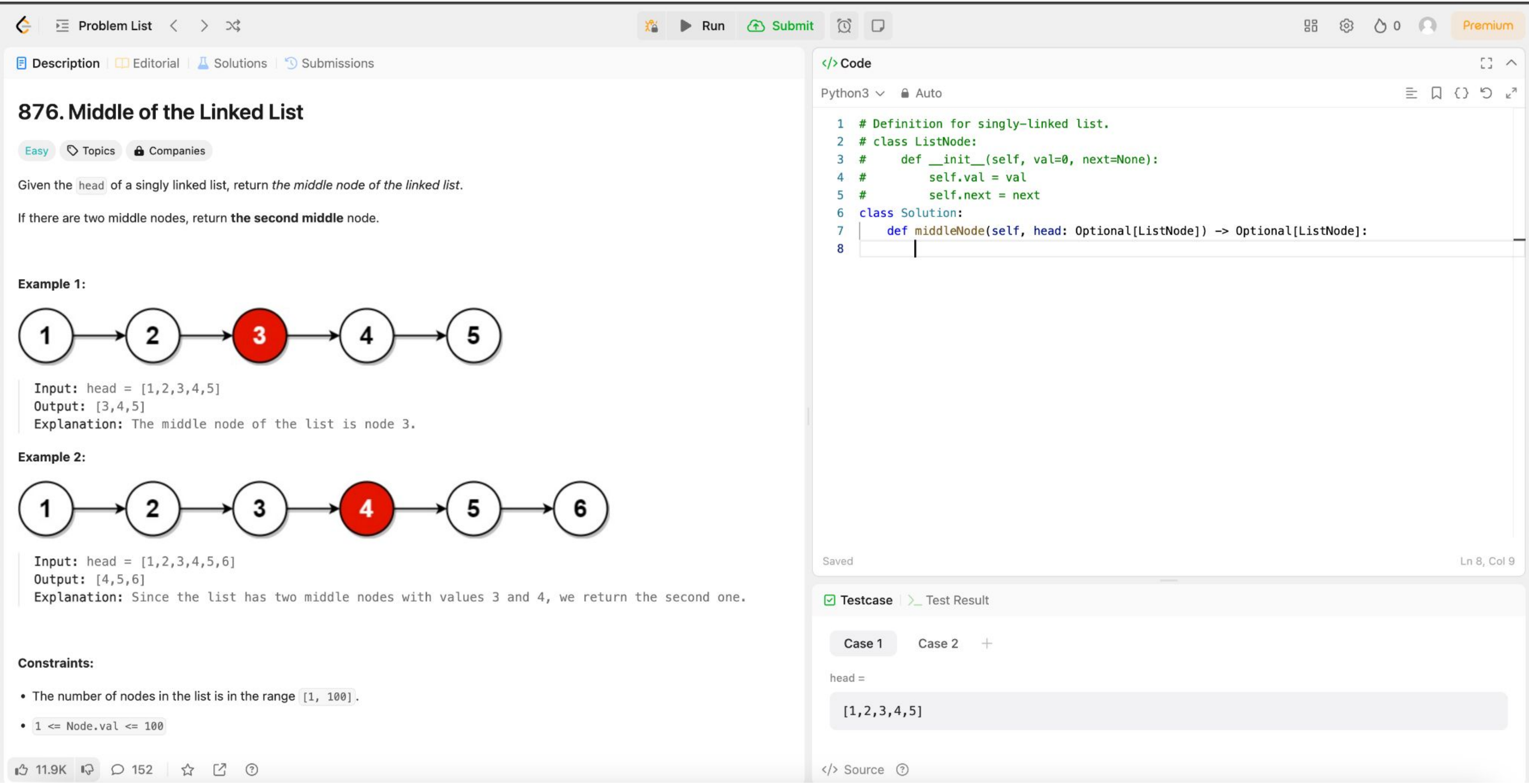


876. Middle of the Linked List

Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Example 1:

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2:

```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.
```

Constraints:

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`

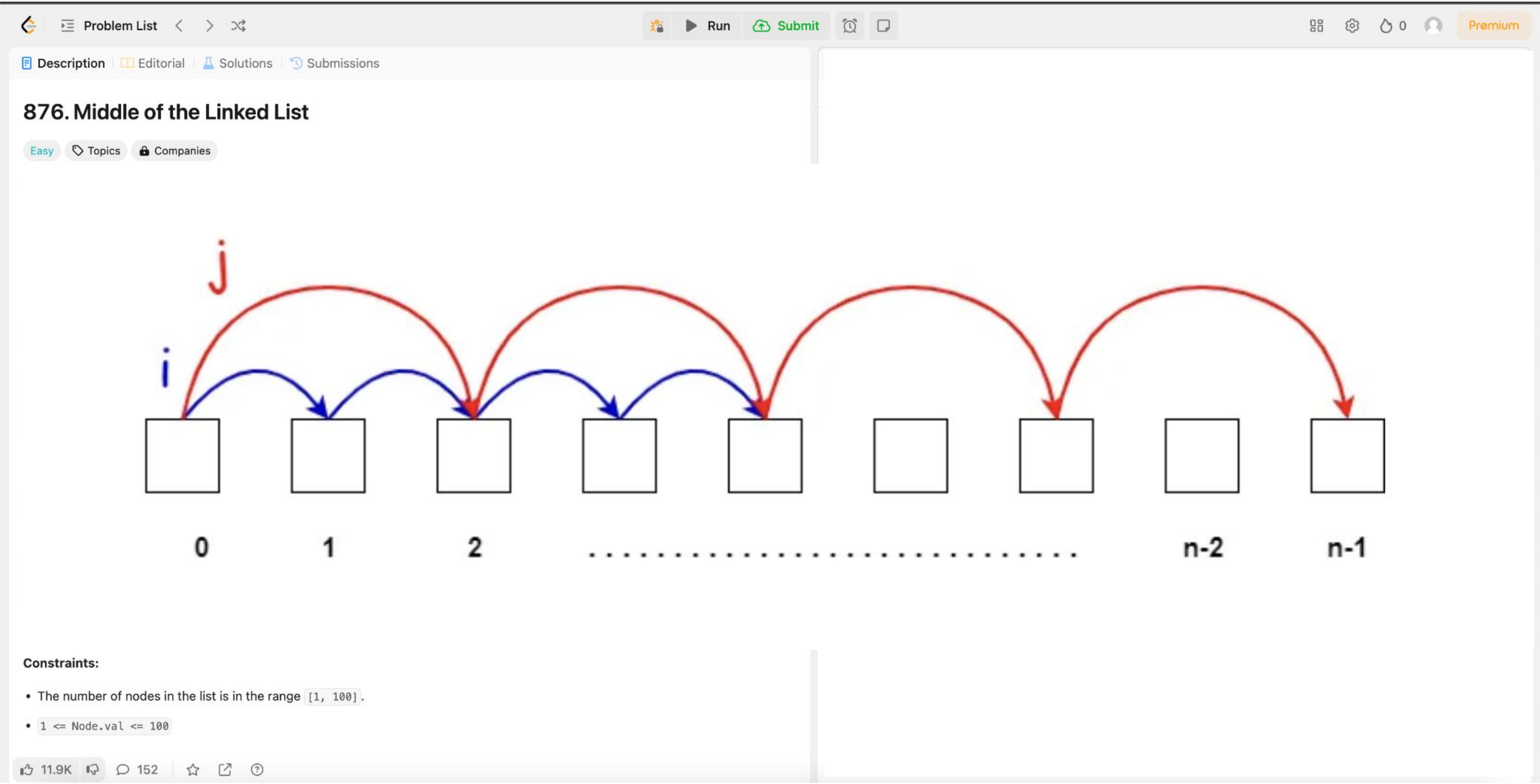
```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        
```

Saved

Ln 8, Col 9

head =

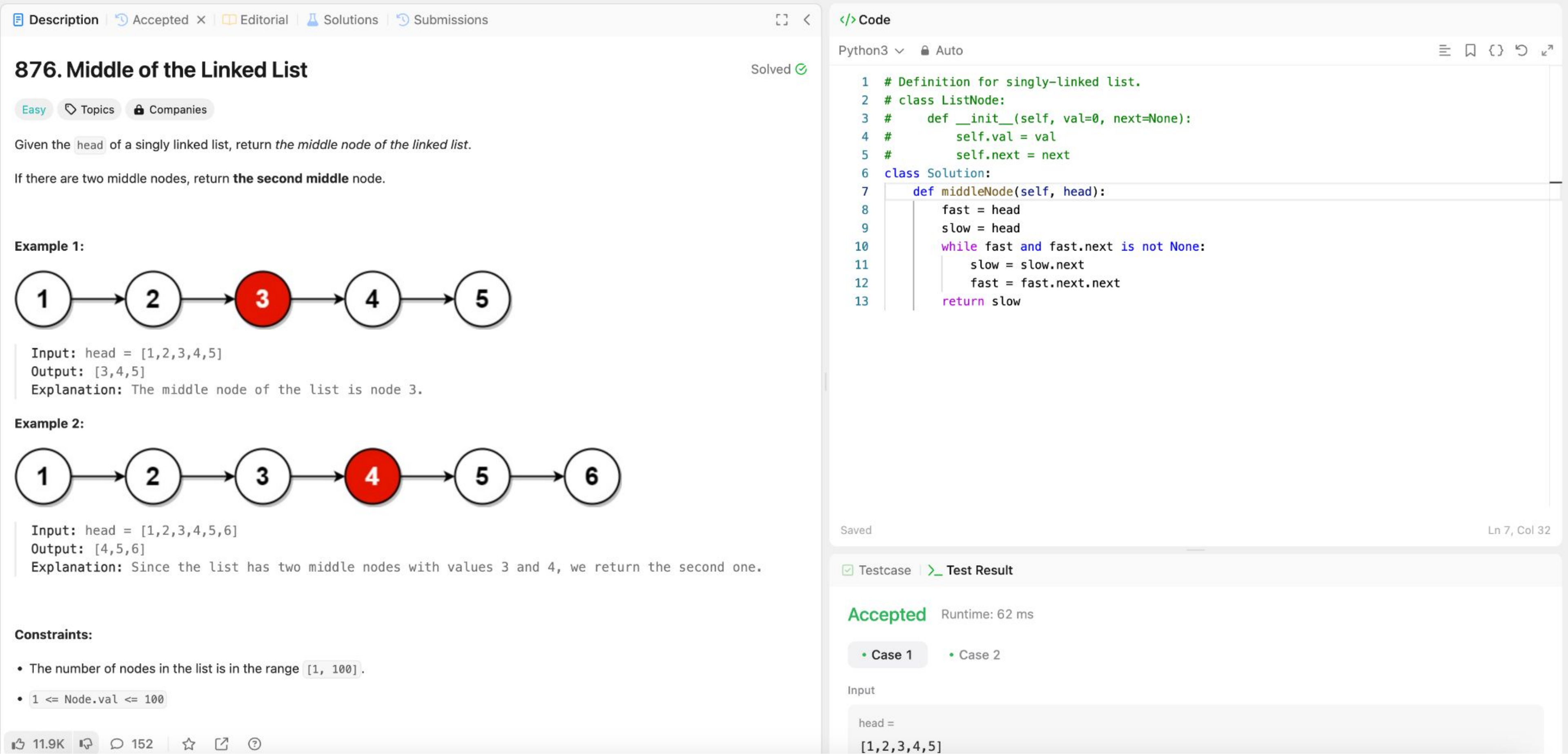
```
[1,2,3,4,5]
```



876. Middle of the Linked List

Constraints:

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`



876. Middle of the Linked List

Solved

Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Example 1:

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2:

```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.
```

Constraints:

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def middleNode(self, head):
        fast = head
        slow = head
        while fast and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

Saved

Ln 7, Col 32

Testcase | Test Result

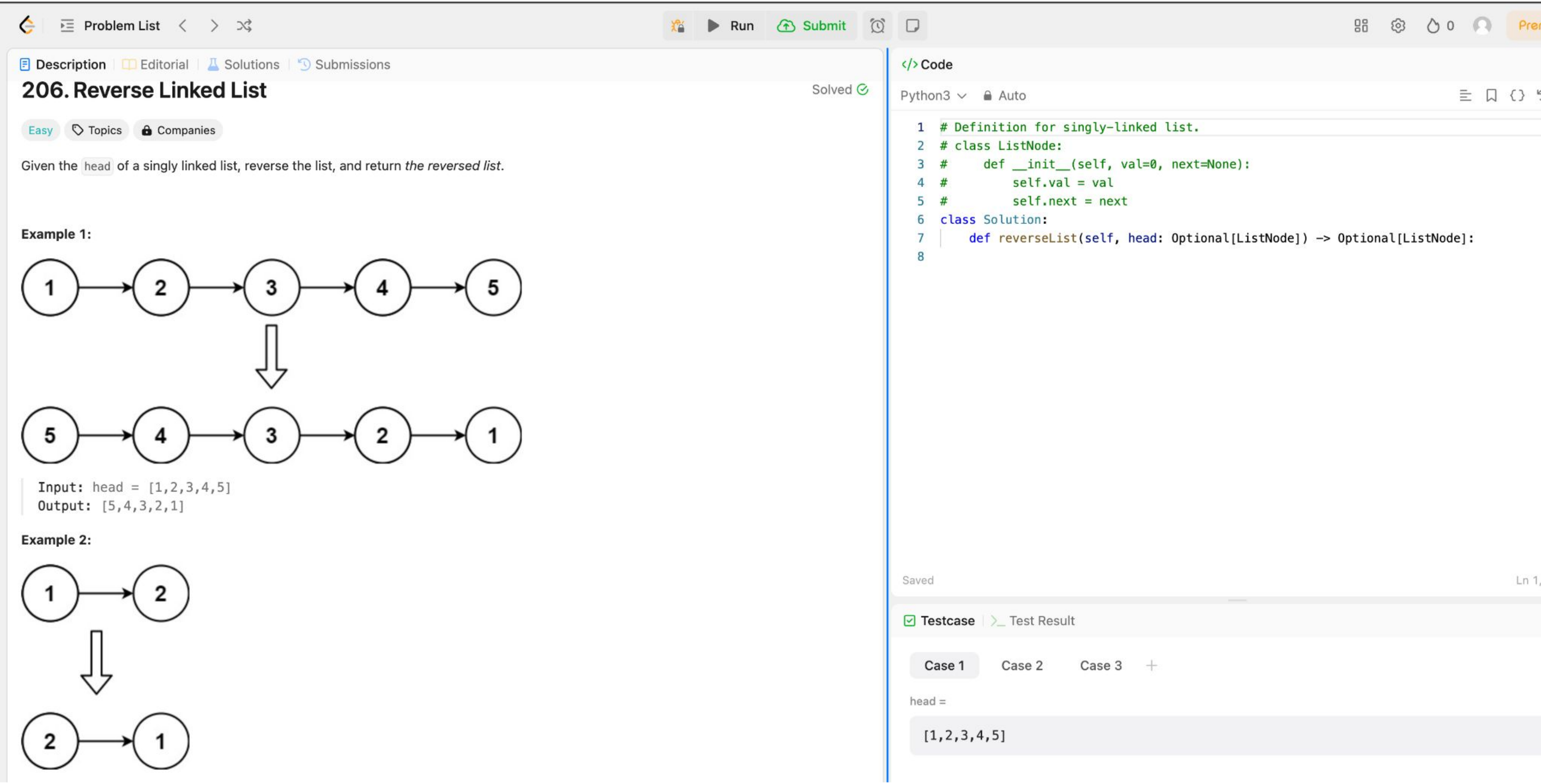
Accepted Runtime: 62 ms

- Case 1
- Case 2

Input

head =

[1,2,3,4,5]



206. Reverse Linked List

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:

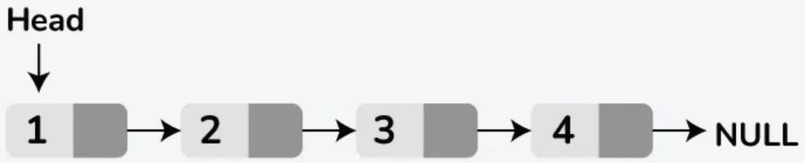
```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:

```

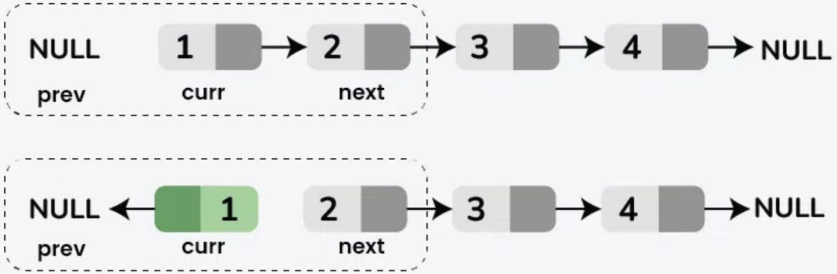
head =

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



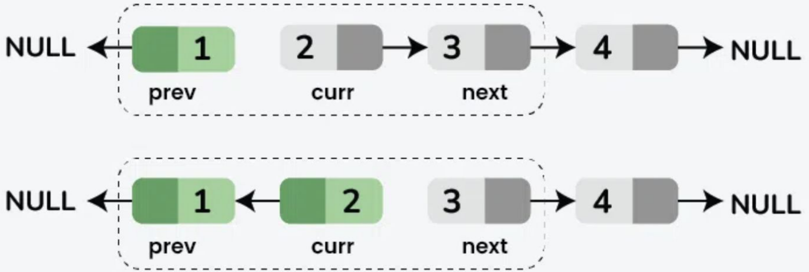
02 Step

Initialize prev pointer = NULL
Store next = curr.next
Update curr.next = prev



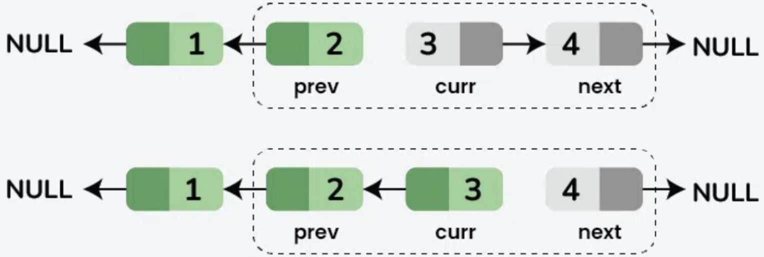
03 Step

Update prev = curr and curr = next
Store next node 3 as next
Update next pointer of current node 2 to previous node 1.



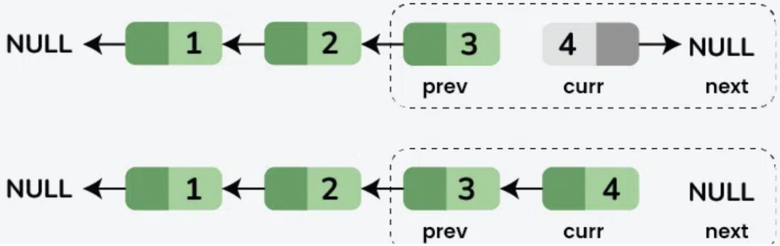
04 Step

Update prev = curr and curr = next
Store next node 4 as next
Update next pointer of current node 3 to previous node 2.



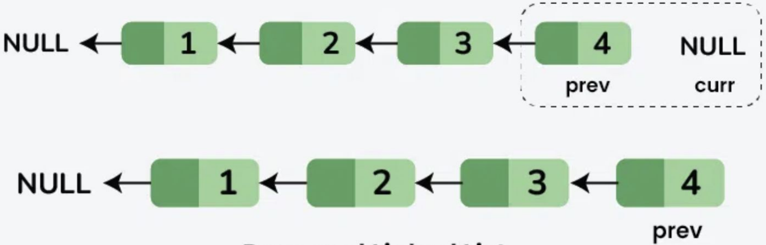
05 Step

Update prev = curr and curr = next
next pointer points to NULL
Update next pointer of current node 4 to previous node 3.

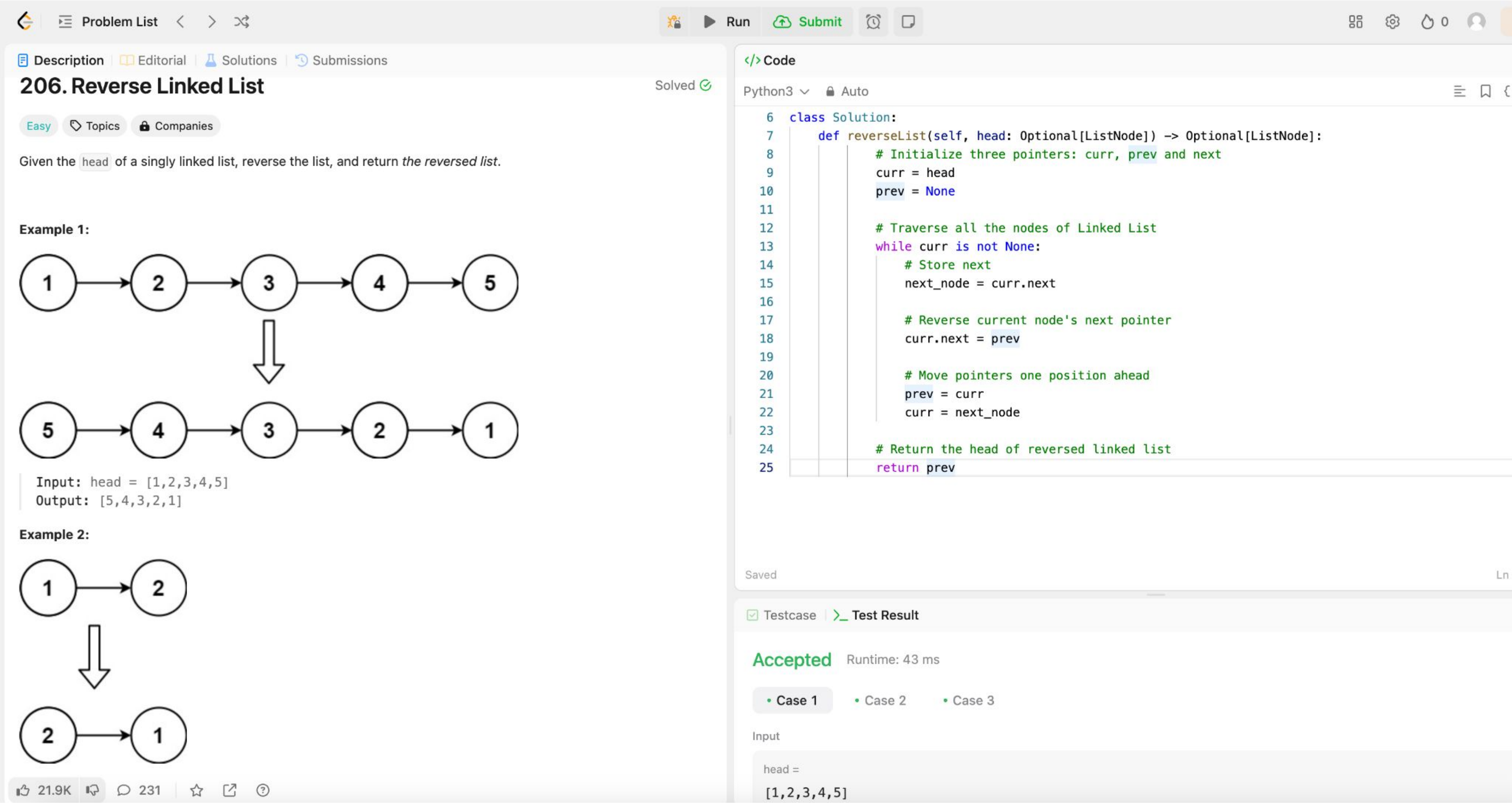


06 Step

Update prev = curr and curr = next
Finally, curr becomes NULL and prev stores the head of the reversed linked list



Reversed Linked List



206. Reverse Linked List

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:

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

Example 2:

```python
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Initialize three pointers: curr, prev and next
        curr = head
        prev = None

        # Traverse all the nodes of Linked List
        while curr is not None:
            # Store next
            next_node = curr.next

            # Reverse current node's next pointer
            curr.next = prev

            # Move pointers one position ahead
            prev = curr
            curr = next_node

        # Return the head of reversed linked list
        return prev
```

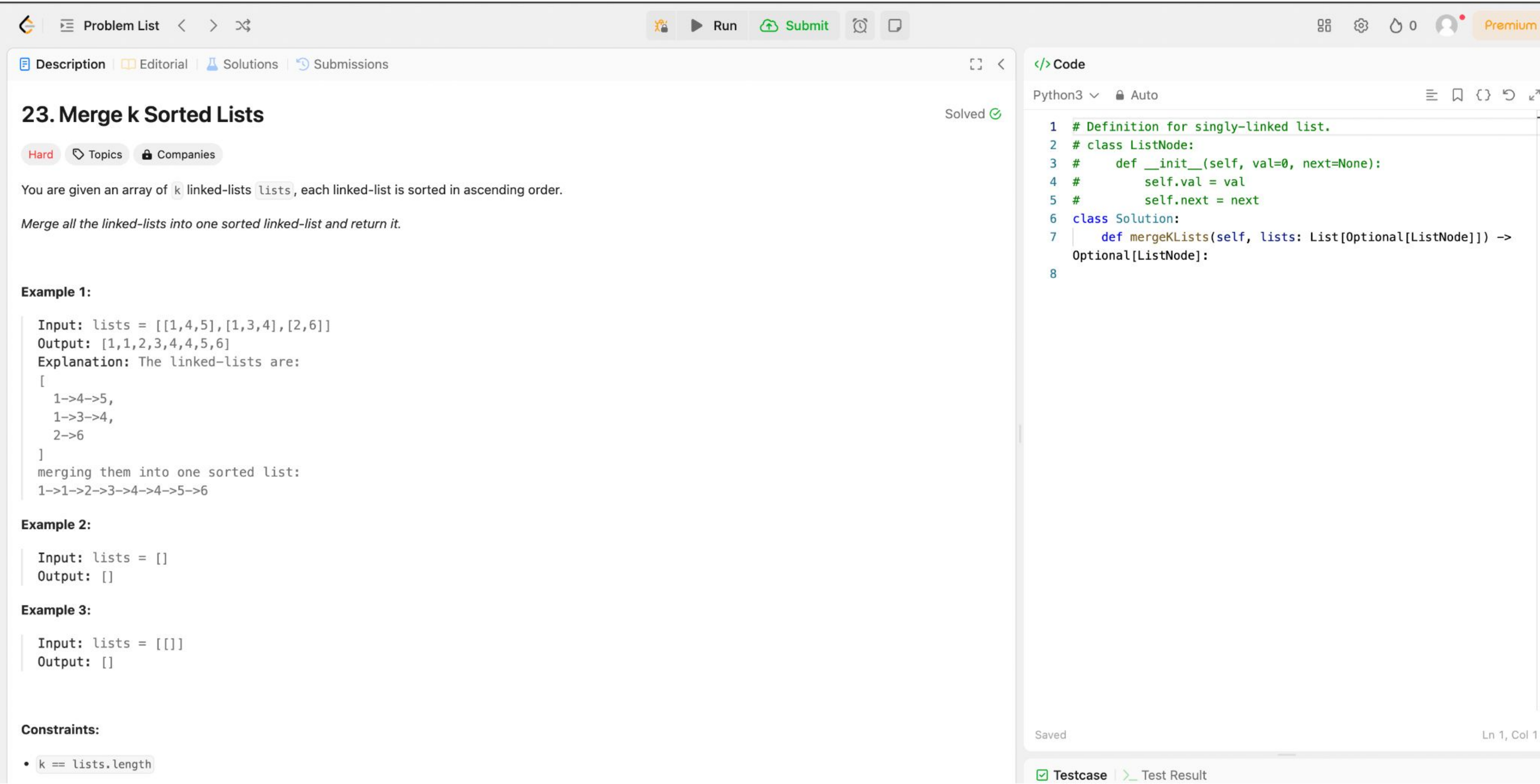
Accepted Runtime: 43 ms

• Case 1 • Case 2 • Case 3

Input

head =

[1,2,3,4,5]



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.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

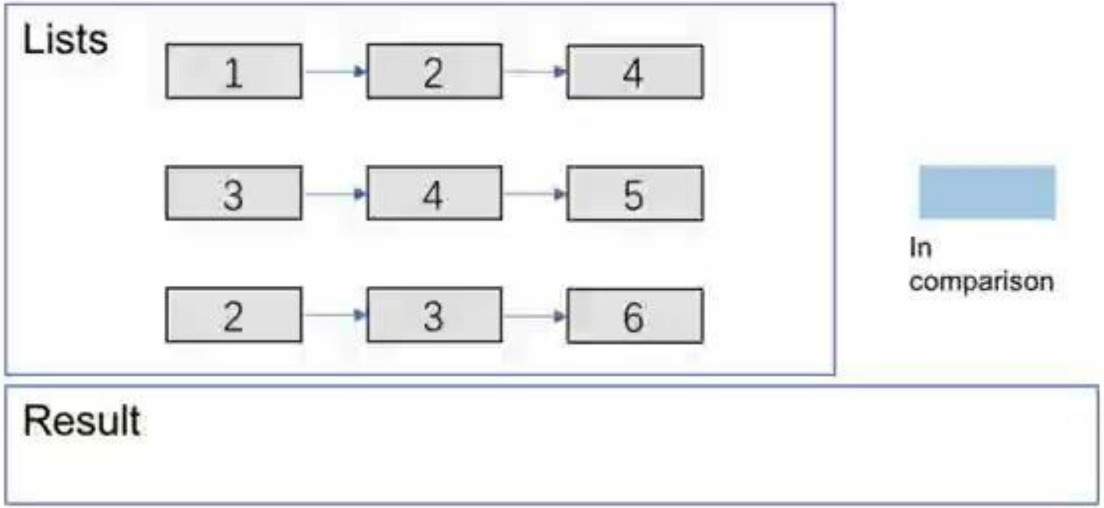
Example 3:

```
Input: lists = [[]]
Output: []
```

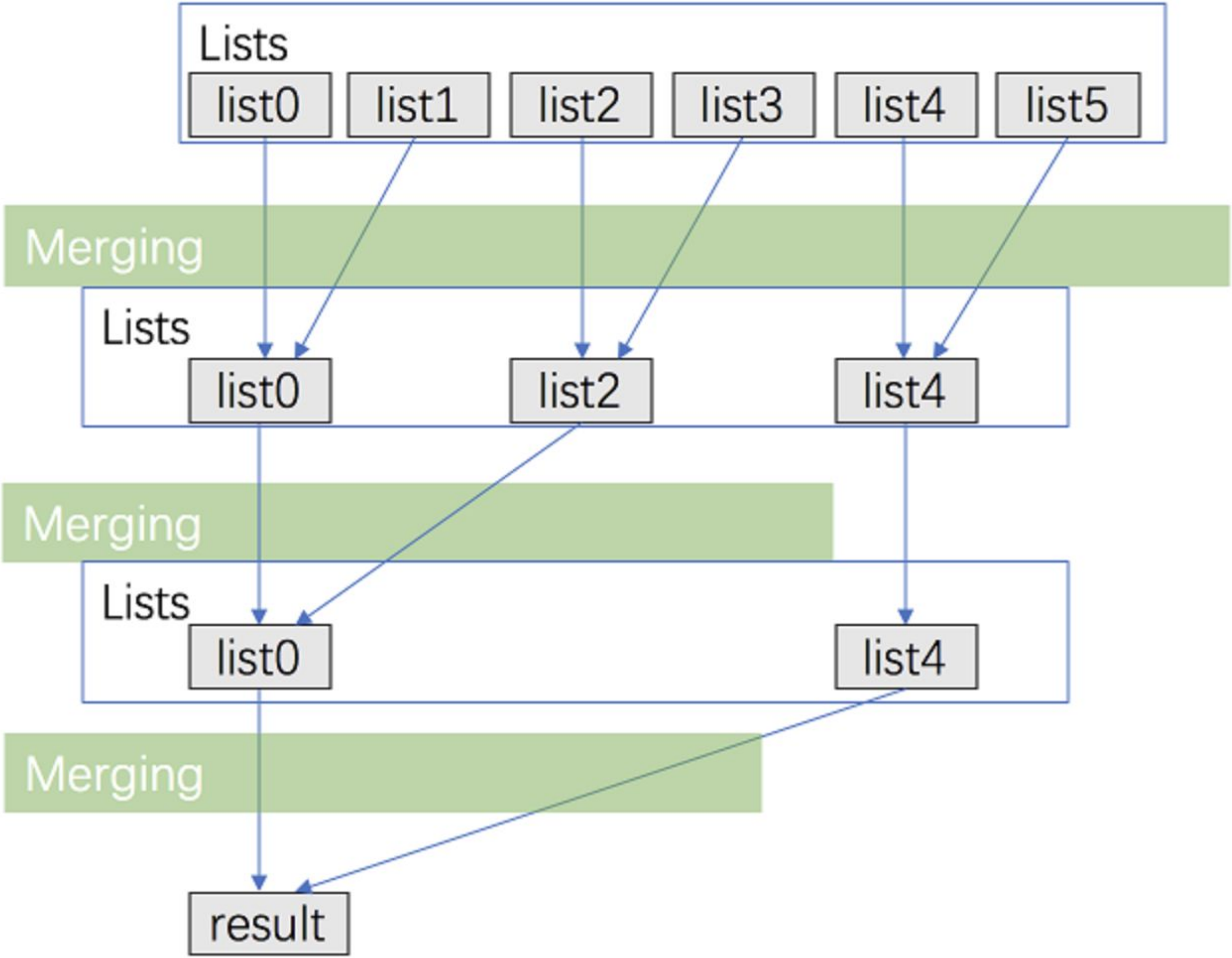
Constraints:

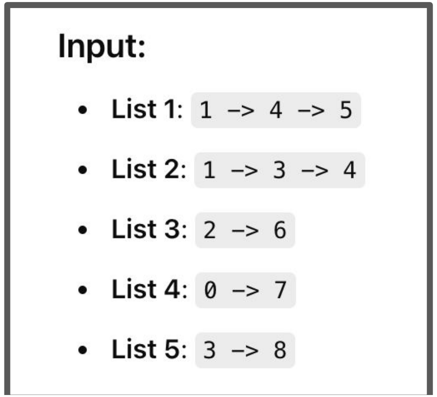
- `k == lists.length`

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:

```



- Time complexity : $O(kN)$ where k is the number of linked lists.
  - Almost every selection of node in final linked costs $O(k)$ (k-1 times comparison).
  - There are N nodes in the final linked list.


Lists
list0
list1
list2
list3
list4
list5
Merging
Lists
list0
list2
list4
Merging
Lists
list0
list4
Merging
result



Input:

- **List 1**: `1 -> 4 -> 5`
- **List 2**: `1 -> 3 -> 4`
- **List 3**: `2 -> 6`
- **List 4**: `0 -> 7`
- **List 5**: `3 -> 8`

1. **First iteration (`interval = 1`)**:
   - Merge `List 1` and `List 2`: `1 -> 1 -> 3 -> 4 -> 4 -> 5`
   - Merge `List 3` and `List 4`: `0 -> 2 -> 6 -> 7`
   - `List 5` remains unchanged.

   After iteration:

   - `lists[0] = 1 -> 1 -> 3 -> 4 -> 4 -> 5`
   - `lists[1] = 1 -> 3 -> 4`
   - `lists[2] = 0 -> 2 -> 6 -> 7`
   - `lists[3] = 3 -> 8`
   - `lists[4] = 3 -> 8`

2. **Second iteration (`interval = 2`)**:
   - Merge `List 0` and `List 2`: `0 -> 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6 -> 7`
   - `List 5` remains unchanged.

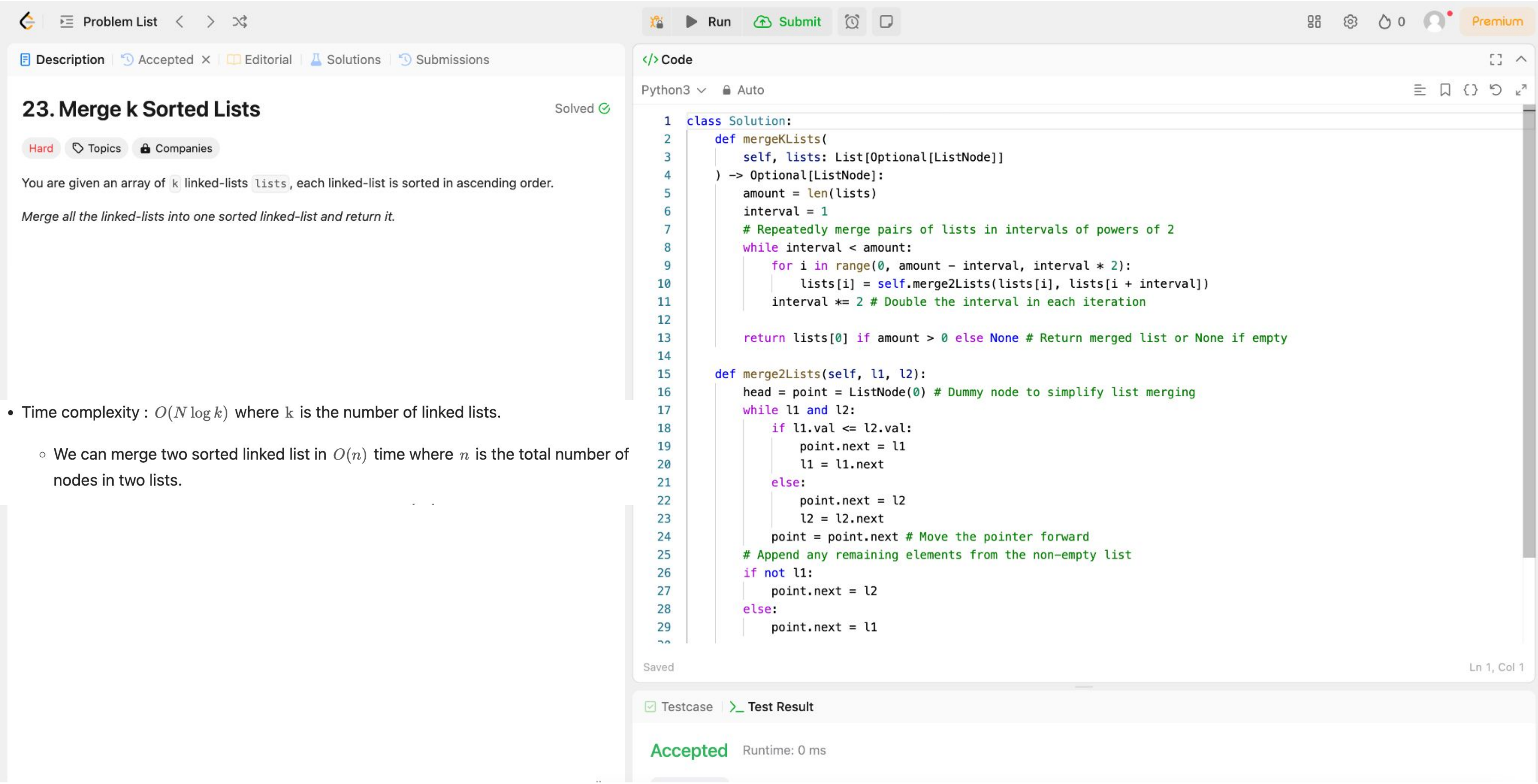
   After iteration:

   - `lists[0] = 0 -> 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6 -> 7`
   - `lists[1] = 1 -> 3 -> 4`
   - `lists[2] = 0 -> 2 -> 6 -> 7`
   - `lists[3] = 3 -> 8`
   - `lists[4] = 3 -> 8`

3. **Third iteration (`interval = 4`)**:
   - Merge `List 0` and `List 4`: `0 -> 1 -> 1 -> 2 -> 3 -> 3 -> 4 -> 4 -> 5 -> 6 -> 7 -> 8`

   Final result:

   - `lists[0] = 0 -> 1 -> 1 -> 2 -> 3 -> 3 -> 4 -> 4 -> 5 -> 6 -> 7 -> 8`



23. Merge k Sorted Lists

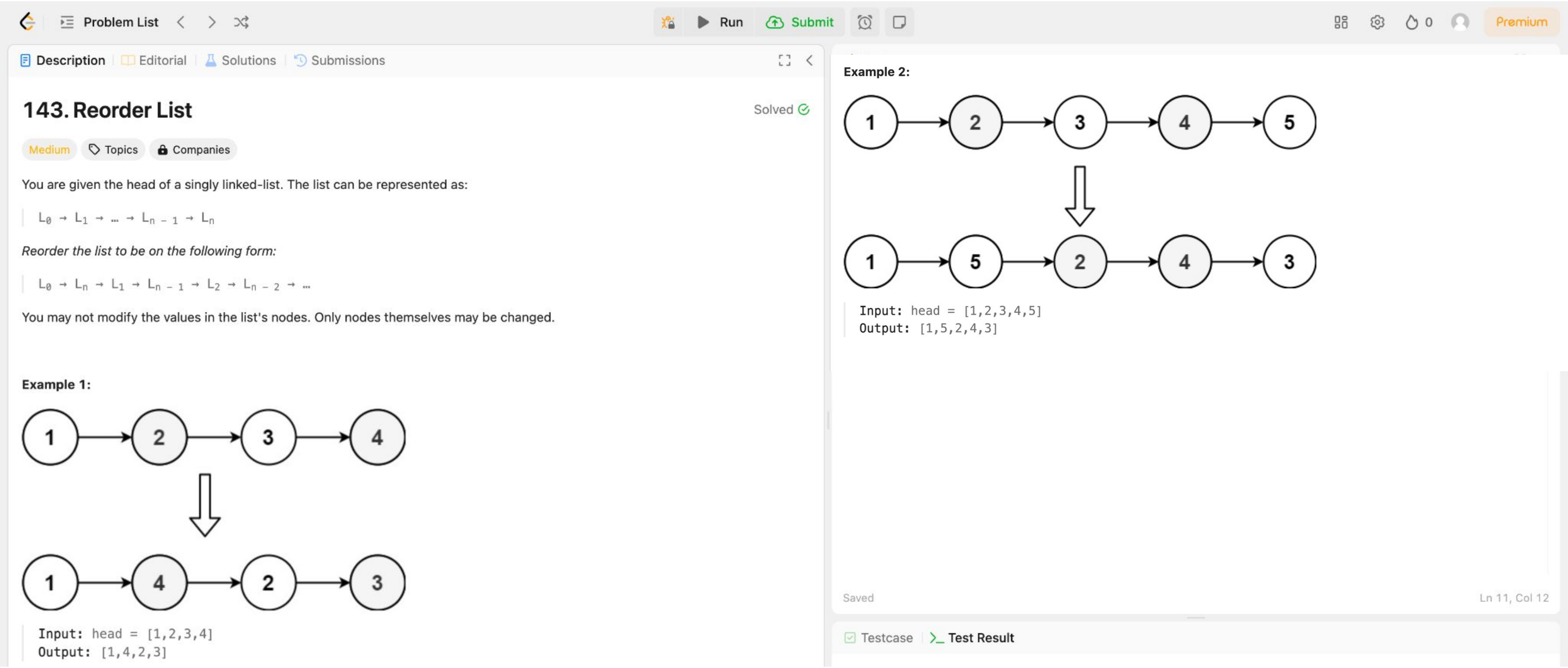
You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

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

- Time complexity : $O(N \log k)$ where k is the number of linked lists.
  - We can merge two sorted linked list in $O(n)$ time where n is the total number of nodes in two lists.

```python
class Solution:
    def mergeKLists(
        self, lists: List[Optional[ListNode]]
    ) -> Optional[ListNode]:
        amount = len(lists)
        interval = 1
        # Repeatedly merge pairs of lists in intervals of powers of 2
        while interval < amount:
            for i in range(0, amount - interval, interval * 2):
                lists[i] = self.merge2Lists(lists[i], lists[i + interval])
            interval *= 2 # Double the interval in each iteration

        return lists[0] if amount > 0 else None # Return merged list or None if empty

    def merge2Lists(self, l1, l2):
        head = point = ListNode(0) # Dummy node to simplify list merging
        while l1 and l2:
            if l1.val <= l2.val:
                point.next = l1
                l1 = l1.next
            else:
                point.next = l2
                l2 = l2.next
            point = point.next # Move the pointer forward
        # Append any remaining elements from the non-empty list
        if not l1:
            point.next = l2
        else:
            point.next = l1
```

Accepted Runtime: 0 ms



143. Reorder List

You are given the head of a singly linked-list. The list can be represented as:

$L_0 \to L_1 \to \ldots \to L_{n-1} \to L_n$

Reorder the list to be on the following form:

$L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \ldots$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

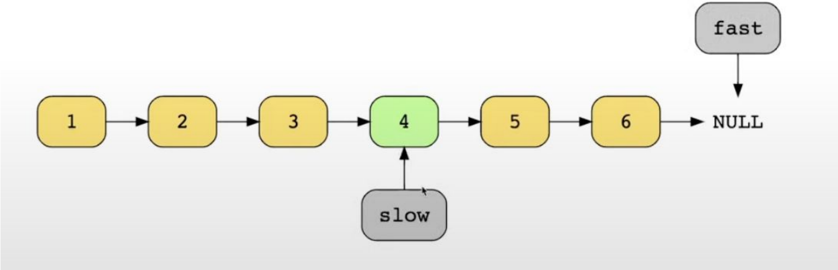
Example 1:

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

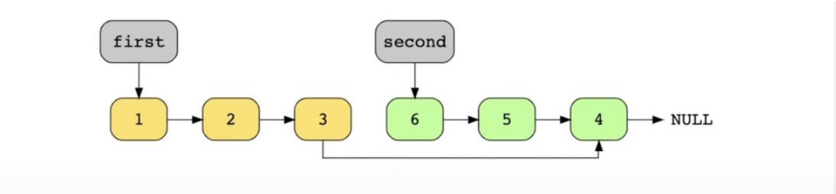
Example 2:

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

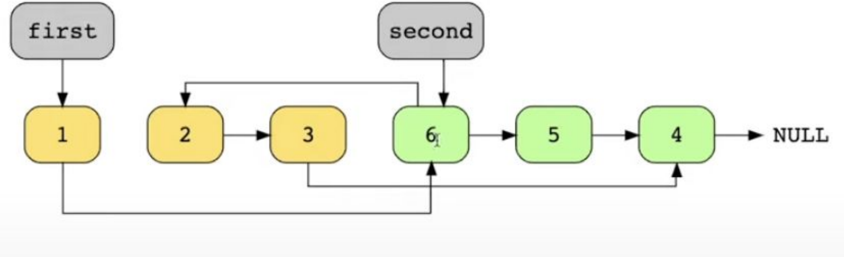
1. Find the middle of the linked list using the fast and slow pointers



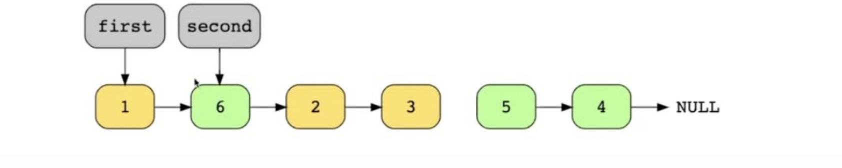
2. Reverse the second half of the linked list using in-place reversal



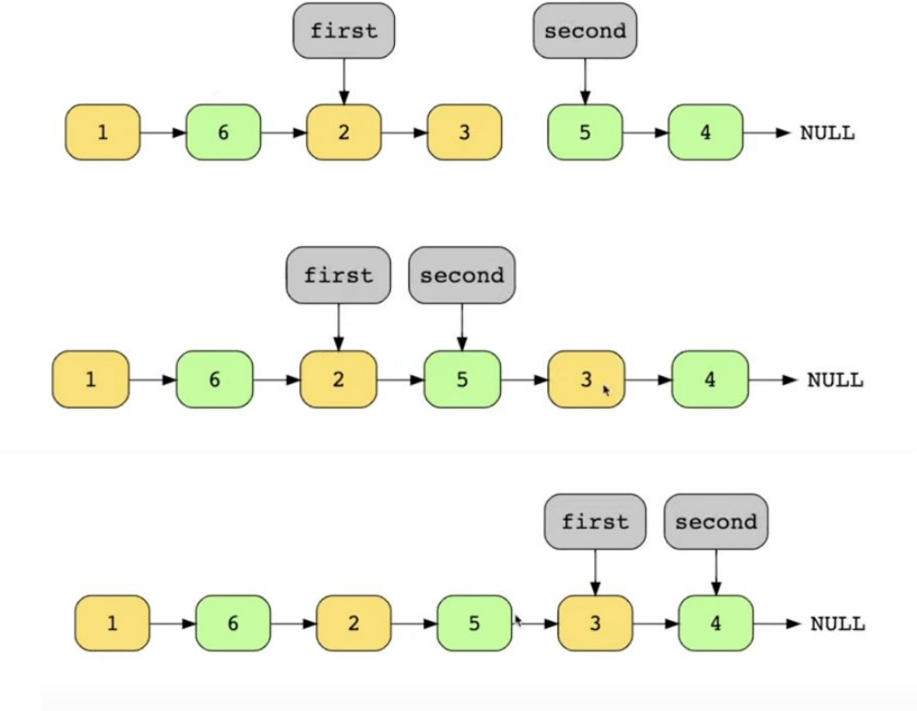
3. Point first.next at second and second.next at first.next

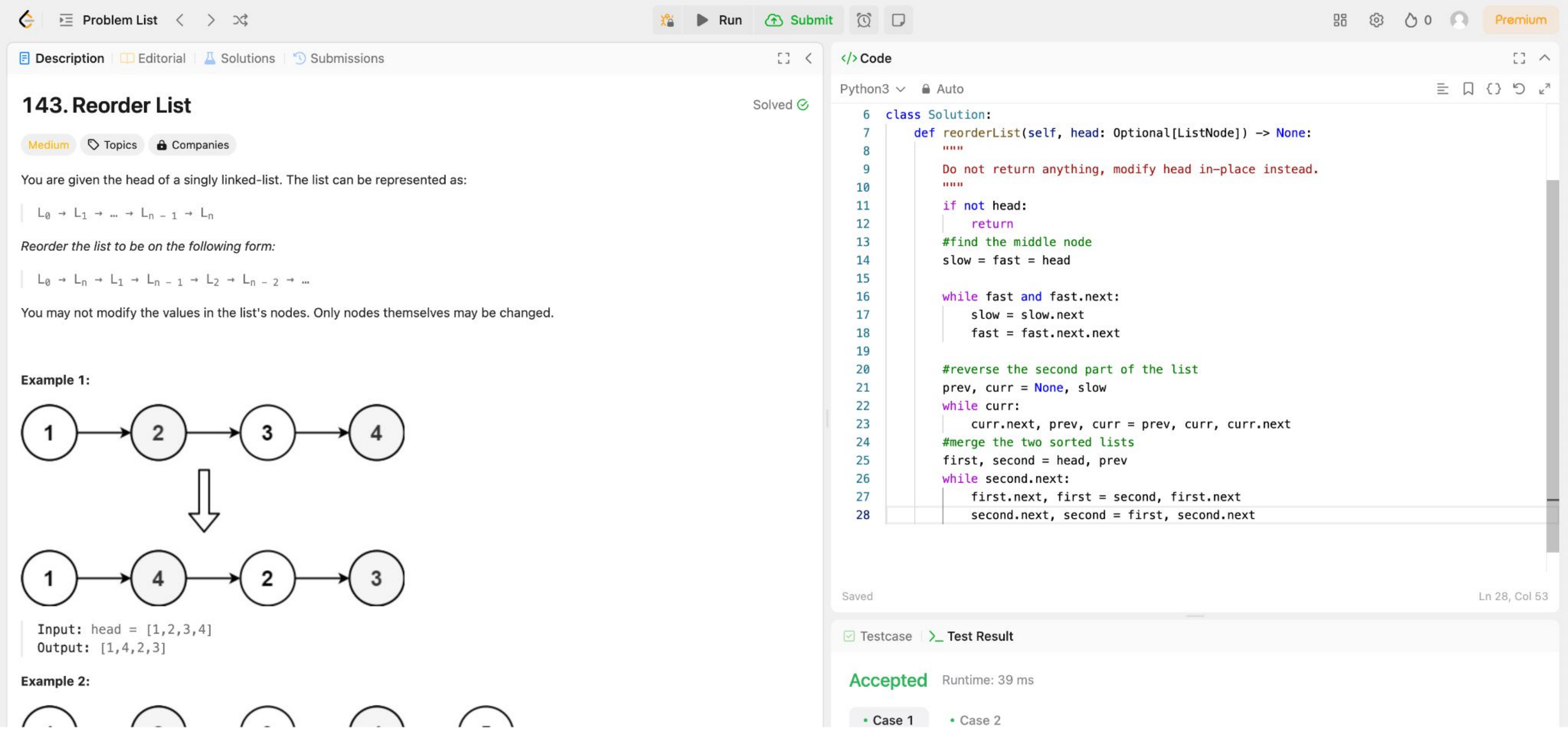


4. After first iteration



5. Move first and second forward





143. Reorder List

Solved ✓

Medium Topics Companies

You are given the head of a singly linked-list. The list can be represented as:

> $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$

Reorder the list to be on the following form:

> $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

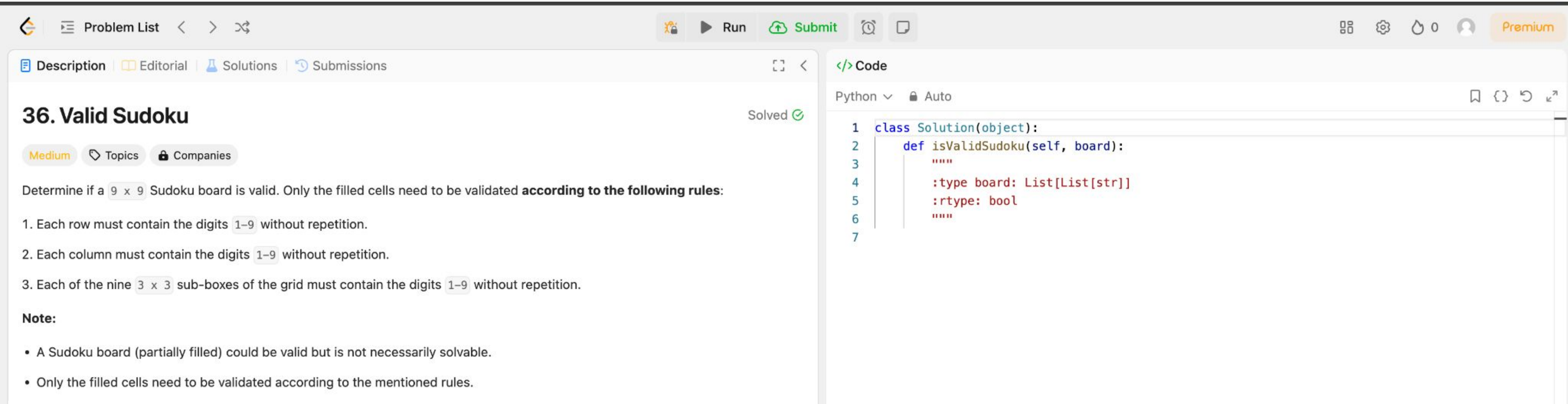
Example 1:

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

Example 2:

```python
class Solution:
    def reorderList(self, head: Optional[ListNode]) -> None:
        """
        Do not return anything, modify head in-place instead.
        """
        if not head:
            return
        #find the middle node
        slow = fast = head

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        #reverse the second part of the list
        prev, curr = None, slow
        while curr:
            curr.next, prev, curr = prev, curr, curr.next
        #merge the two sorted lists
        first, second = head, prev
        while second.next:
            first.next, first = second, first.next
            second.next, second = first, second.next
```

Accepted Runtime: 39 ms



36. Valid Sudoku

Medium · Topics · Companies

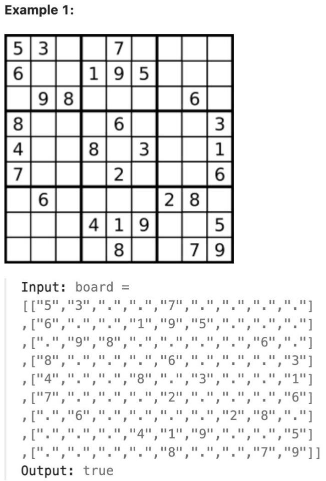
Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

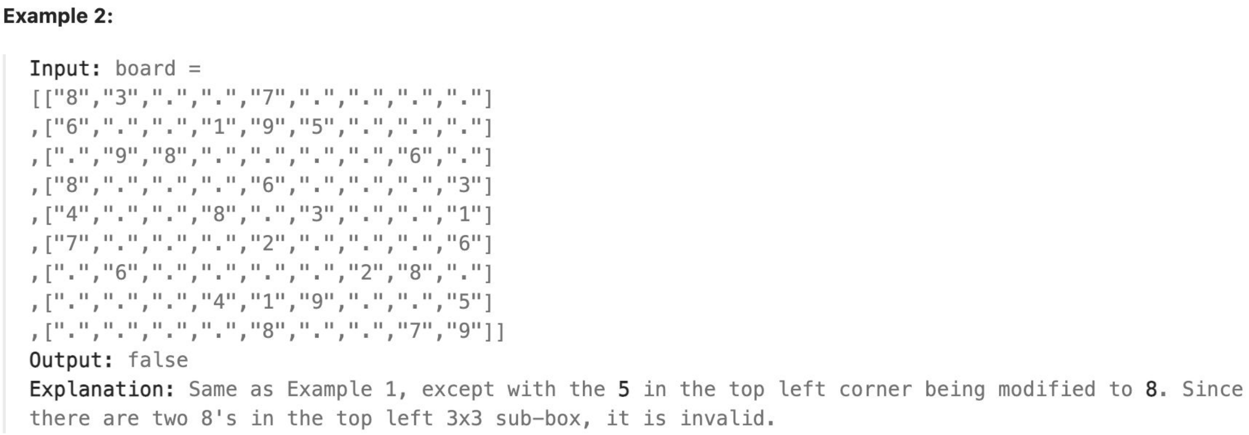
```python
class Solution(object):
    def isValidSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: bool
        """

```



Example 1:

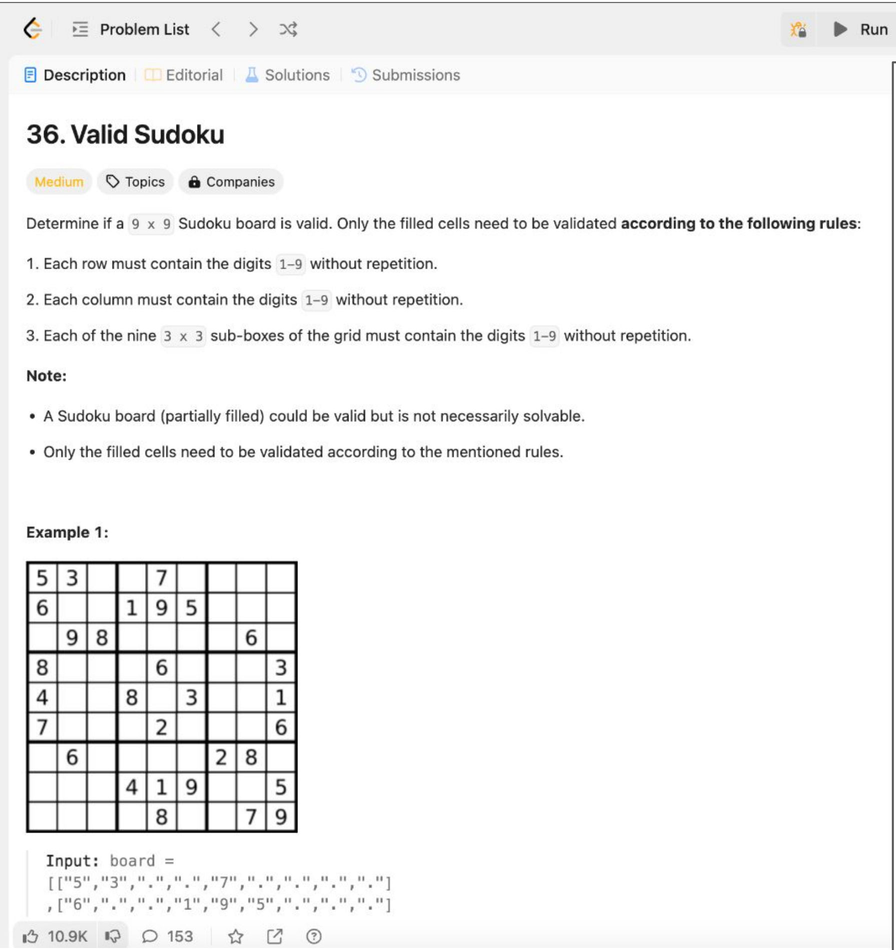
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
Input: board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: true
```



Example 2:

```
Input: board =
[["8","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.
```



36. Valid Sudoku

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

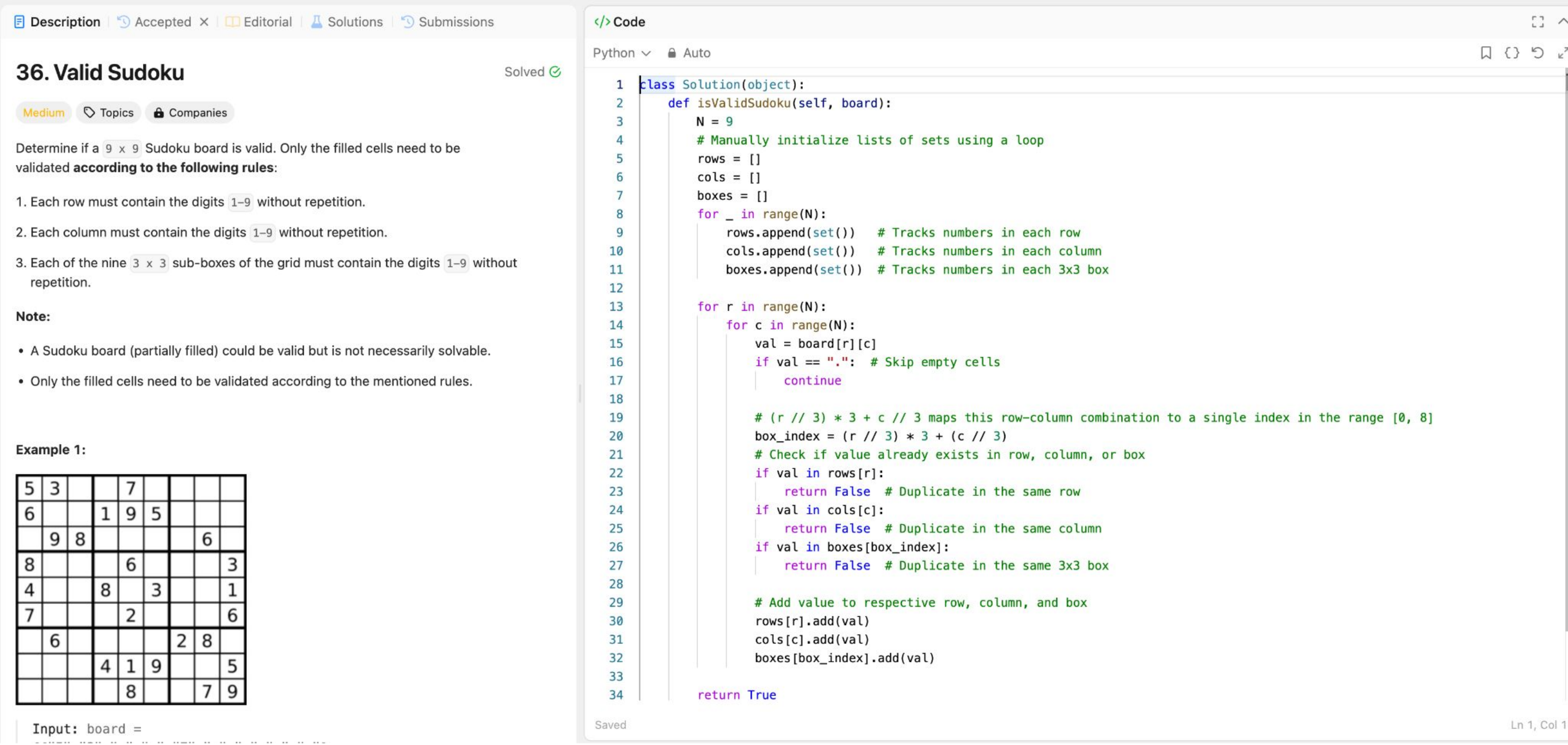
Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
Input: board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
```

`r // 3`: This part divides the row index `r` by 3 using integer division. It determines which of the three rows of 3x3 boxes the cell is in.

`c // 3`: determines which of the three columns of 3x3 boxes the cell is in



36. Valid Sudoku

Solved ✓

Medium · Topics · Companies

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Note:

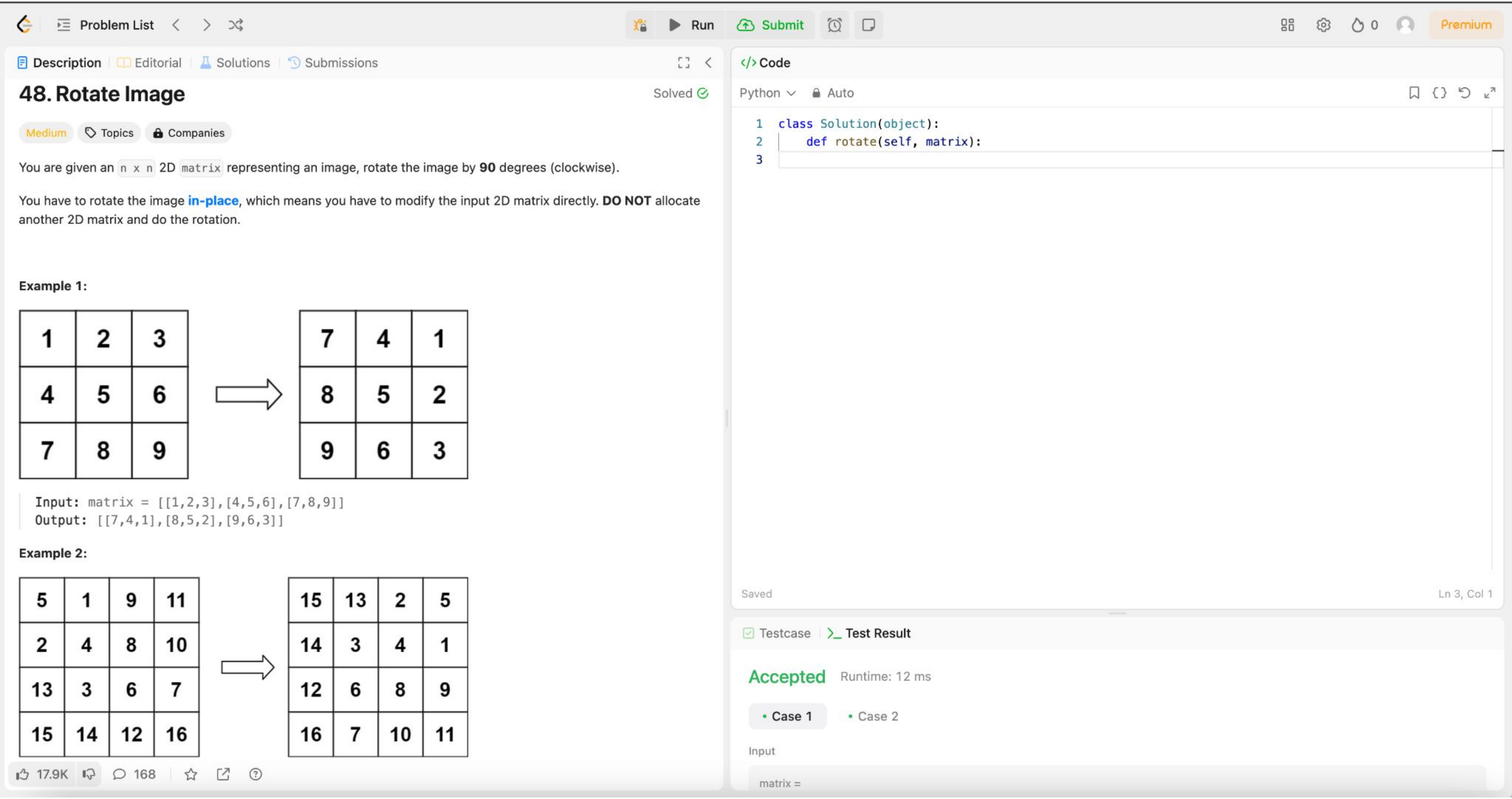
- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Input: board =

```python
class Solution(object):
    def isValidSudoku(self, board):
        N = 9
        # Manually initialize lists of sets using a loop
        rows = []
        cols = []
        boxes = []
        for _ in range(N):
            rows.append(set())   # Tracks numbers in each row
            cols.append(set())   # Tracks numbers in each column
            boxes.append(set())  # Tracks numbers in each 3x3 box

        for r in range(N):
            for c in range(N):
                val = board[r][c]
                if val == ".":  # Skip empty cells
                    continue

                # (r // 3) * 3 + c // 3 maps this row-column combination to a single index in the range [0, 8]
                box_index = (r // 3) * 3 + (c // 3)
                # Check if value already exists in row, column, or box
                if val in rows[r]:
                    return False  # Duplicate in the same row
                if val in cols[c]:
                    return False  # Duplicate in the same column
                if val in boxes[box_index]:
                    return False  # Duplicate in the same 3x3 box

                # Add value to respective row, column, and box
                rows[r].add(val)
                cols[c].add(val)
                boxes[box_index].add(val)

        return True
```



48. Rotate Image

You are given an `n x n` 2D `matrix` representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:

1	2	3
4	5	6
7	8	9

7	4	1
8	5	2
9	6	3

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

Example 2:

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

Python · Auto

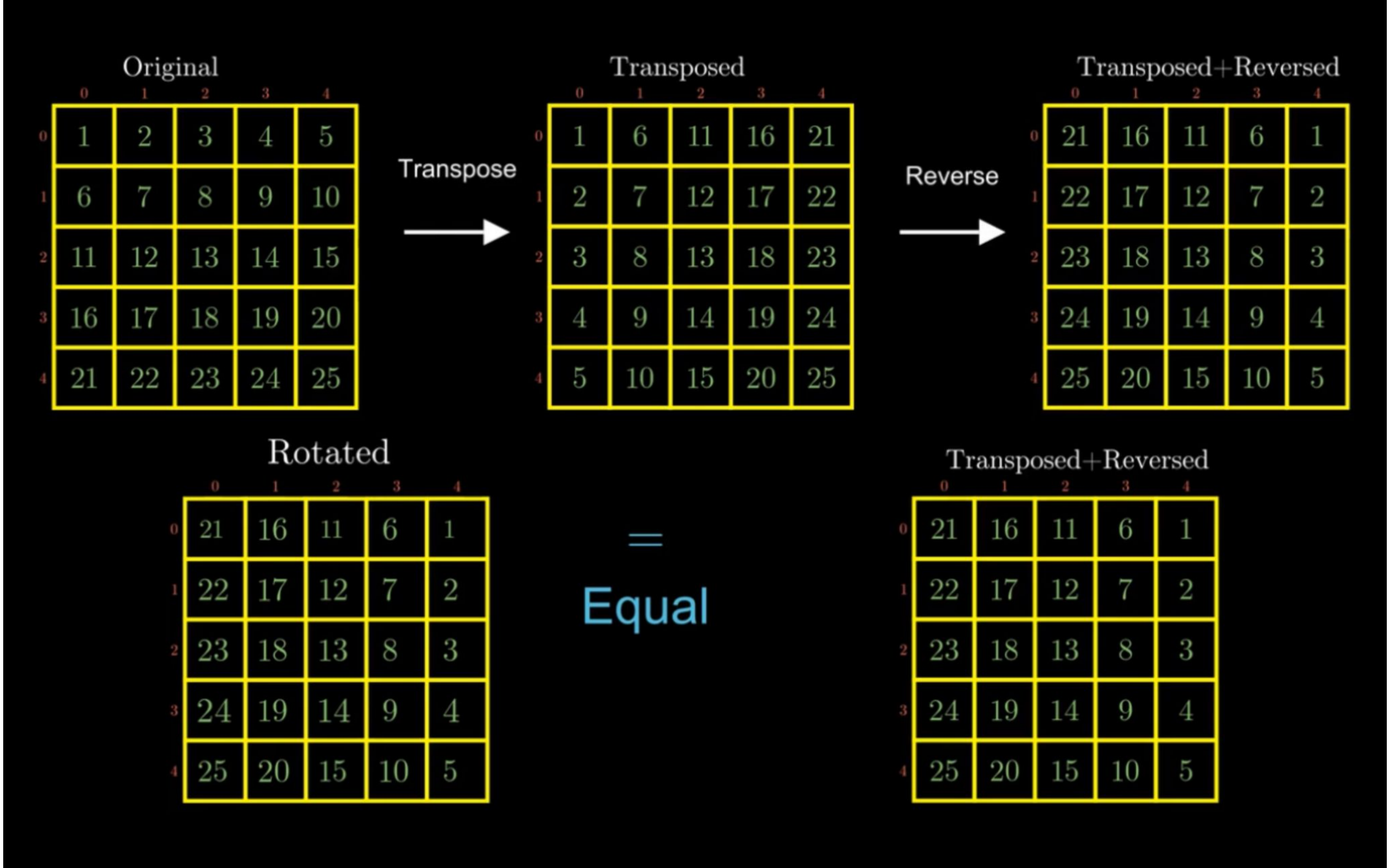
```python
class Solution(object):
    def rotate(self, matrix):

```

Saved   Ln 3, Col 1

Accepted   Runtime: 12 ms

Input

matrix =



Original

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20
4	21	22	23	24	25

Transpose

Transposed

	0	1	2	3	4
0	1	6	11	16	21
1	2	7	12	17	22
2	3	8	13	18	23
3	4	9	14	19	24
4	5	10	15	20	25

Reverse

Transposed+Reversed

	0	1	2	3	4
0	21	16	11	6	1
1	22	17	12	7	2
2	23	18	13	8	3
3	24	19	14	9	4
4	25	20	15	10	5

Rotated

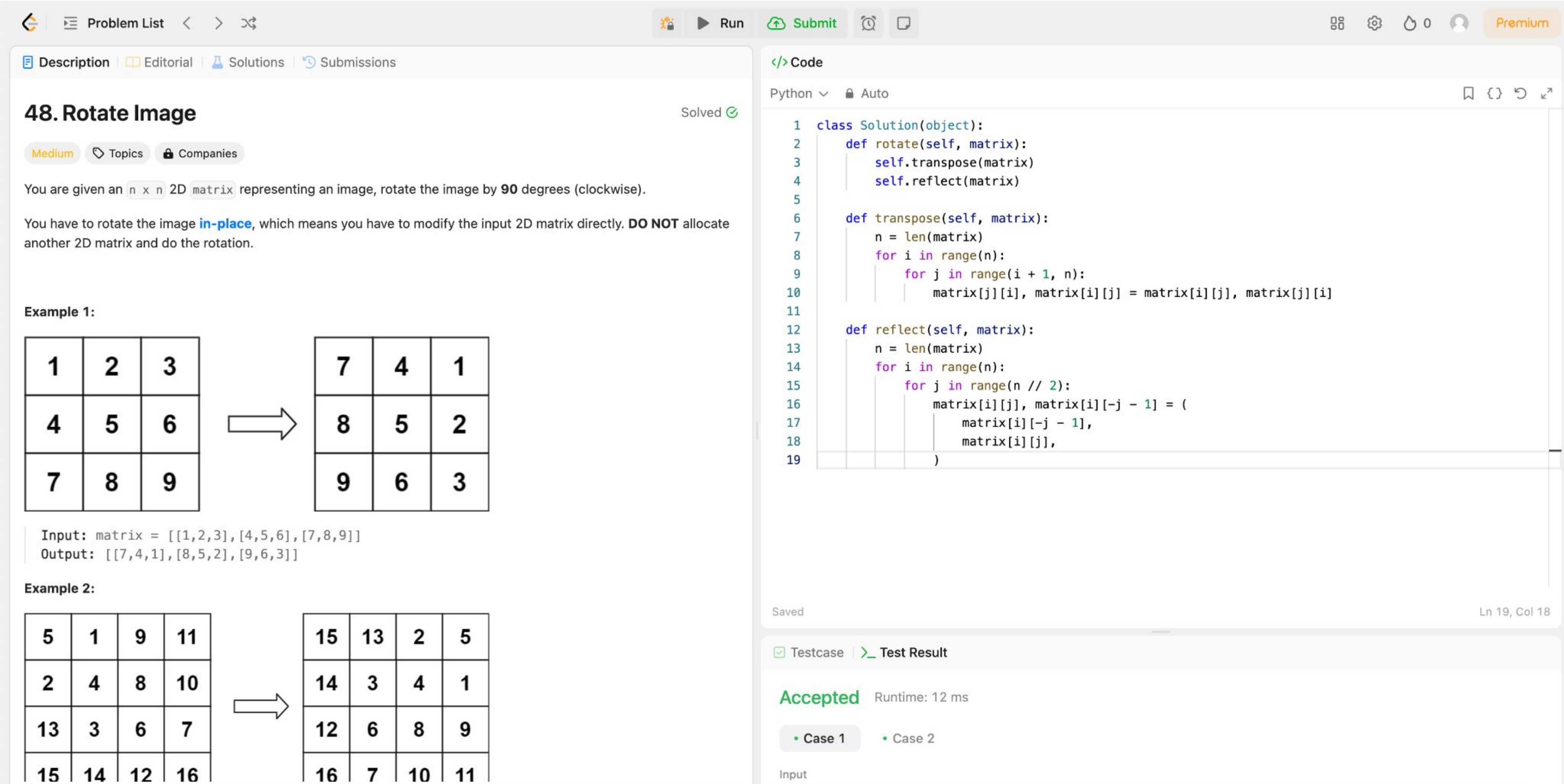
	0	1	2	3	4
0	21	16	11	6	1
1	22	17	12	7	2
2	23	18	13	8	3
3	24	19	14	9	4
4	25	20	15	10	5

=

Equal

Transposed+Reversed

	0	1	2	3	4
0	21	16	11	6	1
1	22	17	12	7	2
2	23	18	13	8	3
3	24	19	14	9	4
4	25	20	15	10	5



48. Rotate Image

Solved

Medium · Topics · Companies

You are given an `n x n` 2D `matrix` representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:

1	2	3
4	5	6
7	8	9

⇒

7	4	1
8	5	2
9	6	3

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

Example 2:

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

⇒

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

```python
class Solution(object):
    def rotate(self, matrix):
        self.transpose(matrix)
        self.reflect(matrix)

    def transpose(self, matrix):
        n = len(matrix)
        for i in range(n):
            for j in range(i + 1, n):
                matrix[j][i], matrix[i][j] = matrix[i][j], matrix[j][i]

    def reflect(self, matrix):
        n = len(matrix)
        for i in range(n):
            for j in range(n // 2):
                matrix[i][j], matrix[i][-j - 1] = (
                    matrix[i][-j - 1],
                    matrix[i][j],
                )
```

Saved

Ln 19, Col 18

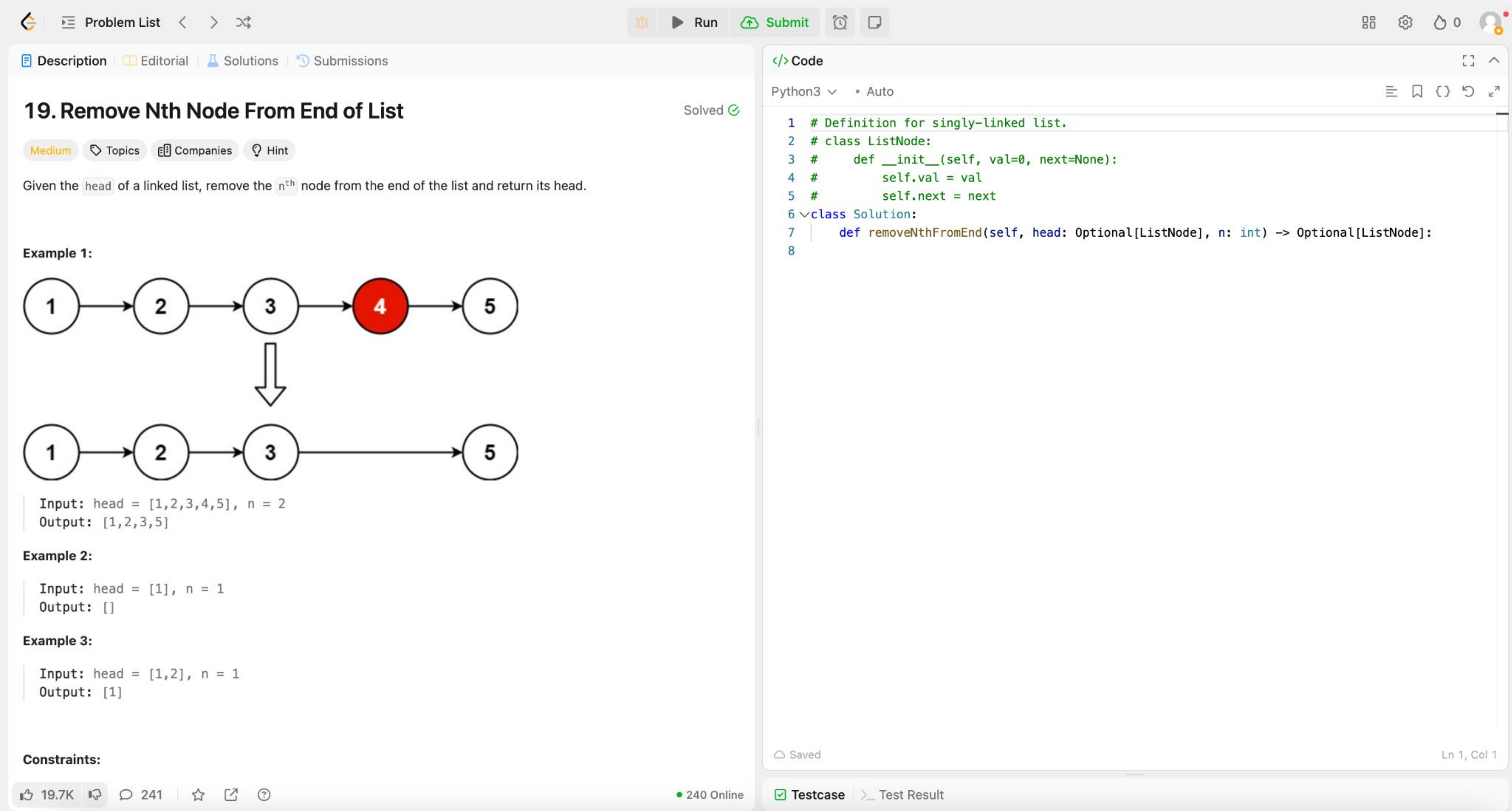
Testcase

Test Result

Accepted Runtime: 12 ms

• Case 1 • Case 2

Input



19. Remove Nth Node From End of List

Given the `head` of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:

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

Example 2:

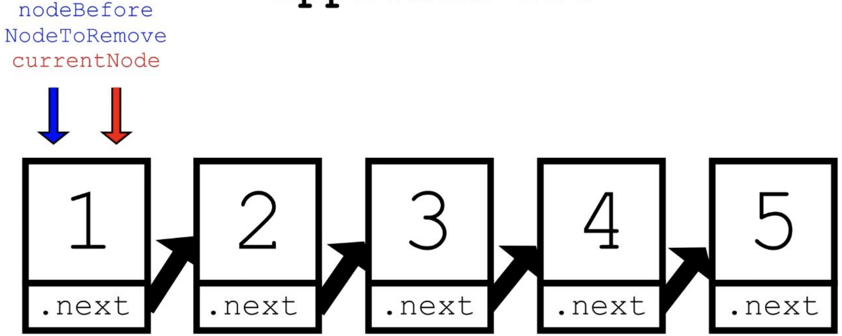
```
Input: head = [1], n = 1
Output: []
```

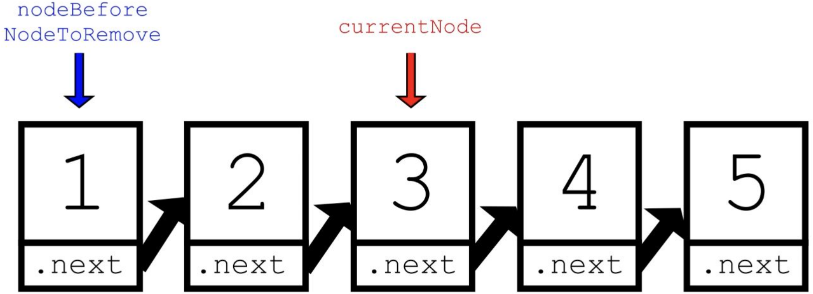
Example 3:

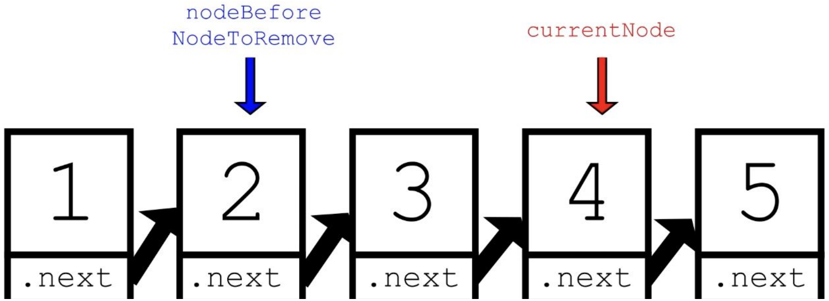
```
Input: head = [1,2], n = 1
Output: [1]
```

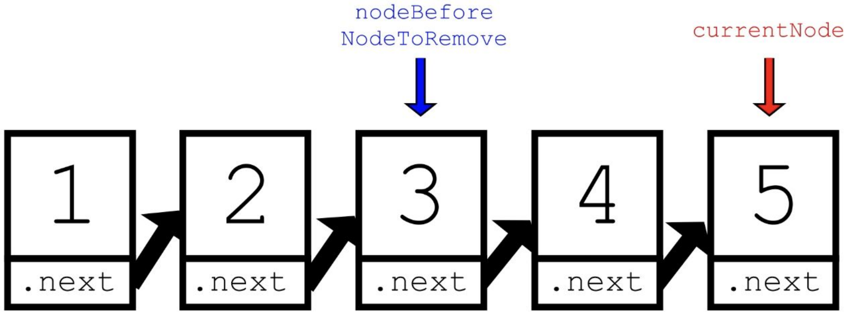
Constraints:

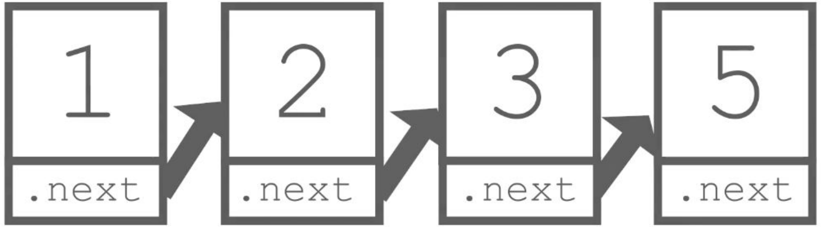
```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

```

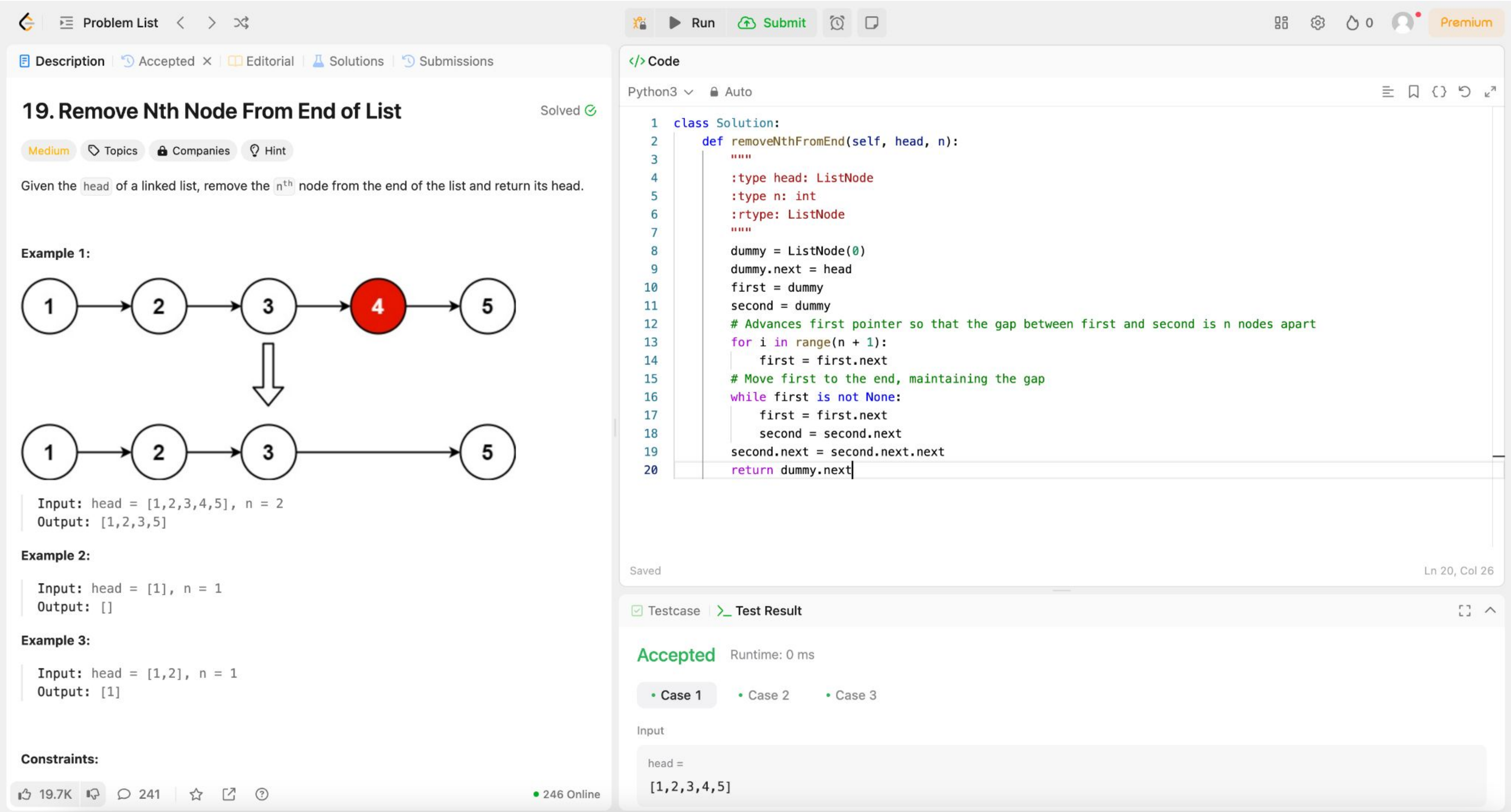

nodeBefore
NodeToRemove
currentNode
1
.next
2
.next
3
.next
4
.next
5
.next


nodeBefore
NodeToRemove
currentNode
1
.next
2
.next
3
.next
4
.next
5
.next


nodeBefore
NodeToRemove
currentNode
1
.next
2
.next
3
.next
4
.next
5
.next


nodeBefore
NodeToRemove
currentNode
1
.next
2
.next
3
.next
4
.next
5
.next


1
.next
2
.next
3
.next
5
.next



19. Remove Nth Node From End of List

Given the `head` of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:

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

Example 2:

```
Input: head = [1], n = 1
Output: []
```

Example 3:

```
Input: head = [1,2], n = 1
Output: [1]
```

Constraints:

```python
class Solution:
    def removeNthFromEnd(self, head, n):
        """
        :type head: ListNode
        :type n: int
        :rtype: ListNode
        """
        dummy = ListNode(0)
        dummy.next = head
        first = dummy
        second = dummy
        # Advances first pointer so that the gap between first and second is n nodes apart
        for i in range(n + 1):
            first = first.next
        # Move first to the end, maintaining the gap
        while first is not None:
            first = first.next
            second = second.next
        second.next = second.next.next
        return dummy.next
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =

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