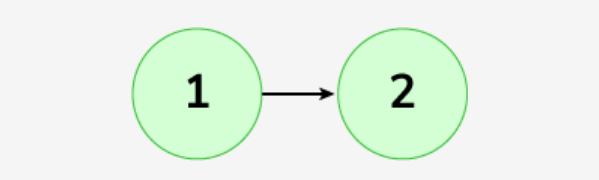
Given a linked list. Print all the elements of the linked list separated by space followed.

Examples:

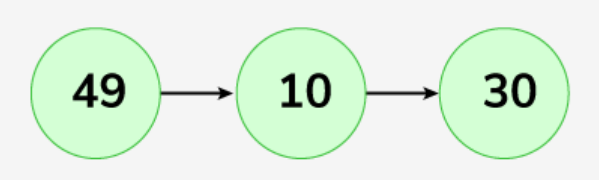
Input: LinkedList : 1 -> 2



Output: 1 2

Explanation: The linked list contains two elements 1 and 2.The elements are printed in a single line.

Input: Linked List : 49 -> 10 -> 30



Output: 49 10 30

Explanation: The linked list contains 3 elements 49, 10 and 30. The elements are printed in a single line.

Constraints :

1 <= numbers of nodes <= 10^5

1 <= node values <= 10^6

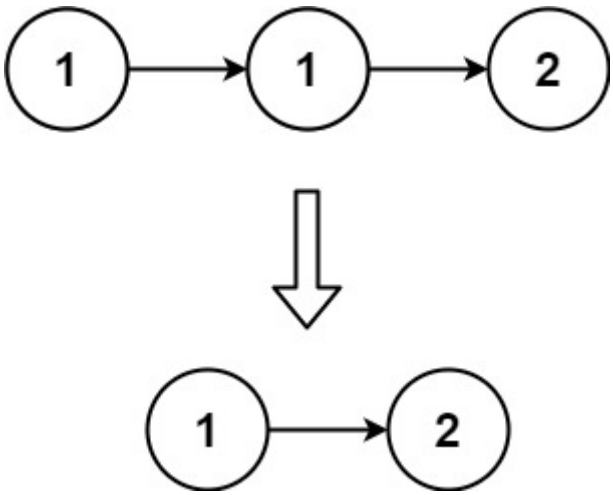
Try more examples

```
class Solution {
public:
   ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
      if (!list1) return list2;
      if (!list2) return list1;
      if (list1->val <= list2->val) {
         list1->next = mergeTwoLists(list1->next, list2);
         return list1;
      } else {
         list2->next = mergeTwoLists(list1, list2->next);
         return list2;
      }
   }
```

};

2 Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

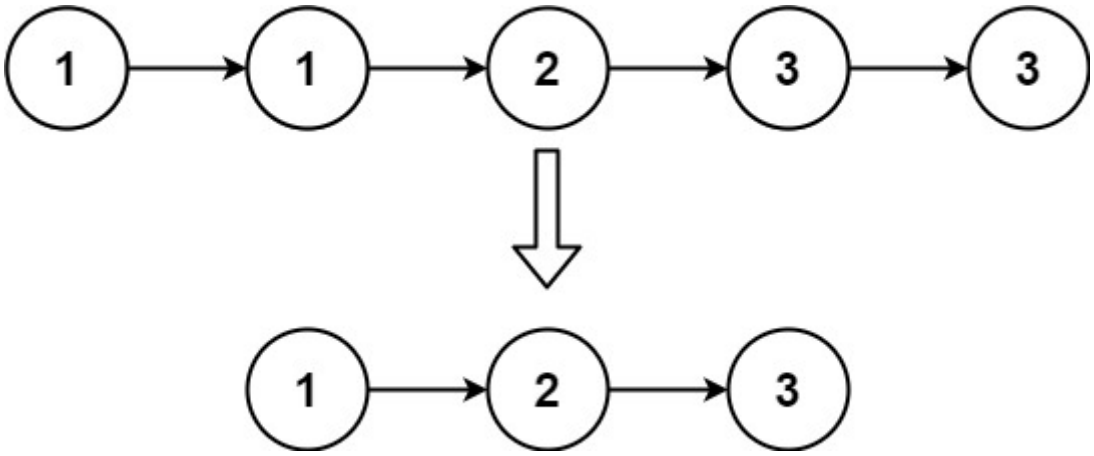
Example 1:



Input: head = [1,1,2]

Output: [1,2]

Example 2:



Input: head = [1,1,2,3,3]

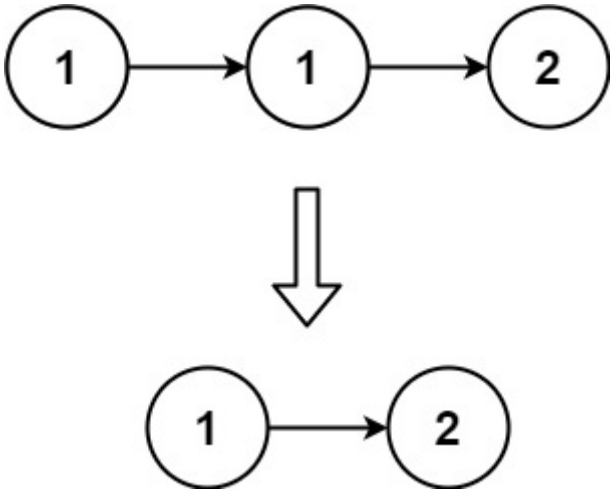
Output: [1,2,3]

Constraints:

- The number of nodes in the list is in the range [0, 300].
- -100 <= Node.val <= 100
- The list is guaranteed to be **sorted** in ascending order.

Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

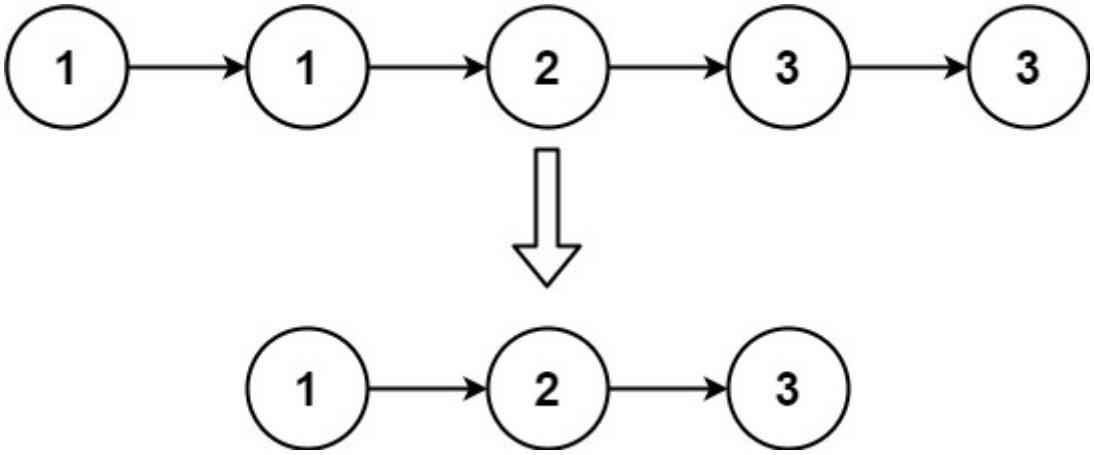
Example 1:



Input: head = [1,1,2]

Output: [1,2]

Example 2:



Input: head = [1,1,2,3,3]

Output: [1,2,3]

Constraints:

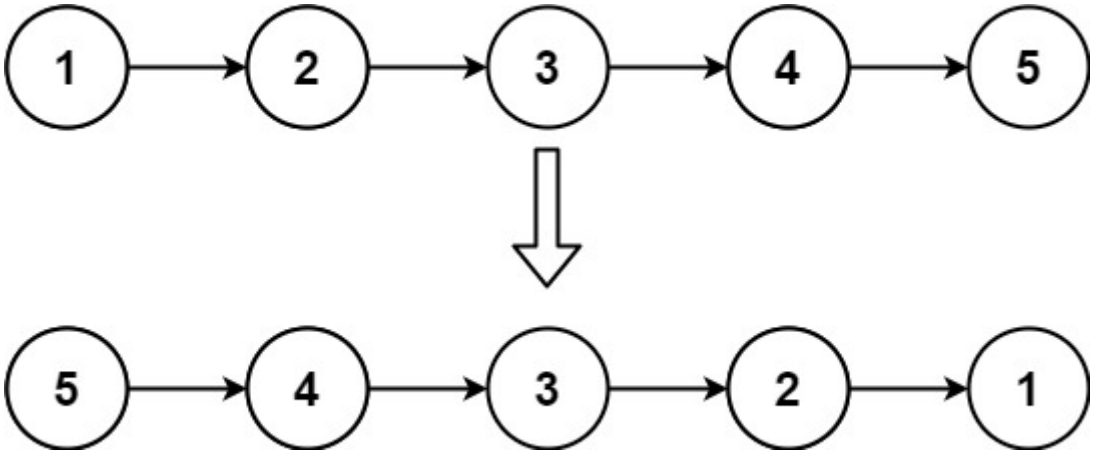
- The number of nodes in the list is in the range [0, 300].
- -100 <= Node.val <= 100
- The list is guaranteed to be **sorted** in ascending order.

```
class Solution {
```

```
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* cur = head;
        while (cur != nullptr && cur->next != nullptr) {
            if (cur->val == cur->next->val) {
                cur->next = cur->next->next;
            } else {
                cur = cur->next;
            }
        }
        return head;
    }
};
```

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

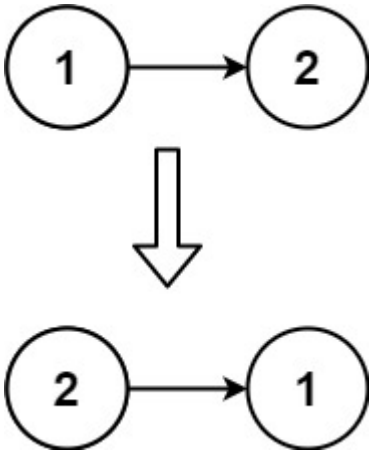
Example 1:



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

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

Constraints:

- The number of nodes in the list is the range [0, 5000].
- -5000 <= Node.val <= 5000

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

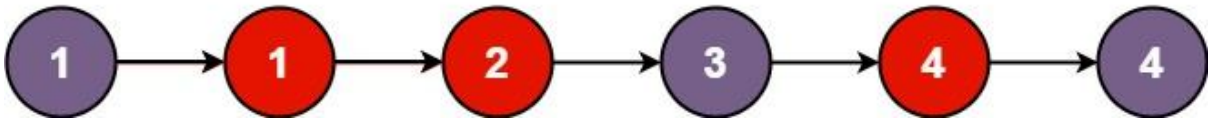
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* dummy = new ListNode();
        ListNode* curr = head;
        while (curr) {
            ListNode* next = curr->next;
            curr->next = dummy->next;
            dummy->next = curr;
            curr = next;
        }
        return dummy->next;
    }
};
```

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*.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- -100 <= Node.val <= 100
- Both list1 and list2 are sorted in **non-decreasing** order.

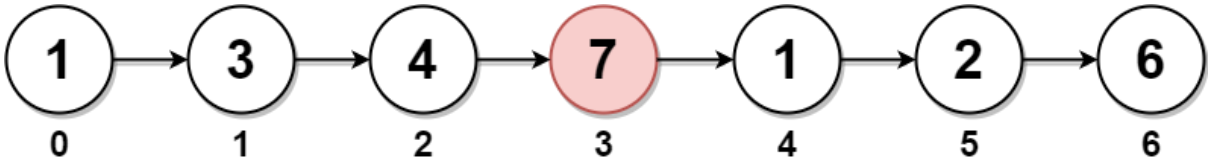
```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;
        if (list1->val <= list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};
```

4 You are given the head of a linked list. **Delete** the **middle node**, and return *the* head *of the modified linked list*.

The **middle node** of a linked list of size n is the $\lfloor n / 2 \rfloor^{th}$ node from the **start** using **0-based indexing**, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

- For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.

Example 1:



Input: head = [1,3,4,7,1,2,6]

Output: [1,3,4,1,2,6]

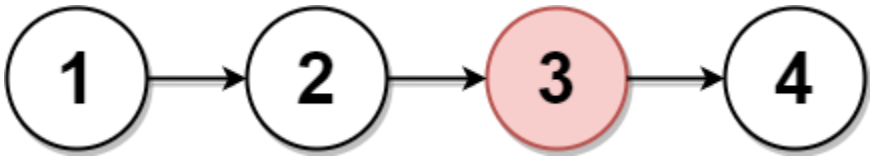
Explanation:

The above figure represents the given linked list. The indices of the nodes are written below.

Since n = 7, node 3 with value 7 is the middle node, which is marked in red.

We return the new list after removing this node.

Example 2:



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

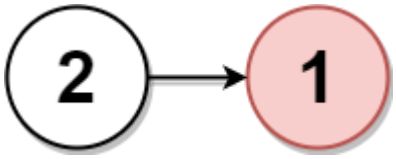
Output: [1,2,4]

Explanation:

The above figure represents the given linked list.

For n = 4, node 2 with value 3 is the middle node, which is marked in red.

Example 3:



Input: head = [2,1]

Output: [2]

Explanation:

The above figure represents the given linked list.

For n = 2, node 1 with value 1 is the middle node, which is marked in red.

Node 0 with value 2 is the only node remaining after removing node 1.

Constraints:

- The number of nodes in the list is in the range [1, 10^5].
- 1 <= Node.val <= 10^5

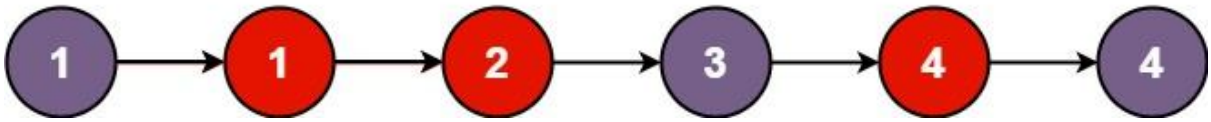
```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        ListNode* dummy = new ListNode(0, head);
        ListNode* slow = dummy;
        ListNode* fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        slow->next = slow->next->next;
        return dummy->next;
    }
};
```

5 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*.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

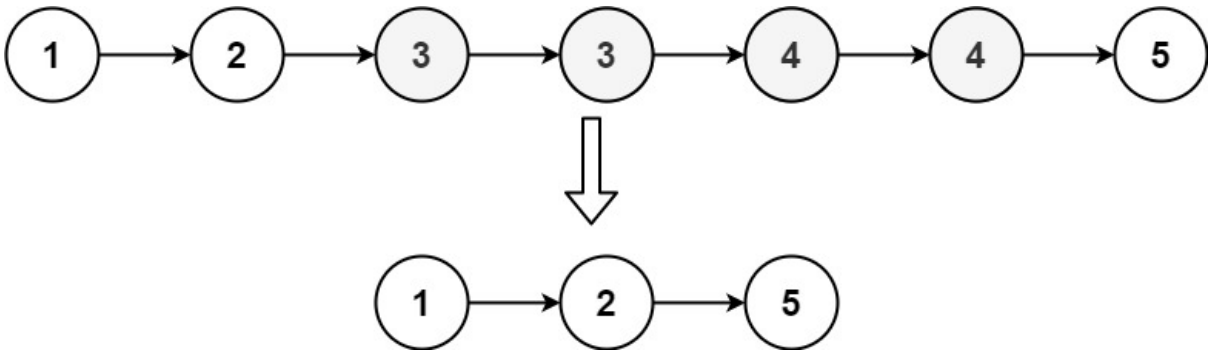
Constraints:

- The number of nodes in both lists is in the range [0, 50].
- -100 <= Node.val <= 100
- Both list1 and list2 are sorted in **non-decreasing** order.

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;
        if (list1->val <= list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};
```

6 Given the head of a sorted linked list, *delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list*. Return *the linked list* ***sorted*** *as well*.

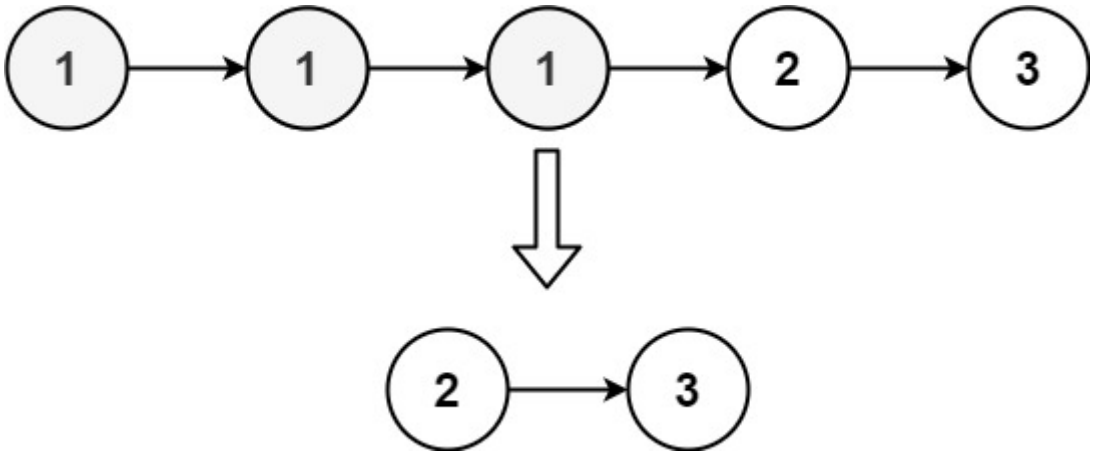
Example 1:



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

Output: [1,2,5]

Example 2:



Input: head = [1,1,1,2,3]

Output: [2,3]

Constraints:

- The number of nodes in the list is in the range [0, 300].
- -100 <= Node.val <= 100
- The list is guaranteed to be **sorted** in ascending order.

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        // Create a dummy node that points to the head of the list
        ListNode* dummyNode = new ListNode(0, head);
```

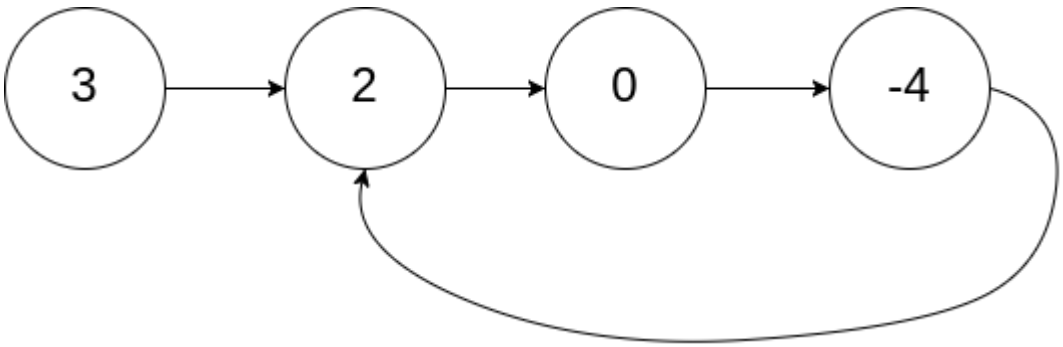
```
        ListNode* pred = dummyNode; // Predecessor node that trails behind
the current node
        ListNode* curr = head; // Current node we're examining

        // Traverse the list
        while (curr) {
            // Skip duplicate nodes
            while (curr->next && curr->next->val == curr->val) {
                curr = curr->next;
            }
            // If the predecessor's next is the current node, no duplicates
were found
            // Move the predecessor to the current node
            if (pred->next == curr) {
                pred = curr;
            } else {
                // Skip the duplicate nodes by linking the predecessor's
next to the current's next
                pred->next = curr->next;
            }
            // Move to the next node
            curr = curr->next;
        }
        // Return the modified list, excluding the dummy node
        return dummyNode->next;
    }
};

```

7 Given head, the head of a linked list, determine if the linked list has a cycle in it.

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**.

Return true *if there is a cycle in the linked list*. Otherwise, return false.

Example 1:

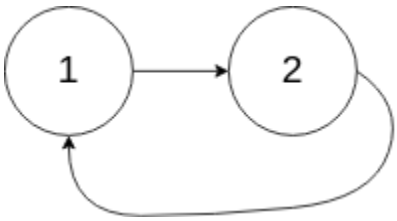


Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:

Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range [0, 10^4].
- -10^5 <= Node.val <= 10^5
- pos is -1 or a **valid index** in the linked-list.

Follow up: Can you solve it using O(1) (i.e. constant) memory?

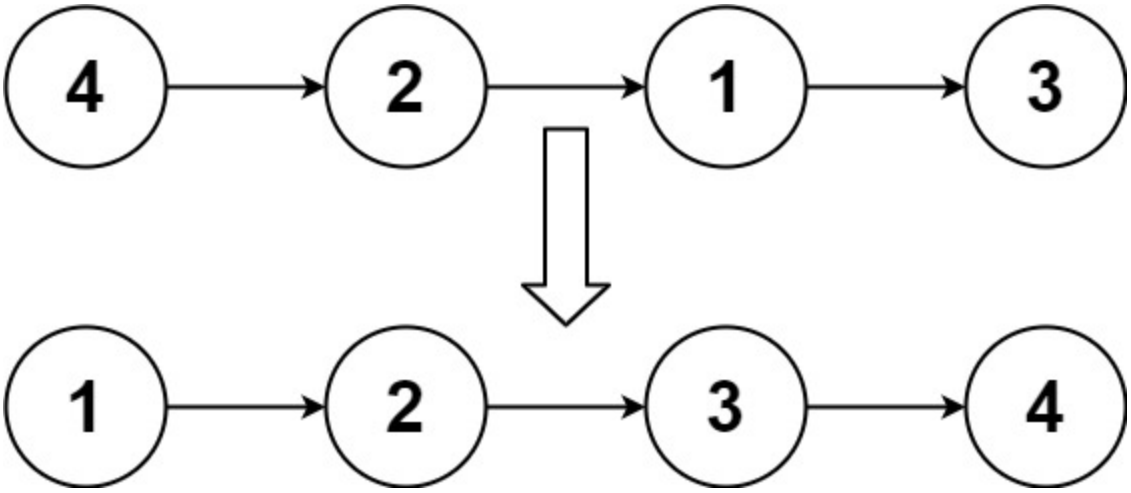
```
class Solution {
public:
    bool hasCycle(ListNode* head) {
        unordered_set<ListNode*> s;
        for (; head; head = head->next) {
            if (s.contains(head)) {
                return true;
```

```
            }
            s.insert(head);
        }
        return false;
    }
};
8 class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        // If the head is null or there is only one node, return the head as rotation isn't needed
        if (!head || !head->next) {
            return head;
        }

        // find the length of the linked list
        ListNode* current = head;
        int length = 0;
        while (current) {
            ++length;
            current = current->next;
        }

        // Normalize k to prevent unnecessary rotations if k >= length
        k %= length;

        // If k is 0 after modulo operation, no rotation is needed; return the original head
        if (k == 0) {
            return head;
        }

        // Set two pointers, fast and slow initially at head
```

```
        ListNode* fast = head;
        ListNode* slow = head;

        // Move fast pointer k steps ahead
        while (k--) {
            fast = fast->next;
        }

        // Move both slow and fast pointers until fast reaches the end of the list
        while (fast->next) {
            fast = fast->next;
            slow = slow->next;
        }

        // The slow pointer now points to the node just before rotation point
        // The fast pointer points to the last node of the list
        ListNode* newHead = slow->next; // This will be the new head after rotation
        slow->next = nullptr; // Break the chain to form a new end of the list
        fast->next = head; // Connect the original end of the list to the original head

        // Return the new head of the rotated list
        return newHead;
    }
};
```

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

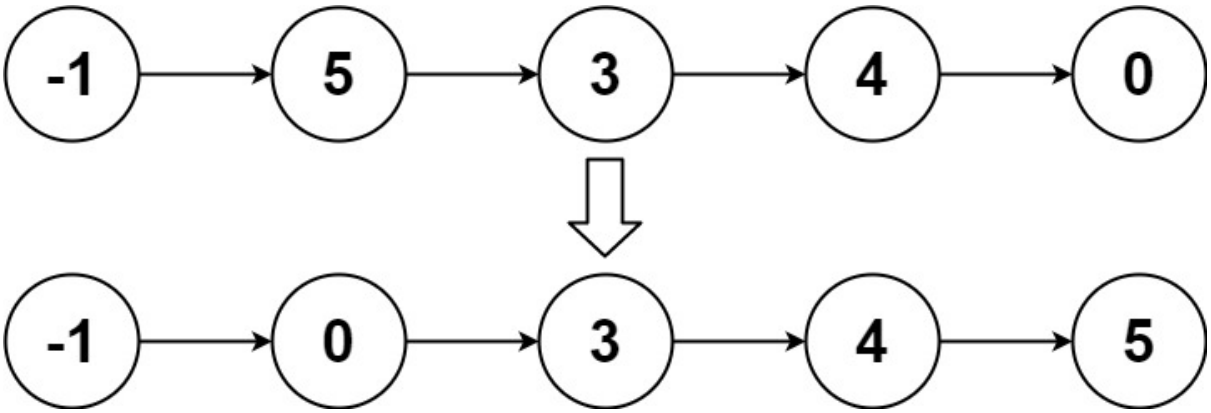
Example 1:



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

Output: [1,2,3,4]

Example 2:



Input: head = [-1,5,3,4,0]

Output: [-1,0,3,4,5]

Example 3:

Input: head = []

Output: []

Constraints:

- The number of nodes in the list is in the range [0, 5 * 10^4].
- -10^5 <= Node.val <= 10^5

Follow up: Can you sort the linked list in O(n logn) time and O(1) memory (i.e. constant space)?

class Solution {

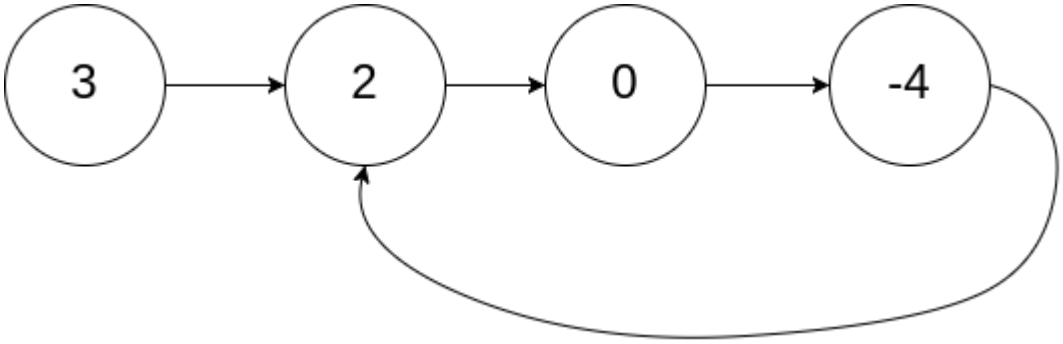
```
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *slow = head, *fast = head, *prev = NULL;
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = NULL;
        ListNode *l1 = sortList(head);
        ListNode *l2 = sortList(slow);
        return merge(l1, l2);
    }

    ListNode* merge(ListNode *l1, ListNode *l2) {
        if (!l1) return l2;
        if (!l2) return l1;
        if (l1->val < l2->val) {
            l1->next = merge(l1->next, l2);
            return l1;
        }
        l2->next = merge(l1, l2->next);
        return l2;
    }
};
```

9 Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null.

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 (**0-indexed**). It is -1 if there is no cycle. **Note that** pos **is not passed as a parameter**.

Do not modify the linked list.

Example 1:

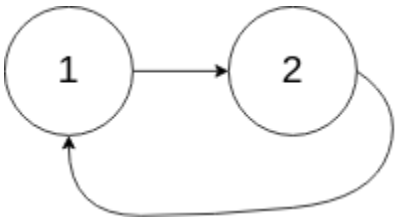


Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:



Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:

Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range $[0, 10^4]$.
- $-10^5 <= Node.val <= 10^5$

- pos is -1 or a **valid index** in the linked-list.

Follow up: Can you solve it using O(1) (i.e. constant) memory?

```
public class Solution {
   public ListNode detectCycle(ListNode head) {
      if(head == null){
         return null;
      }

      ListNode fast = head ;
      ListNode slow = head ;
      ListNode ptr = head ;
      while(fast!=null && fast.next!=null){
         fast = fast.next.next;
         slow = slow.next;

         if(slow==fast){ // there is a cycle in ll

            while(ptr!=slow){
               ptr = ptr.next;
               slow = slow.next;
            }

            return ptr ;
         }
      }

      return null;
   }
}
```