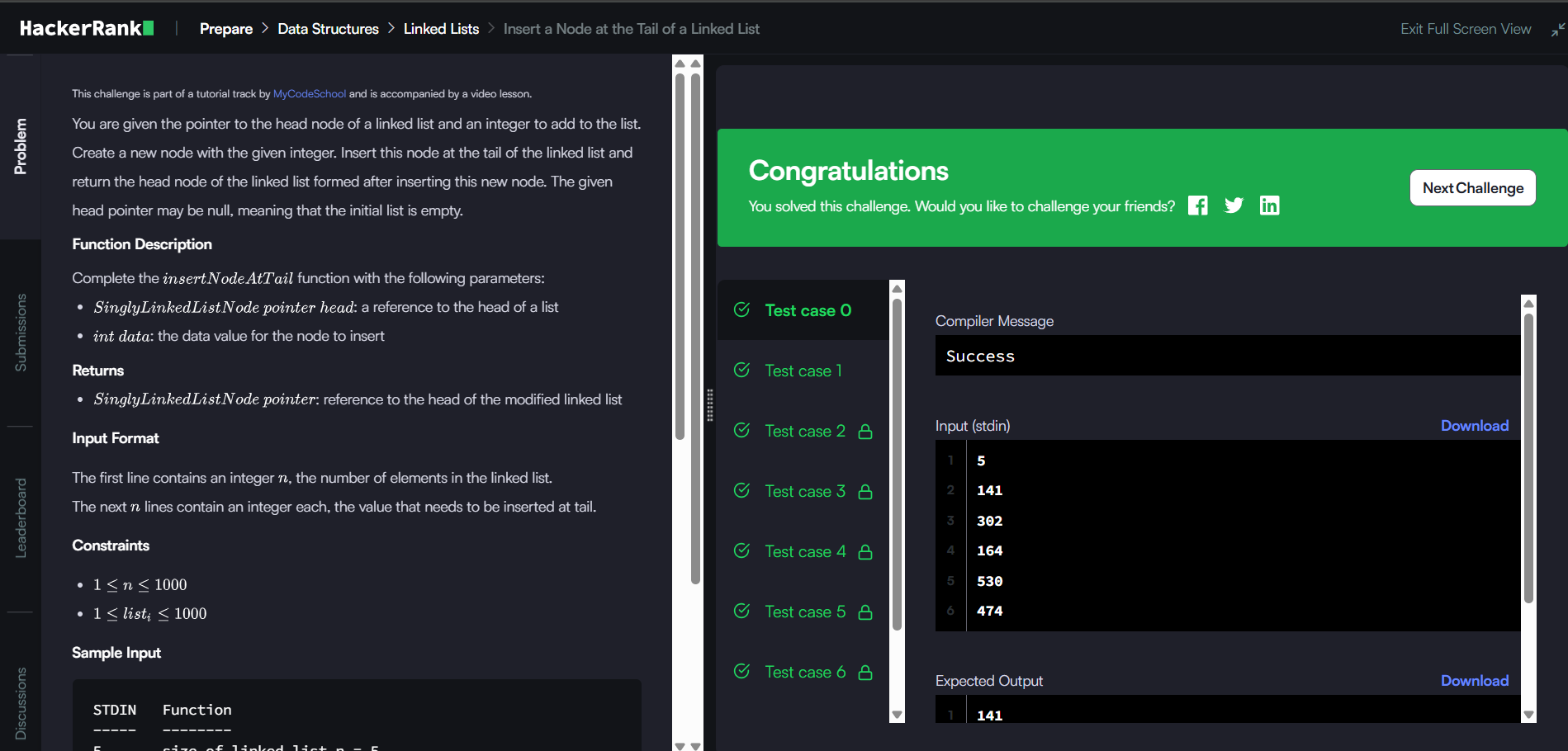
// Append the new node at the tail

current->next = new\_node;

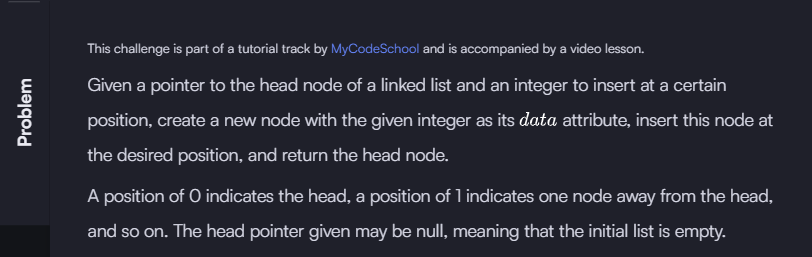
// Return the head of the list

return head;

}



Problem:



SinglyLinkedListNode\* insertNodeAtPosition(SinglyLinkedListNode\* llist, int data, int position) {

// Create new node

SinglyLinkedListNode\* newNode = malloc(sizeof(SinglyLinkedListNode));

newNode->data = data;

newNode->next = NULL;

// If inserting at the head (position 0)

if (position == 0) {

newNode->next = llist;

return newNode;

}

// Traverse to the node just before the insertion point

SinglyLinkedListNode\* current = llist;

for (int i = 0; i < position - 1 && current != NULL; i++) {

current = current->next;

}

// Insert the new node

if (current != NULL) {

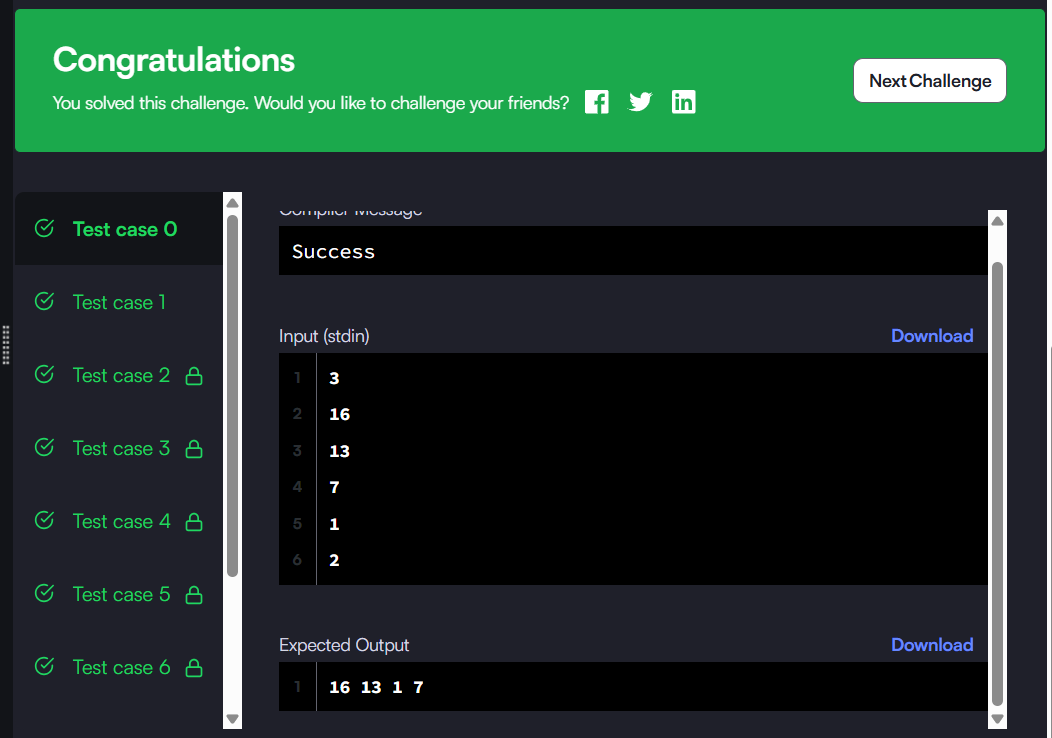
newNode->next = current->next;

current->next = newNode;

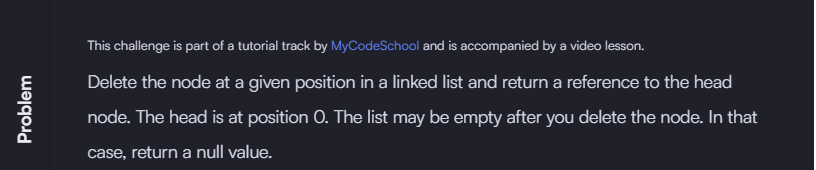
}

return llist;

}



Problem:



SinglyLinkedListNode\* deleteNode(SinglyLinkedListNode\* llist, int position) {

if (!llist) return NULL; // empty list, nothing to delete

// If the node to delete is the head

if (position == 0) {

SinglyLinkedListNode\* temp = llist;

llist = llist->next; // move head to next node

free(temp); // free the old head

return llist;

}

SinglyLinkedListNode\* current = llist;

int index = 0;

// Traverse to the node just before the one to delete

while (current != NULL && index < position - 1) {

current = current->next;

index++;

}

// If position is valid and next node exists

if (current != NULL && current->next != NULL) {

SinglyLinkedListNode\* temp = current->next;

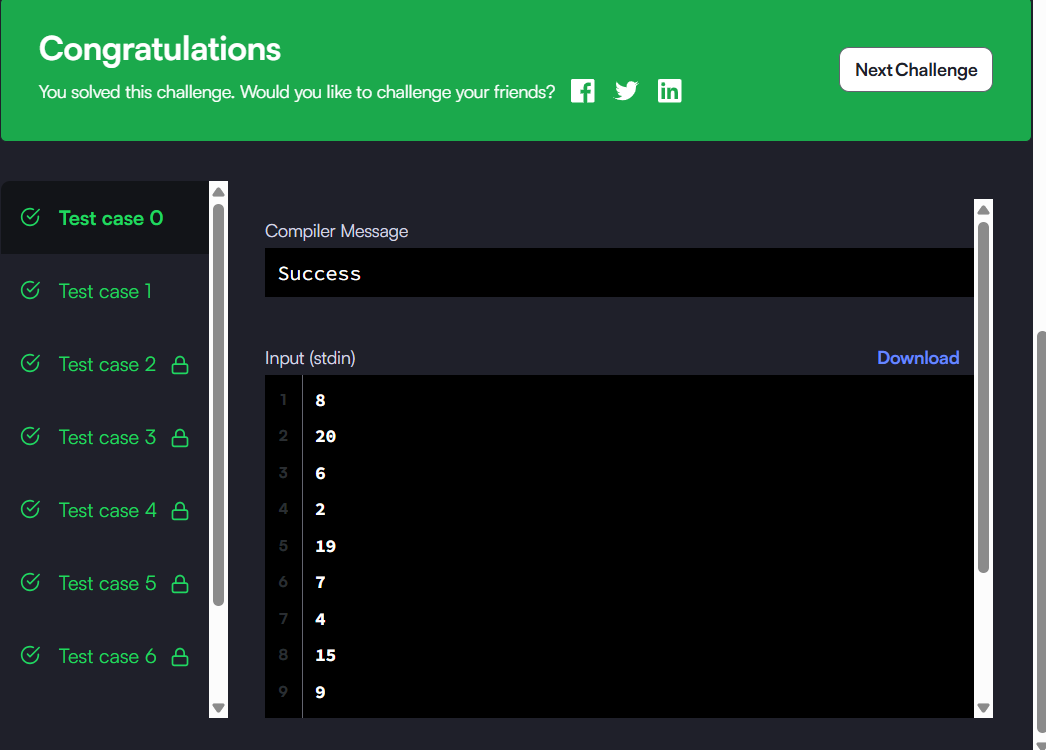
current->next = temp->next; // bypass the node to delete

free(temp); // free memory

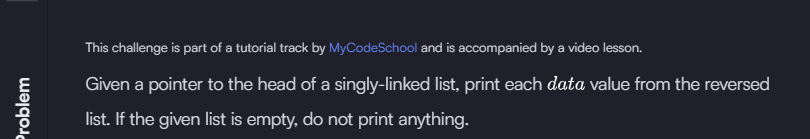
}

return llist;

}



Problem:



Function:

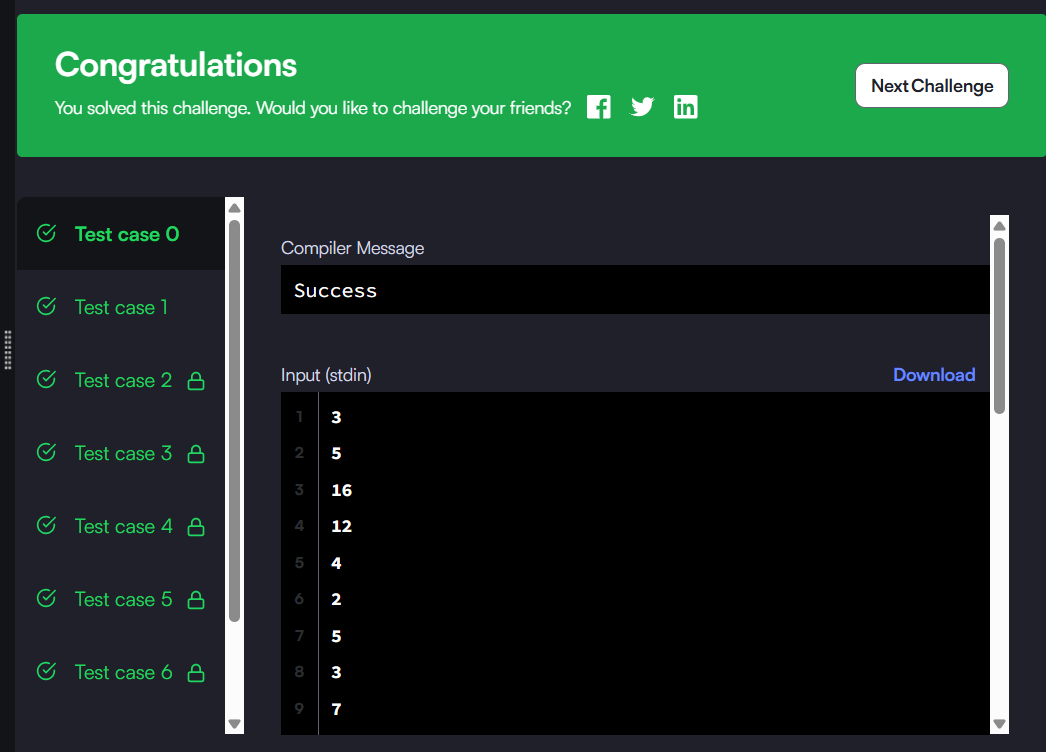
void reversePrint(SinglyLinkedListNode\* llist) {

if (llist == NULL) return;

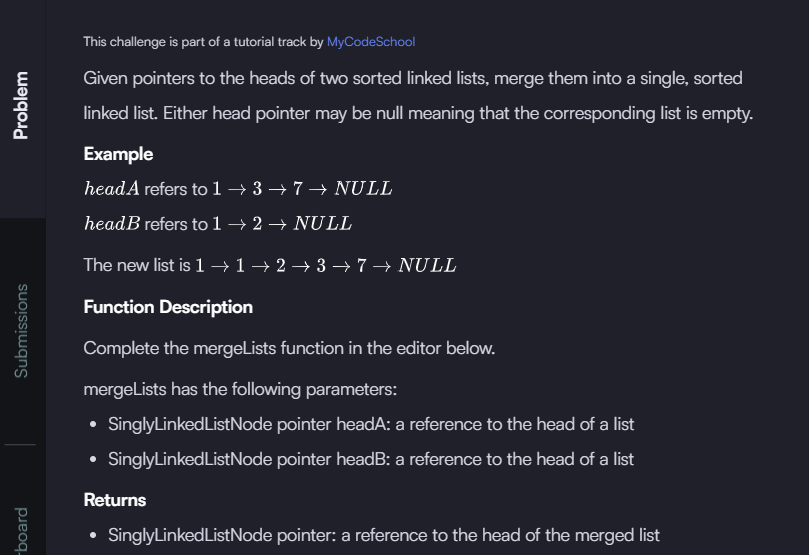
reversePrint(llist->next);

printf("%d\n", llist->data);

}



Problem:



Function:

SinglyLinkedListNode\* mergeLists(SinglyLinkedListNode\* head1, SinglyLinkedListNode\* head2) {

// Base cases

if (head1 == NULL) return head2;

if (head2 == NULL) return head1;

// Recursive merge

if (head1->data <= head2->data) {

head1->next = mergeLists(head1->next, head2);

return head1;

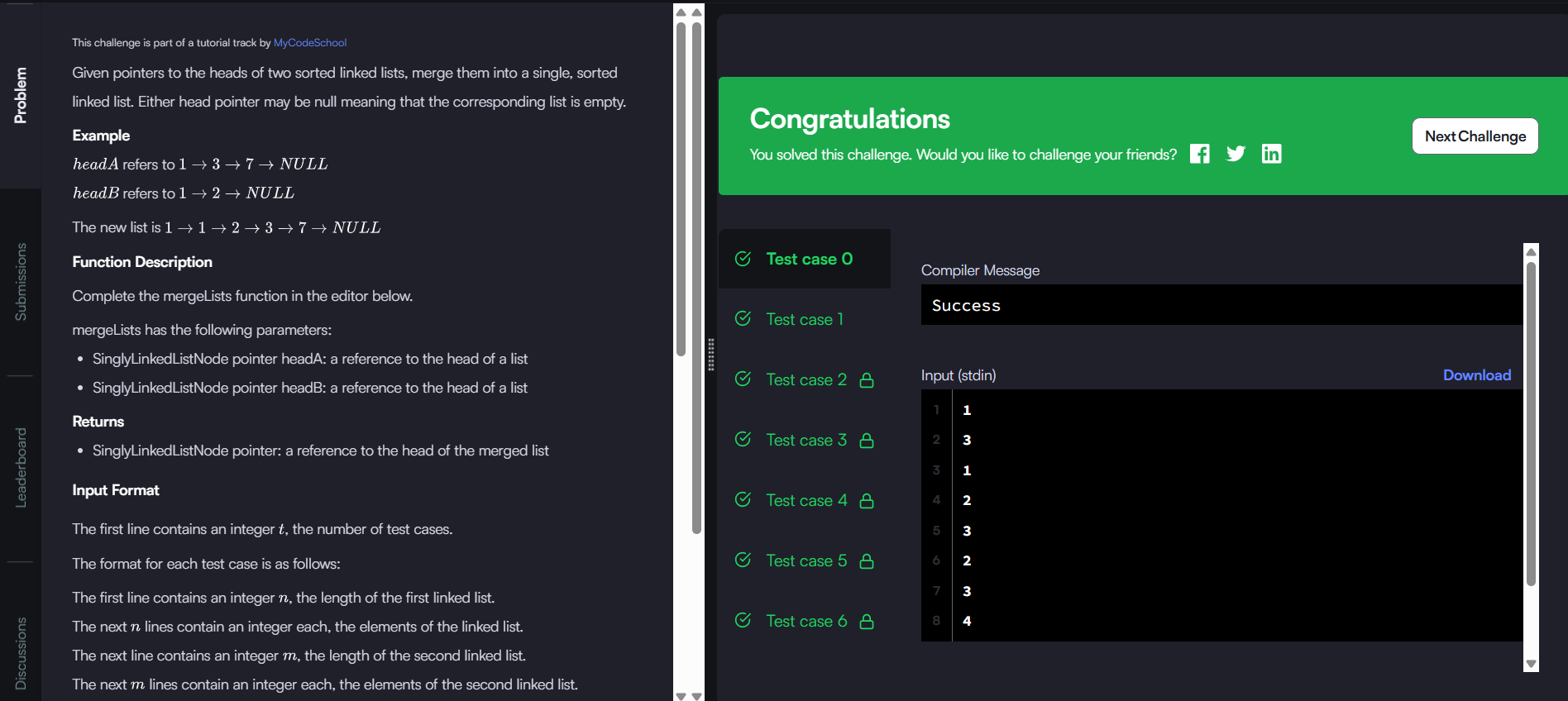
} else {

head2->next = mergeLists(head1, head2->next);

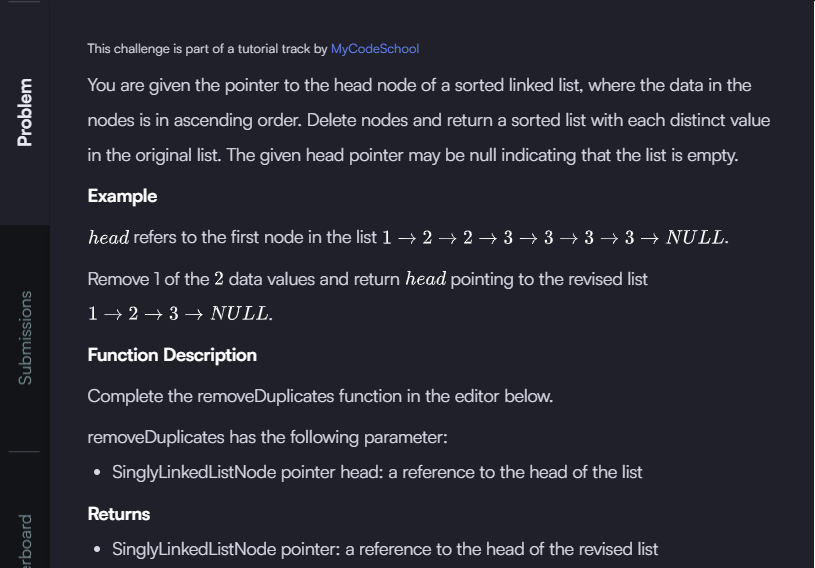
return head2;

}

}



Problem:



Function:

SinglyLinkedListNode\* removeDuplicates(SinglyLinkedListNode\* llist) {

if (llist == NULL) return NULL;

SinglyLinkedListNode\* current = llist;

while (current->next != NULL) {

if (current->data == current->next->data) {

// Duplicate found; remove next node

SinglyLinkedListNode\* temp = current->next;

current->next = current->next->next;

free(temp);

} else {

// Move to next node

current = current->next;

}

}

return llist;

}

