class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (!head || !head->next)

return head;

ListNode\* newHead = reverseList(head->next);

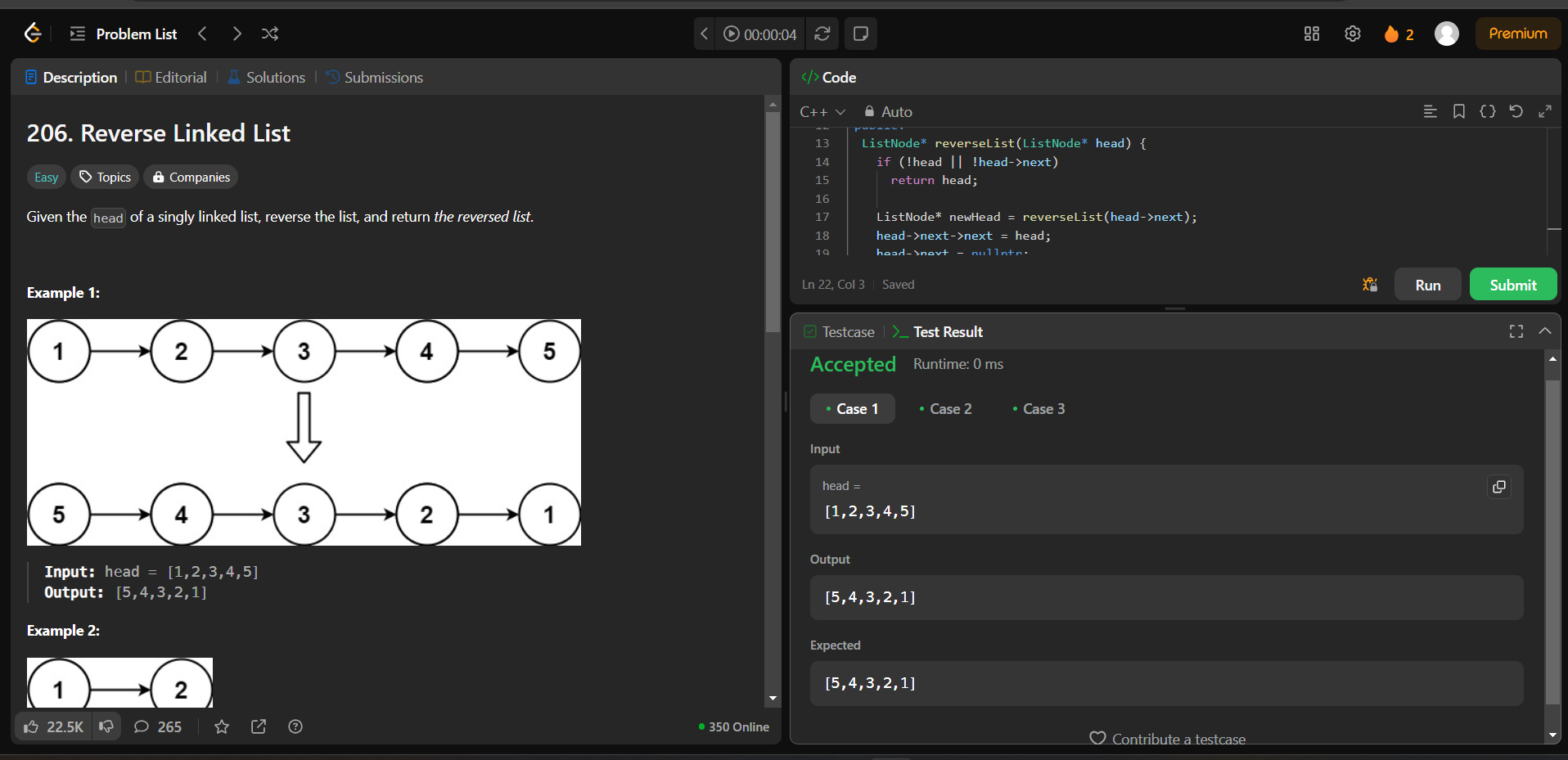
head->next->next = head;

head->next = nullptr;

return newHead;

}

};



Problem 83.

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

ListNode\* curr = head;

while (curr != nullptr) {

while (curr->next && curr->val == curr->next->val)

curr->next = curr->next->next;

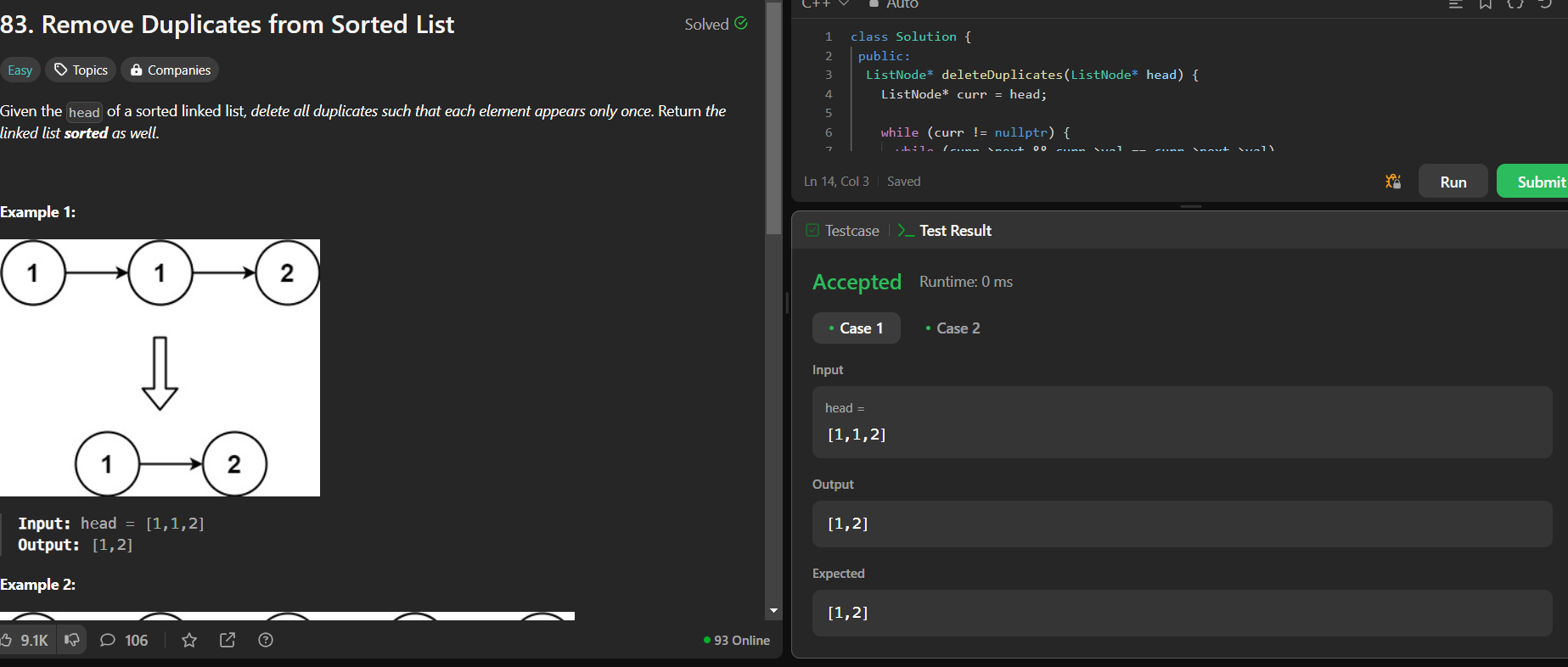
curr = curr->next;

}

return head;

}

};



2095.[Delete middle node of a list](https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list/description/)

class Solution {

 public:

  ListNode\* deleteMiddle(ListNode\* head) {

    ListNode dummy(0, head);

    ListNode\* slow = &dummy;

    ListNode\* fast = &dummy;

    while (fast->next != nullptr && fast->next->next != nullptr) {

      slow = slow->next;

      fast = fast->next->next;

    }

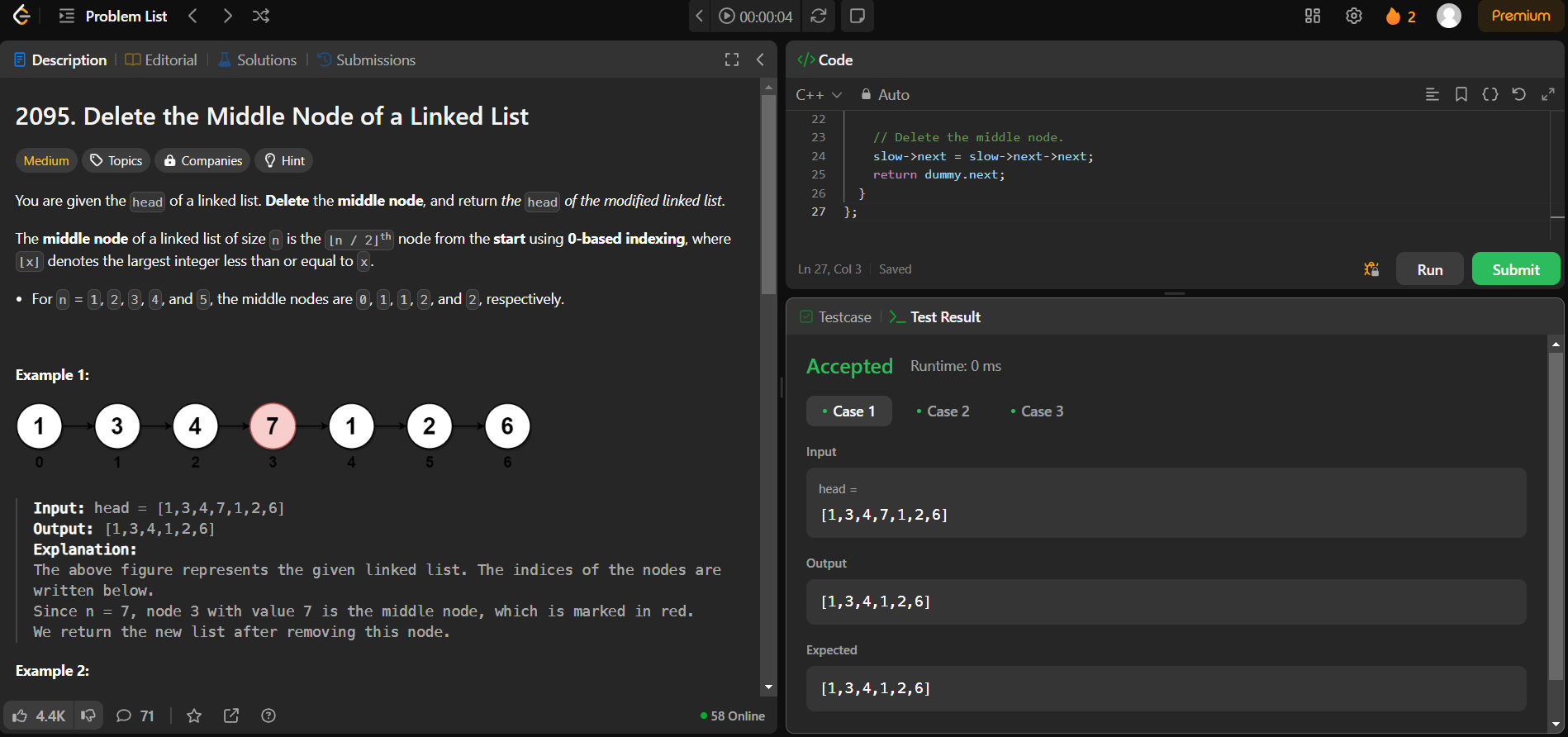
    // Delete the middle node.

    slow->next = slow->next->next;

    return dummy.next;

  }

};



21.[Merge two sorted linked lists](https://leetcode.com/problems/merge-two-sorted-lists/description/)

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

if (!list1 || !list2)

return list1 ? list1 : list2;

if (list1->val > list2->val)

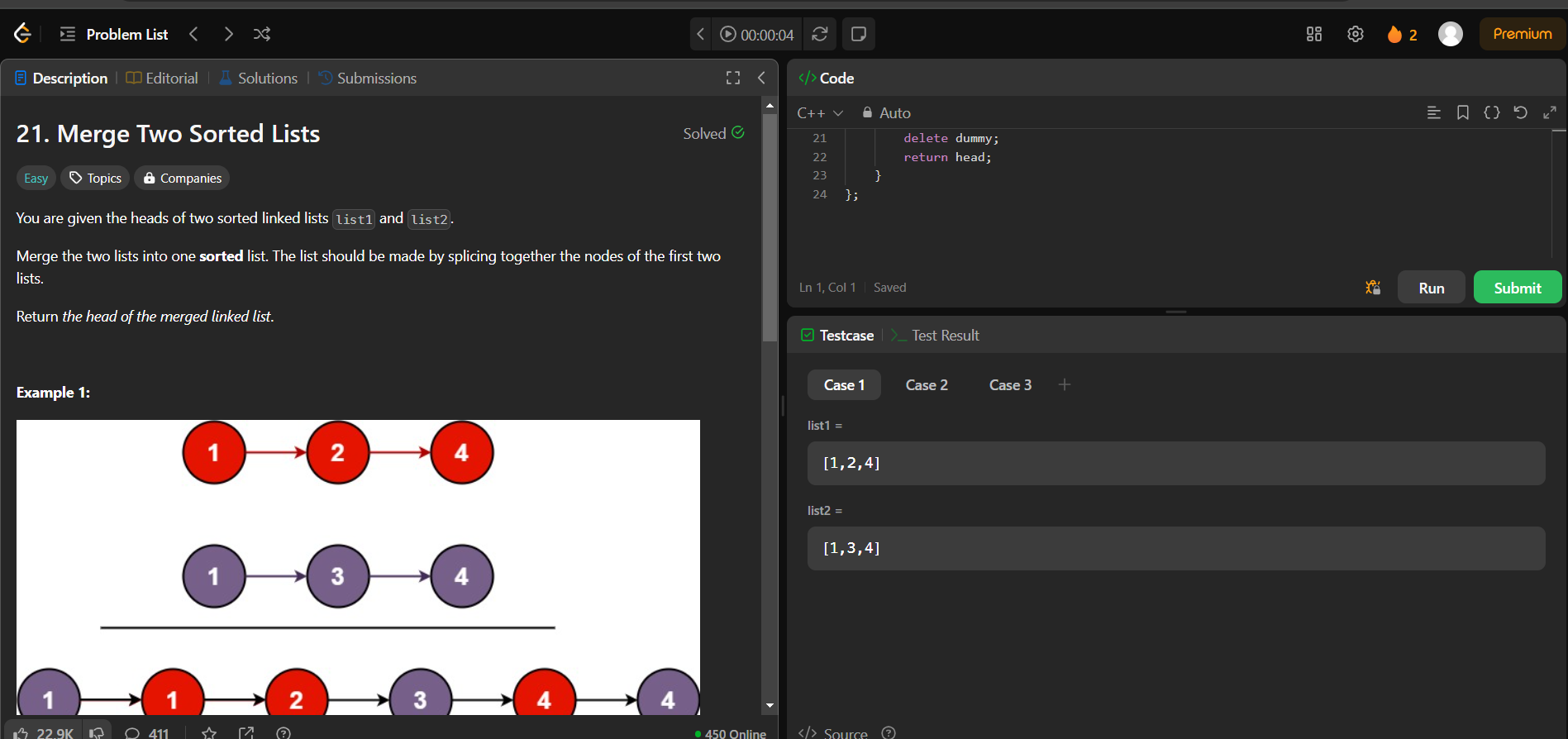
swap(list1, list2);

list1->next = mergeTwoLists(list1->next, list2);

return list1;

}

};



class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

ListNode dummy(0, head);

ListNode\* prev = &dummy;

while (head != nullptr) {

while (head->next && head->val == head->next->val)

head = head->next;

if (prev->next == head)

prev = prev->next;

else

prev->next = head->next;

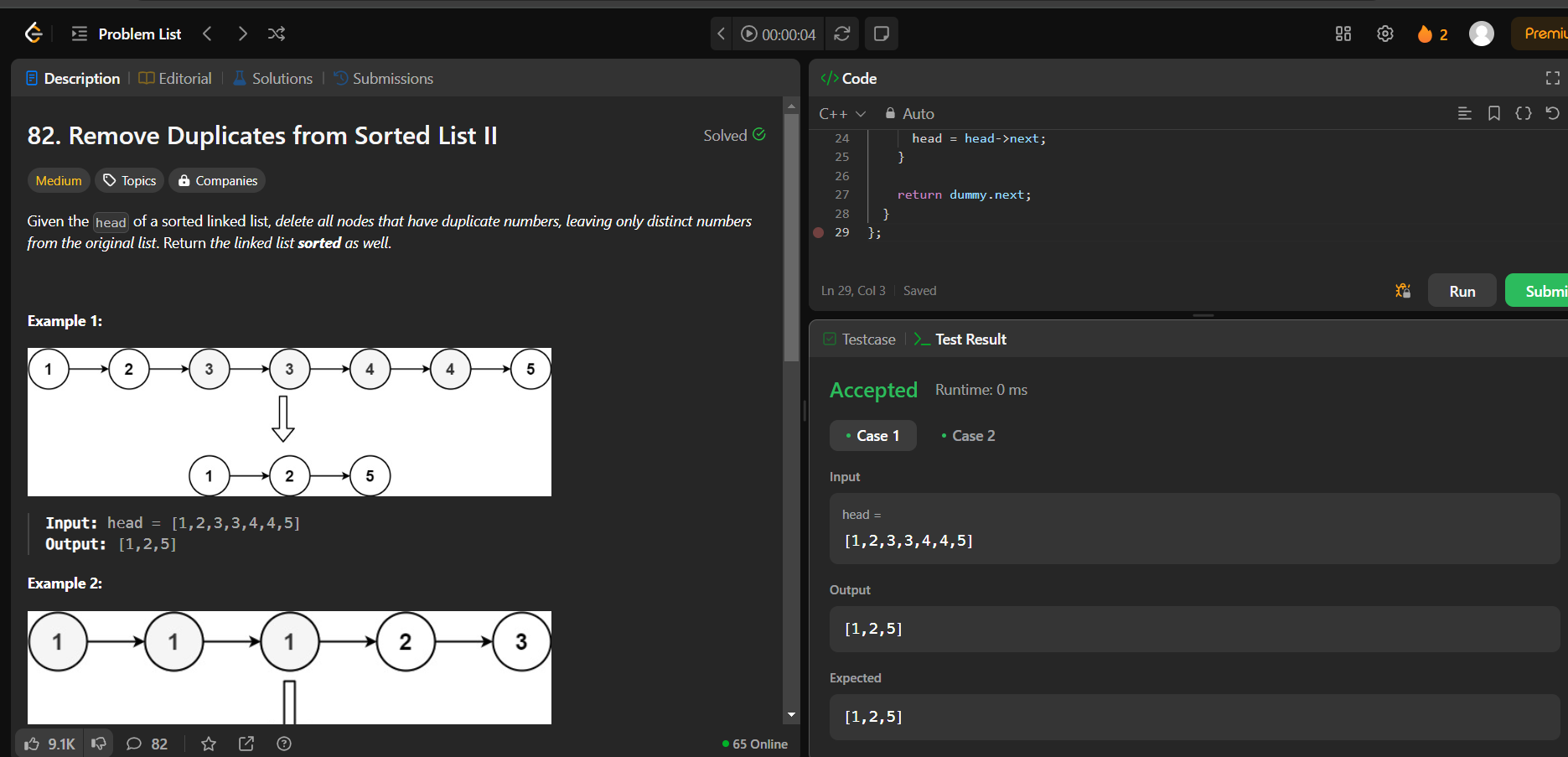
head = head->next;

}

return dummy.next;

}

};



class Solution {

public:

bool hasCycle(ListNode\* head) {

ListNode\* slow = head;

ListNode\* fast = head;

while (fast != nullptr && fast->next != nullptr) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast)

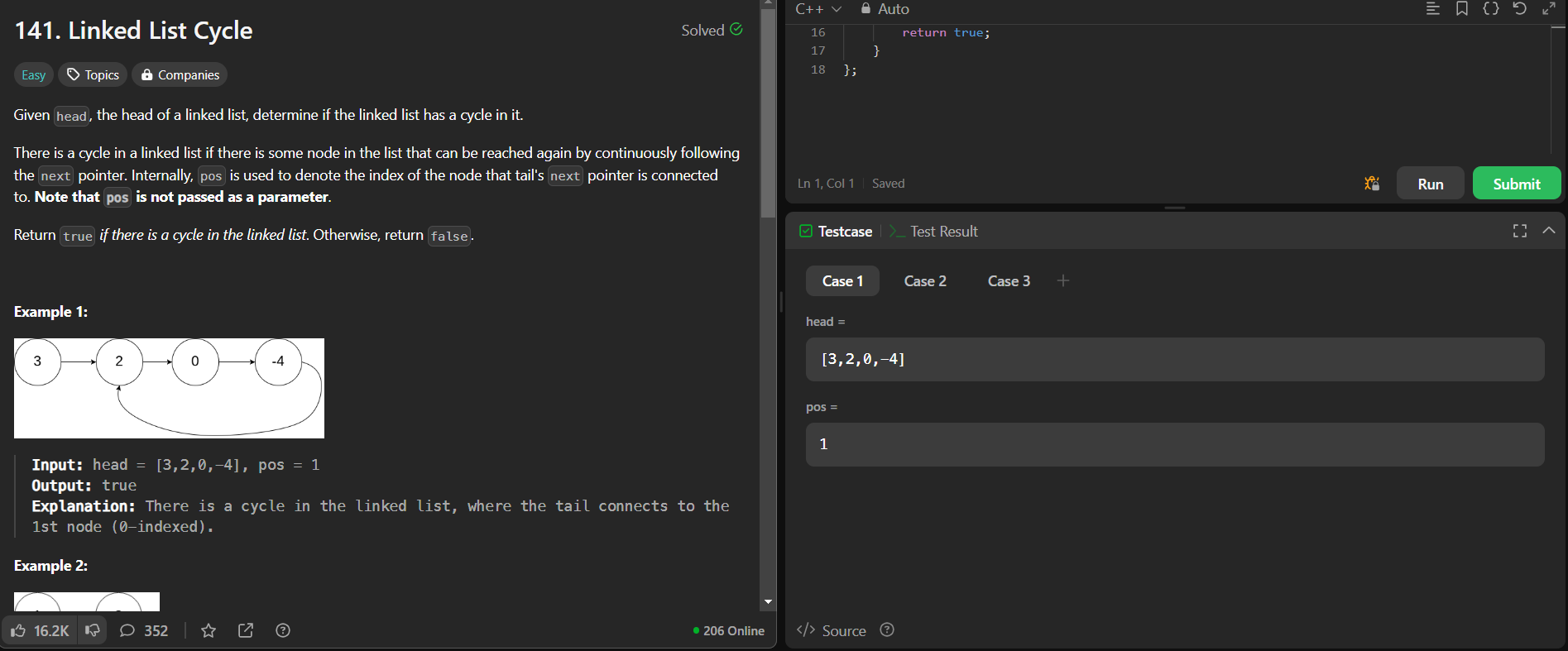
return true;

}

return false;

}

};



class Solution {

 public:

  ListNode\* reverseBetween(ListNode\* head, int left, int right) {

    if (left == 1)

      return reverseN(head, right);

    head->next = reverseBetween(head->next, left - 1, right - 1);

    return head;

  }

 private:

  ListNode\* reverseN(ListNode\* head, int n) {

    if (n == 1)

      return head;

    ListNode\* newHead = reverseN(head->next, n - 1);

    ListNode\* headNext = head->next;

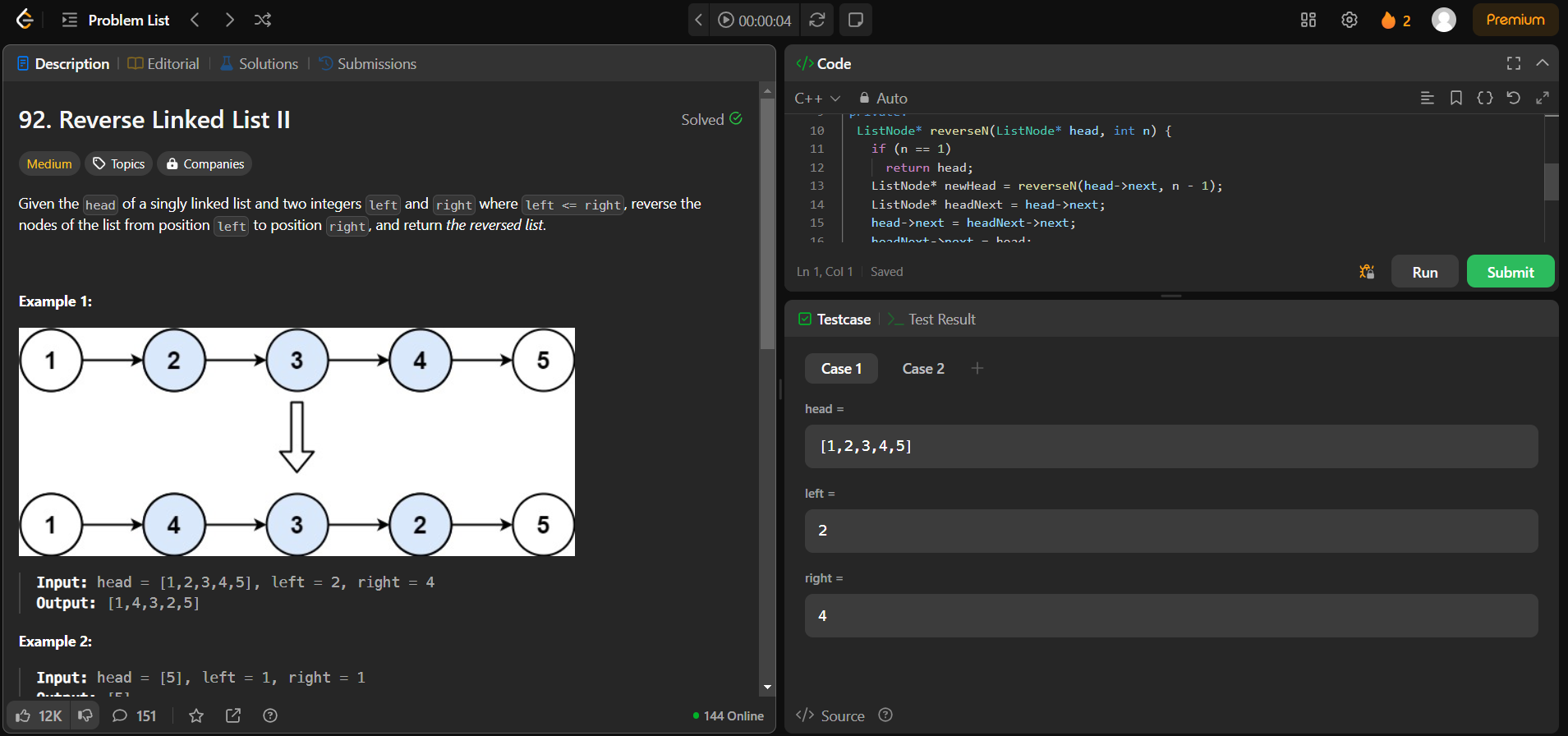
    head->next = headNext->next;

    headNext->next = head;

    return newHead;

  }

};



class Solution {

public:

    ListNode\* rotateRight(ListNode\* head, int k) {

        if (!head || !head->next || k == 0) return head;

        ListNode\* tail = head;

        int length = 1;

        while (tail->next) {

            tail = tail->next;

            length++;

        }k %= length;

        if (k == 0) return head;

        tail->next = head;

        int stepsToNewHead = length - k;

        ListNode\* newTail = tail;

        while (stepsToNewHead--) {

         newTail = newTail->next;

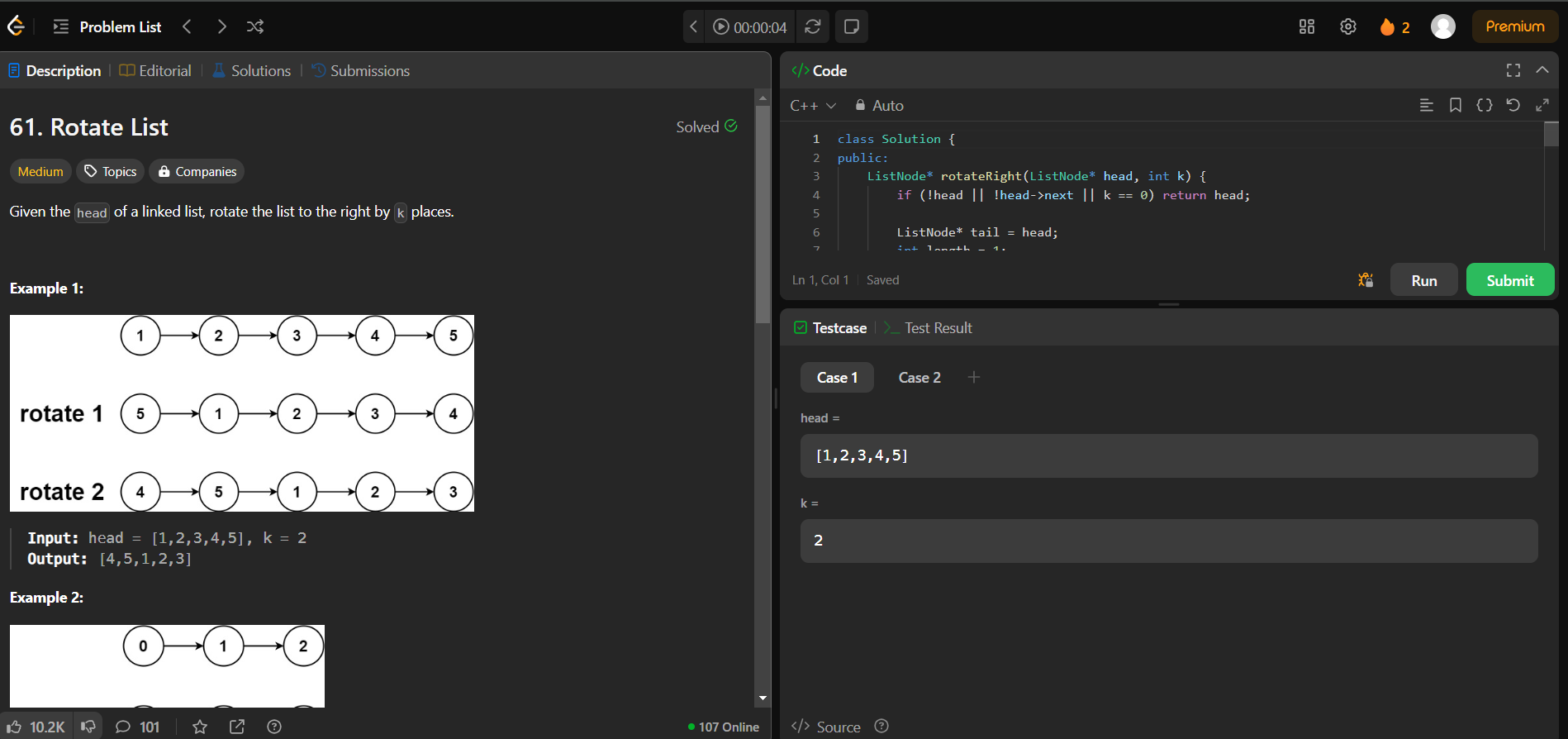
        } ListNode\* newHead = newTail->next;

        newTail->next = nullptr;

        return newHead;

    }

};



class Solution {

public:

    ListNode\* rotateRight(ListNode\* head, int k) {

        if (!head || !head->next || k == 0) return head;

        ListNode\* tail = head;

        int length = 1;

       while (tail->next) {

            tail = tail->next;

            length++;

        }k %= length;

        if (k == 0) return head;

        tail->next = head;

        int stepsToNewHead = length - k;

        ListNode\* newTail = tail;

       while (stepsToNewHead--) {

            newTail = newTail->next;

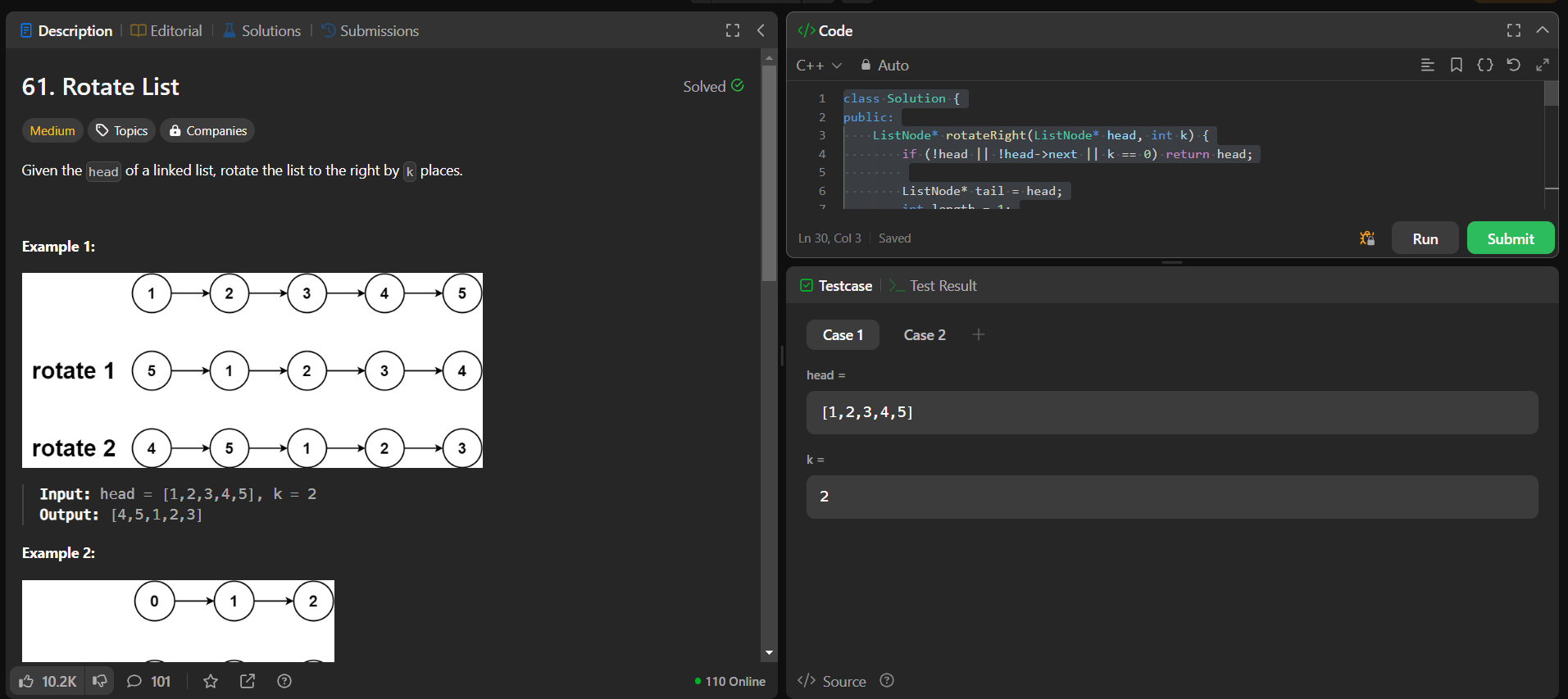
        }ListNode\* newHead = newTail->next;

        newTail->next = nullptr;

        return newHead;

    }

};



lass Solution {

public:

    ListNode\* getmid(ListNode\* head) {

        ListNode\* slow = head;

        ListNode\* fast = head->next;

        while (fast != NULL && fast->next != NULL) {

            slow = slow->next;

            fast = fast->next->next;

        }

        return slow;

    }

    ListNode\* merge(ListNode\* left, ListNode\* right) {

        if (left == NULL)

            return right;

        if (right == NULL)

            return left;

        ListNode\* dummy = new ListNode(0);

        ListNode\* temp = dummy;

        while (left != NULL && right != NULL) {

            if (left->val < right->val) {

                temp->next = left;

                temp = left;

                left = left->next;

            } else {

                temp->next = right;

                temp = right;

                right = right->next;

            }

        }

        while (left != NULL) {

            temp->next = left;

            temp = left;

            left = left->next;

        }

        while (right != NULL) {

            temp->next = right;

            temp = right;

            right = right->next;

        }

        dummy = dummy->next;

        return dummy;

    }

    ListNode\* sortList(ListNode\* head) {

        // using merge sort

        // base case

        if (head == NULL || head->next == NULL)

            return head;

        ListNode\* mid = getmid(head);

        ListNode\* left = head;

        ListNode\* right = mid->next;

        mid->next = NULL;

        left = sortList(left);

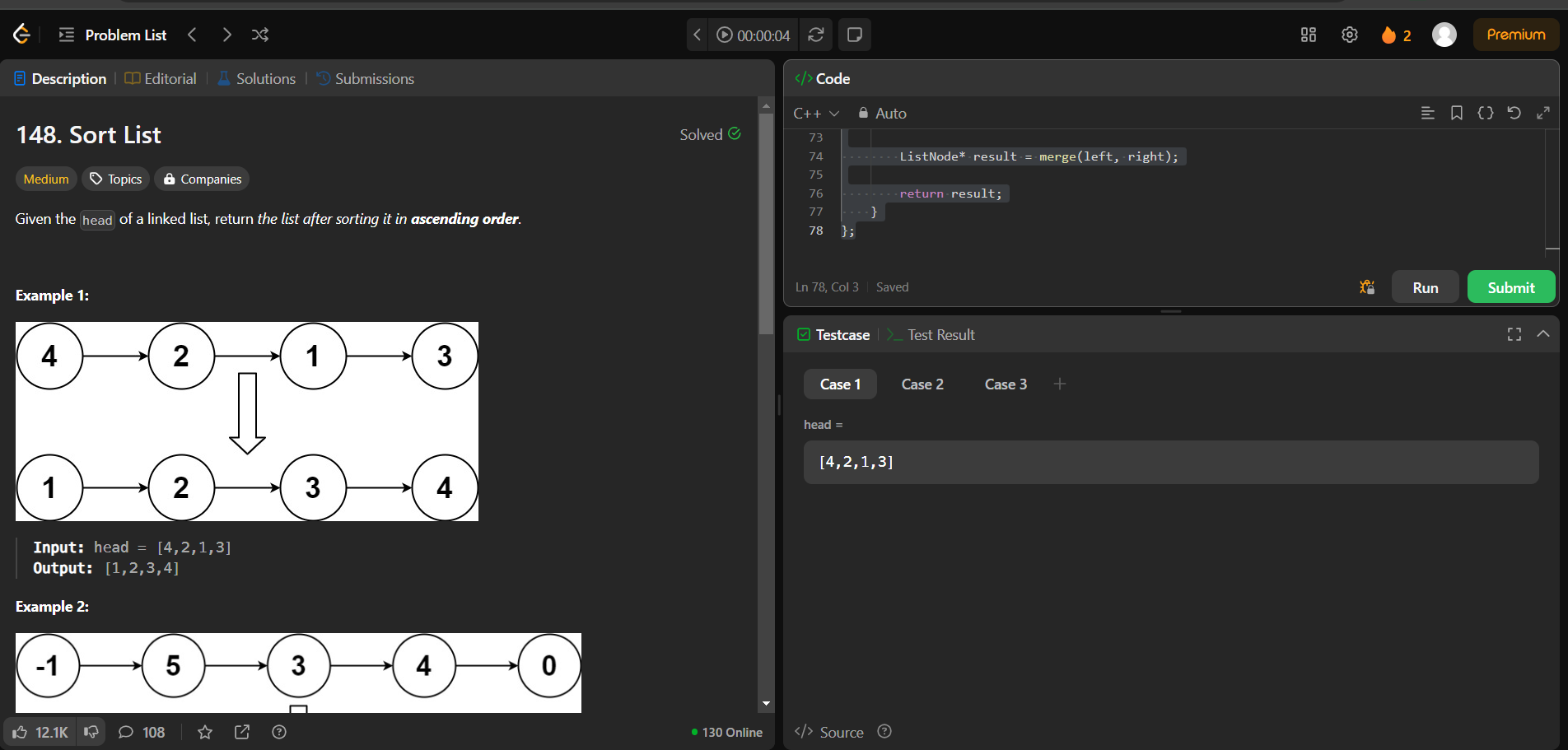
        right = sortList(right);

        ListNode\* result = merge(left, right);

        return result;

    }

};



class Solution {

 public:

  ListNode\* detectCycle(ListNode\* head) {

    // Initialize two pointers, slow and fast, to the head of the linked list.

    ListNode\* slow = head;

    ListNode\* fast = head;

    // Move the slow pointer one step and the fast pointer two steps at a time through the linked list,

    // until they either meet or the fast pointer reaches the end of the list.

    while (fast && fast->next) {

      slow = slow->next;

      fast = fast->next->next;

      if (slow == fast) {

        // If the pointers meet, there is a cycle in the linked list.

        // Reset the slow pointer to the head of the linked list, and move both pointers one step at a time

        // until they meet again. The node where they meet is the starting point of the cycle.

        slow = head;

        while (slow != fast) {

          slow = slow->next;

          fast = fast->next;

        }

        return slow;

      }

    }

    // If the fast pointer reaches the end of the list without meeting the slow pointer,

    // there is no cycle in the linked list. Return nullptr.

    return nullptr;

  }

};

