**EXPERIMENT 1.3**

**NAME: Aaditya Maheshwari UID: 22BET10094**

**BRANCH: IT SECTION:22BET\_IOT-702(B)**

**SEMESTER: 6th DOP: 31/01/2025**

**SUBJECT: Advance Programming SUBJECT CODE: 22ITP-351**

**Aim:** Detect a cycle in a linked list

**Objective:** There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

**Code:**

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

class Solution:

def hasCycle(self, head):

slow = head

fast = head

while fast and fast.next:

slow = slow.next

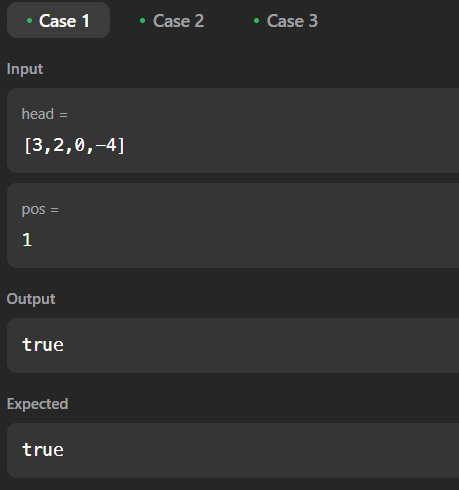
fast = fast.next.next

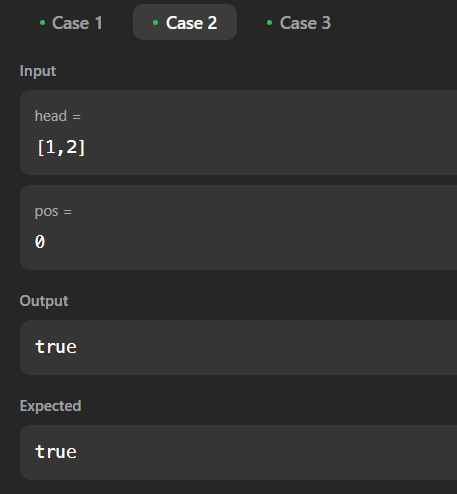
if slow == fast:

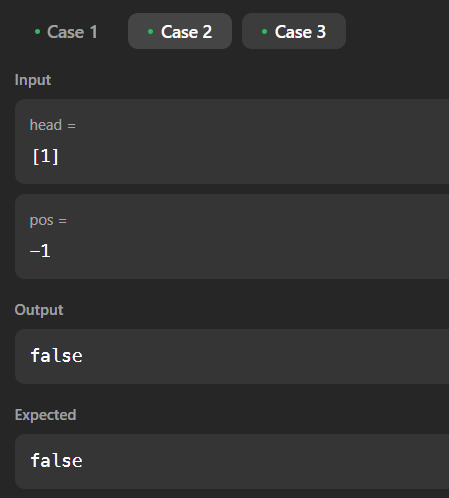
return True

return False

**OUTPUT:**

****

****



**Aim:** Reverse linked list 2

**Objective:** Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.

**Code:**

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

class Solution:

    def reverseBetween(self, head, left, right):

        if not head or left == right:

            return head

        dummy = ListNode(0)

        dummy.next = head

        prev = dummy

        for \_ in range(left - 1):

            prev = prev.next

        curr = prev.next

        next\_node = None

        for \_ in range(right - left):

            next\_node = curr.next

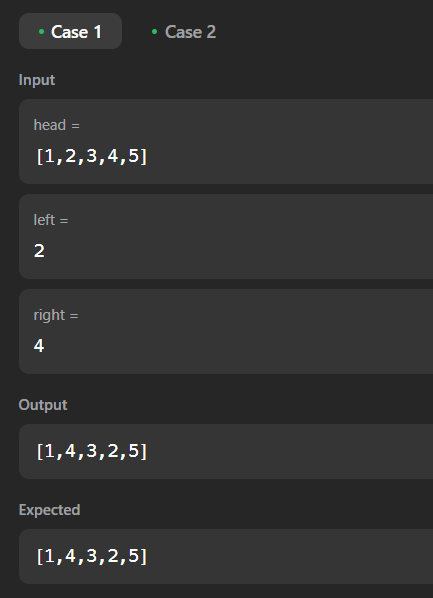
            curr.next = next\_node.next

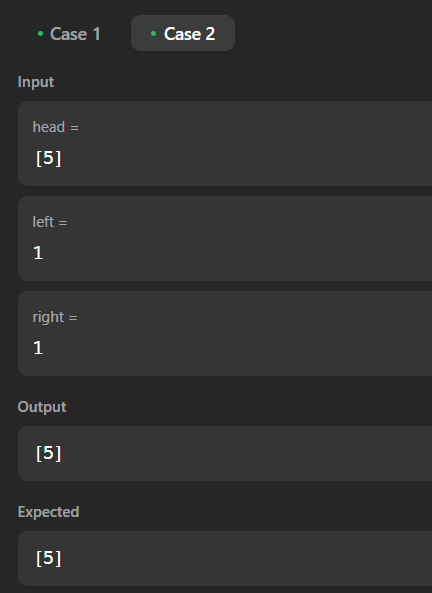
            next\_node.next = prev.next

            prev.next = next\_node

        return dummy.next

**OUTPUT:**



****

**Aim:** rotate a list

**Objective:** Given the head of a linked list, rotate the list to the right by k places.

**Code:**

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

class Solution:

    def rotateRight(self, head, k):

        if not head or not head.next or k == 0:

            return head

        # Compute the length of the list

        length = 1

        tail = head

        while tail.next:

            tail = tail.next

            length += 1

        # Make it a circular list

        tail.next = head

        # Find the new tail position

        k = k % length

        steps\_to\_new\_tail = length - k

        new\_tail = head

        for \_ in range(steps\_to\_new\_tail - 1):

            new\_tail = new\_tail.next

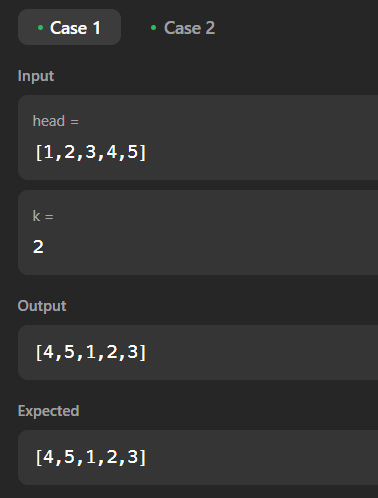
        # Break the circle and set the new head

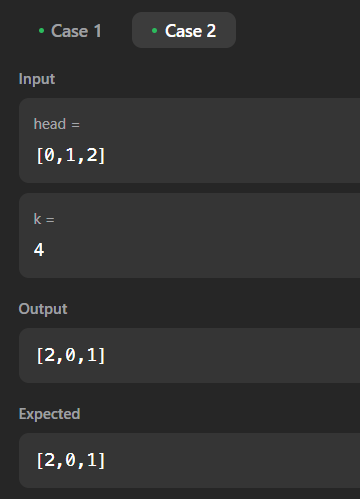
        new\_head = new\_tail.next

        new\_tail.next = None

        return new\_head

**OUTPUT:**



****

**Aim:** Merge k sorted lists

**Objective:** 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.*

**Code:**

import heapq

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

class Solution:

    def mergeKLists(self, lists):

        heap = []

        # Push the head nodes of all lists into the heap

        for i, node in enumerate(lists):

            if node:

                heapq.heappush(heap, (node.val, i, node))

        dummy = ListNode(0)

        curr = dummy

        while heap:

            val, i, node = heapq.heappop(heap)

            curr.next = node

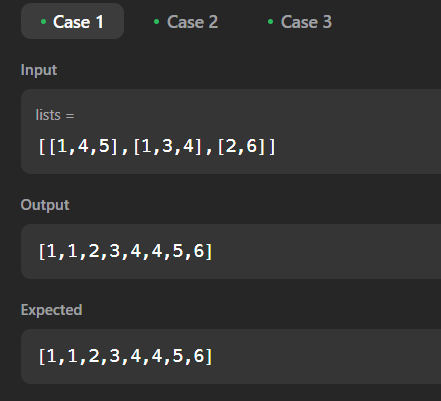
            curr = curr.next

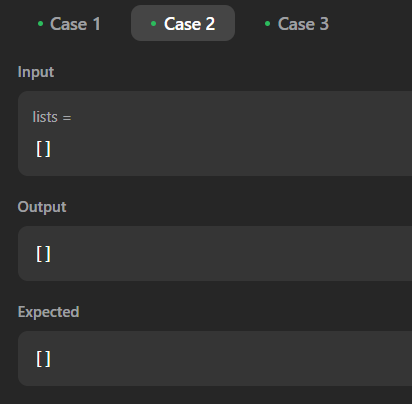
            if node.next:

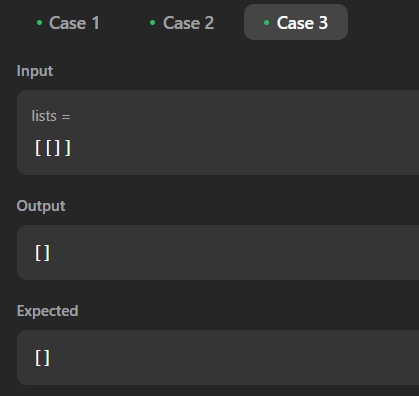
                heapq.heappush(heap, (node.next.val, i, node.next))

        return dummy.next

**OUTPUT:**



****

****

**Aim:** Sort List

**OBJECTIVE:** Given the head of a linked list, return *the list after sorting it in ascending order*.

**CODE:**

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

class Solution:

    def sortList(self, head):

        if not head or not head.next:

            return head

        # Split the list into halves

        slow, fast = head, head.next

        while fast and fast.next:

            slow = slow.next

            fast = fast.next.next

        mid = slow.next

        slow.next = None

        # Recursively sort both halves

        left = self.sortList(head)

        right = self.sortList(mid)

        # Merge the sorted halves

        return self.merge(left, right)

    def merge(self, l1, l2):

        dummy = ListNode(0)

        curr = dummy

        while l1 and l2:

            if l1.val < l2.val:

                curr.next = l1

                l1 = l1.next

            else:

                curr.next = l2

                l2 = l2.next

            curr = curr.next

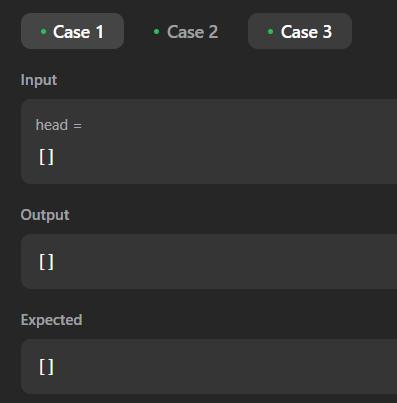
        curr.next = l1 if l1 else l2

        return dummy.next

**OUTPUT:**

****

****

****