# Linked List-3

## Lesson Plan

PW SKILLS

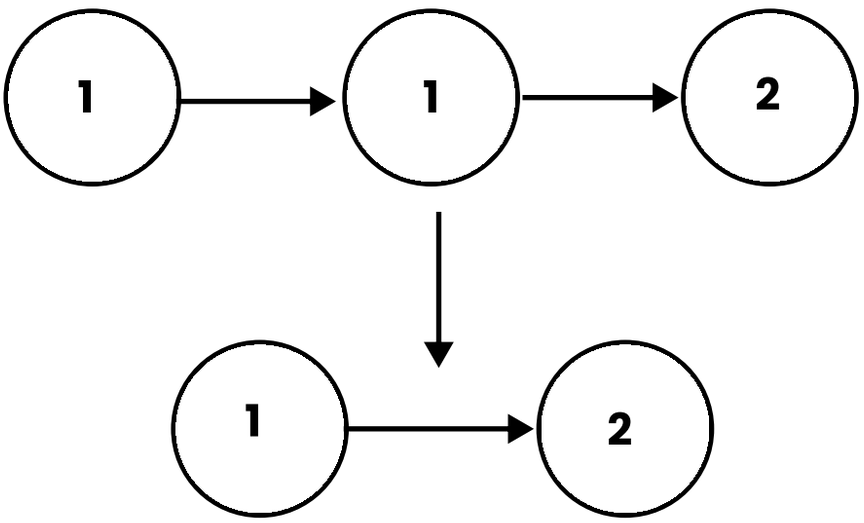
# Today's Checklist:

1. Remove Duplicates from the Sorted List (Leetcode-83)
2. Rotate List (Leetcode-61)
3. Spiral Matrix IV (Leetcode-2326)
4. Merge 2 sorted lists (Leetcode-21)
5. Merge k sorted lists (Leetcode-23)
6. Sort List (Leetcode-148)
7. Partition List (Leetcode-86)
8. Reverse Linked List (Leetcode-206)
9. Palindrome Linked List (Leetcode-234)
10. Reverse Linked List II (Leetcode-92)
11. Reorder List (Leetcode-143)

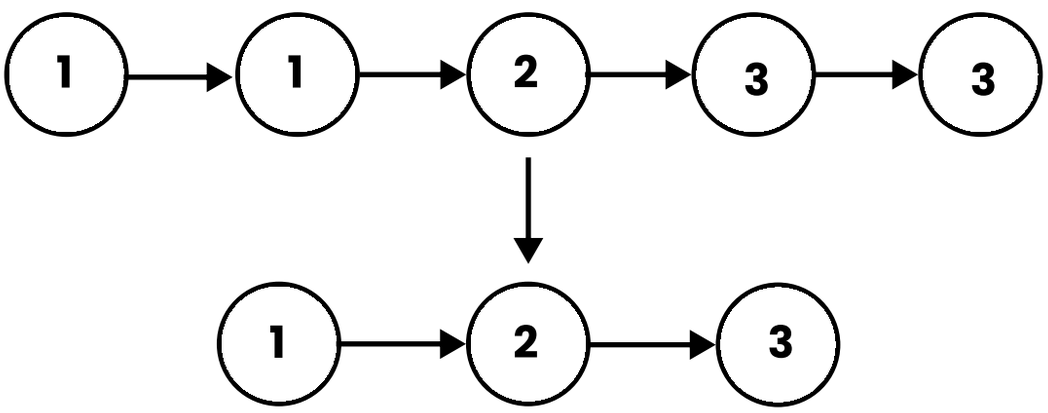
# Remove Duplicates from the Sorted List (Leetcode-83)

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.

**Input**: head = [1,1,2]
**Output**: [1,2]




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



**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head

        slow = head
        fast = head.next

        while fast:
            if slow.val == fast.val:
                slow.next = fast.next
            else:
                slow = slow.next
            fast = fast.next
 return head
```

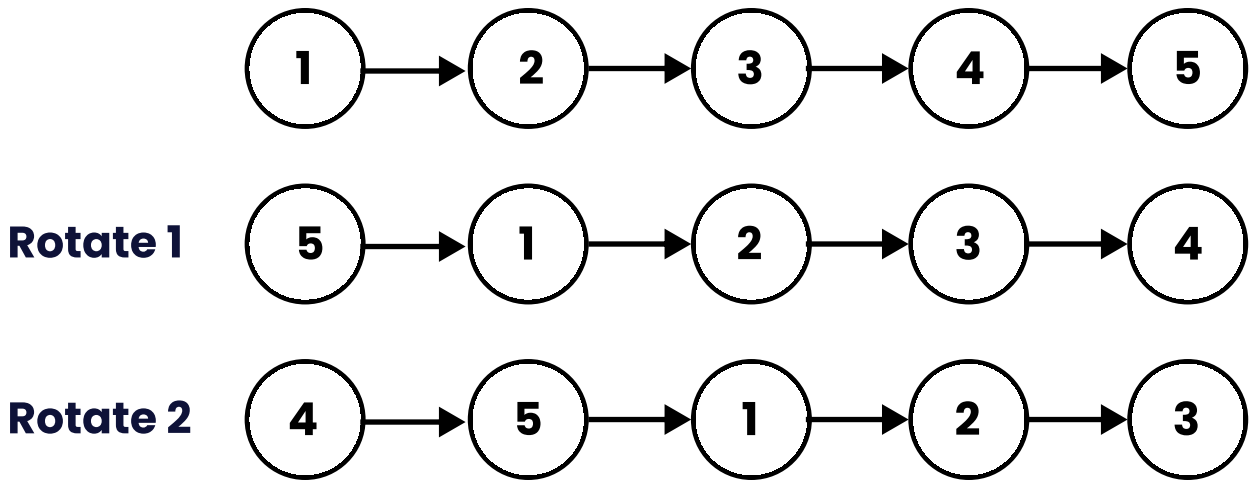
**Time complexity:** O(n) - where n is the number of nodes in the linked list.

**Space complexity:** O(1) - constant space is used, as no additional data structures are employed in the algorithm.

# Rotate List (Leetcode-61)

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

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



**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        if not head:
            return head

        # Compute the length of the list
        length = 1
        tail = head
        while tail.next:
            tail = tail.next
            length += 1

        # Find the new head
        k = k % length
        if k == 0:
            return head

        # Move to the (length - k)th node
        new_tail = head
        for _ in range(length - k - 1):
            new_tail = new_tail.next

        new_head = new_tail.next
        new_tail.next = None
        tail.next = head

        return new_head
```

**Time complexity:** O(n) - where n is the number of nodes in the linked list. The algorithm iterates through the list twice: once to calculate the length and once to find the new head.

**Space complexity:** O(1) - constant space is used, as only a constant number of pointers are used regardless of the size of the input linked list.

# Spiral Matrix IV (Leetcode-2326)

You are given two integers m and n, which represent the dimensions of a matrix.

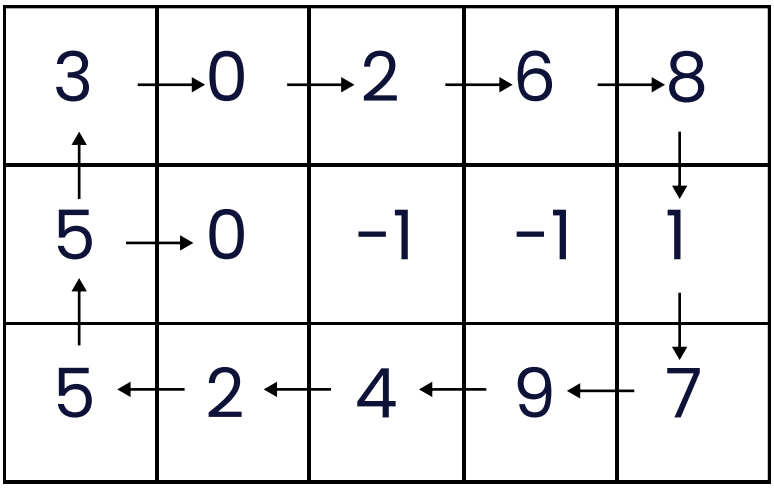
You are also given the head of a linked list of integers.

Generate an m x n matrix that contains the integers in the linked list presented in spiral order (clockwise), starting from the top-left of the matrix. If there are remaining empty spaces, fill them with -1.

Return the generated matrix.

**Input:** m = 3, n = 5, head = [3,0,2,6,8,1,7,9,4,2,5,5,0]
**Output:** [[3,0,2,6,8],[5,0,-1,-1,1],[5,2,4,9,7]]

**Explanation:** The diagram above shows how the values are printed in the matrix.
Note that the remaining spaces in the matrix are filled with -1.




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def spiralMatrix(self, m: int, n: int, head: ListNode) →
list[list[int]]:
        matrix = [[-1] * n for _ in range(m)]
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        row, col, d = 0, 0, 0

        while head:
            matrix[row][col] = head.val
 head = head.next
            next_row, next_col = row + directions[d][0], col +
directions[d][1]

            if not (0 ⩽ next_row < m and 0 ⩽ next_col < n and
matrix[next_row][next_col] == -1):
                d = (d + 1) % 4
                next_row, next_col = row + directions[d][0], col
+ directions[d][1]

            row, col = next_row, next_col

        return matrix
```

**Time complexity:** O(m * n) - where m is the number of rows and n is the number of columns in the resulting matrix. The algorithm iterates through each cell in the matrix to fill it with values from the linked list.

**Space complexity:** O(m * n) - the space used by the result matrix. The space complexity is determined by the size of the output matrix, which is m x n.

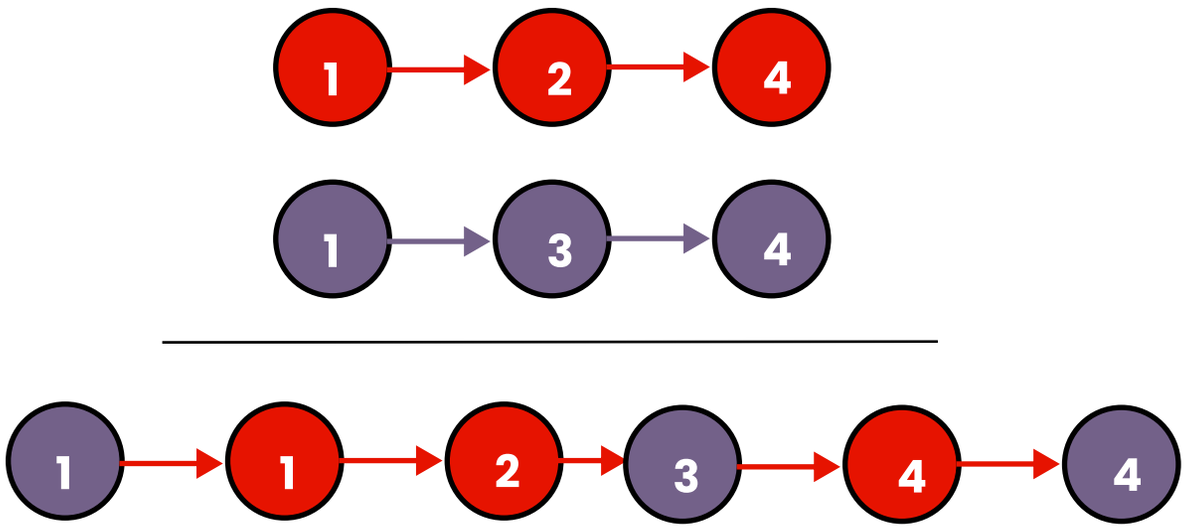
# Merge 2 sorted lists (Leetcode-21)

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.

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




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(self, list1: ListNode, list2: ListNode) →
ListNode:
        if not list1:
            return list2
        if not list2:
            return list1

        if list1.val < list2.val:
            list1.next = self.mergeTwoLists(list1.next, list2)
            return list1
        else:
list2.next = self.mergeTwoLists(list1, list2.next)
            return list2
```

**Time complexity:** O(m + n) where m and n are the lengths of the two linked lists, l1 and l2. The function recursively traverses through each node in the linked lists.

**Space complexity:** O(m + n) due to the recursive stack. In the worst case, the maximum depth of the recursion would be the length of the longer linked list between l1 and l2.

# Merge k sorted lists (Leetcode-23)

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.

**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

**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
self.val = val
        self.next = next

class Solution:
    def mergeKLists(self, lists: list[ListNode]) → ListNode:
        if not lists:
            return None
        return self.mergeKListsHelper(lists, 0, len(lists) - 1)

    def mergeKListsHelper(self, lists, start, end):
        if start == end:
            return lists[start]
        if start + 1 == end:
            return self.merge(lists[start], lists[end])
        mid = (start + end) // 2
        left = self.mergeKListsHelper(lists, start, mid)
        right = self.mergeKListsHelper(lists, mid + 1, end)
        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
```

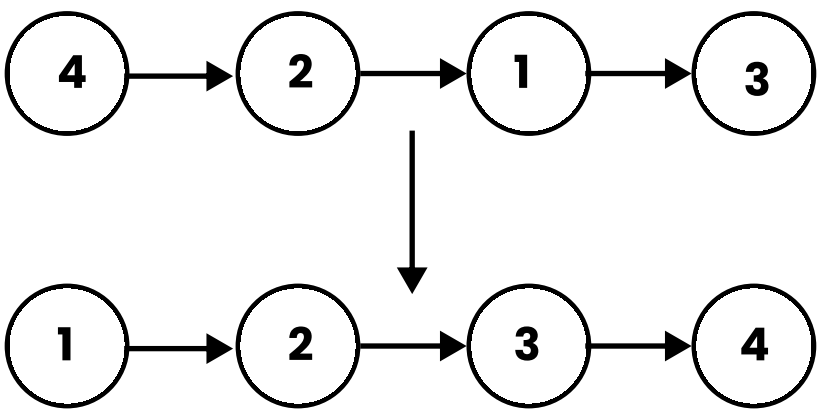
**Time complexity:** O(N log k), where N is the total number of nodes in all linked lists, and k is the number of linked lists. The mergeKLists function uses a divide-and-conquer strategy to merge the lists, and at each level of the recursion, it performs a linear-time merge operation.

**Space complexity:** O(log k), where k is the number of linked lists. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log k.

# Sort List (Leetcode-148)

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

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




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head

        # Find the middle of the list
        slow, fast = head, head
        prev = None
        while fast and fast.next:
            prev = slow
            slow = slow.next
            fast = fast.next.next

 # Split the list into two halves
        prev.next = None

        # Sort each half
        l1 = self.sortList(head)
        l2 = self.sortList(slow)
```

```
# Merge the sorted halves
        return self.merge(l1, l2)

    def merge(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode(0)
 current = dummy

        while l1 and l2:
            if l1.val < l2.val:
                current.next = l1
                l1 = l1.next
            else:
                current.next = l2
                l2 = l2.next
            current = current.next

        current.next = l1 if l1 else l2
        return dummy.next
```

**Time complexity:** O(n log n), where n is the number of nodes in the linked list. This is because the sortList function recursively divides the list into halves, and the merge function performs a linear-time merge operation at each level of the recursion.

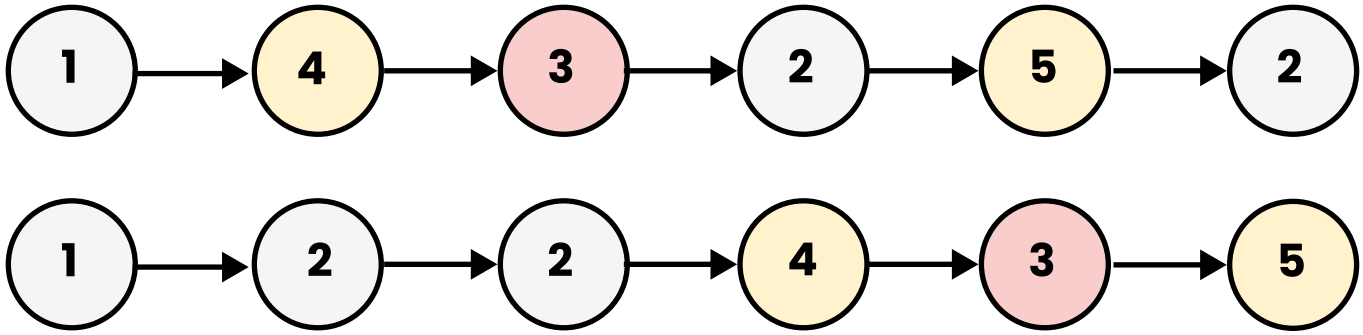
**Space complexity:** O(log n), where n is the number of nodes in the linked list. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log n.

# Partition List (Leetcode-86)

Given the head of a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

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



**Code:**

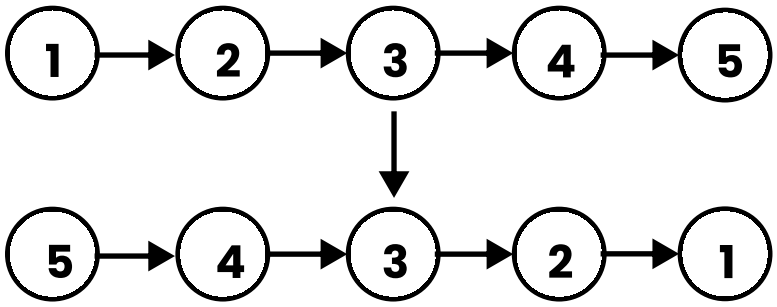
```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def partition(self, head: ListNode, x: int) → ListNode:
        if not head:
            return head

dummyHeadSml = ListNode(0)
        dummyHeadEg = ListNode(0)
        curSmlNode = dummyHeadSml
        curEgNode = dummyHeadEg

        current = head
        while current:
            next_node = current.next
            current.next = None
            if current.val < x:
                curSmlNode.next = current
                curSmlNode = current
            else:
                curEgNode.next = current
                curEgNode = current
            current = next_node
curSmlNode.next = dummyHeadEg.next
        return dummyHeadSml.next
```

**Time complexity:** O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once and performs constant-time operations.

**Space complexity:** O(1). The algorithm uses a constant amount of extra space to store the left and right partitions, regardless of the size of the input linked list.

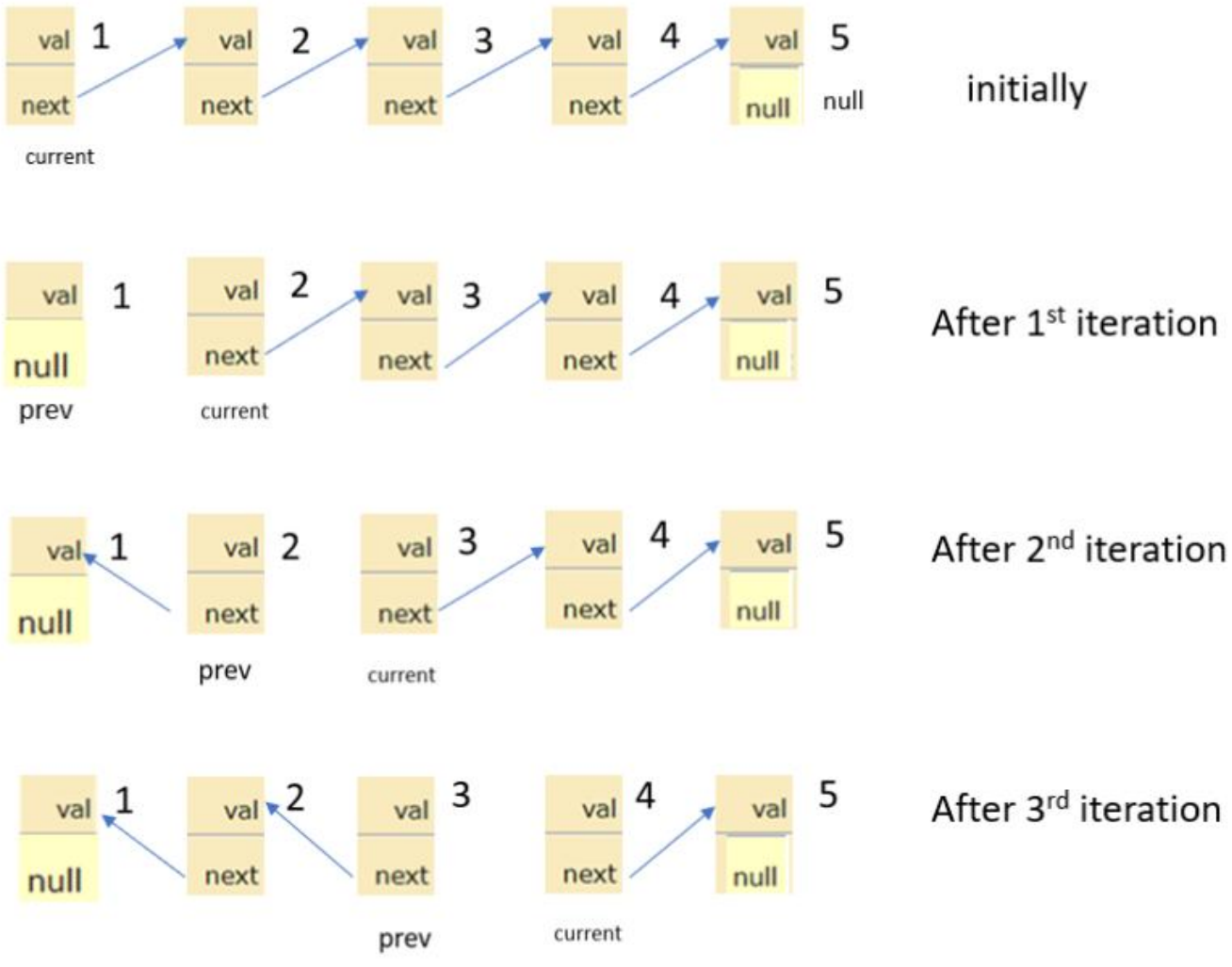
# Reverse Linked List (Leetcode-206)

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



**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev = None
        current = head

        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        return prev
```



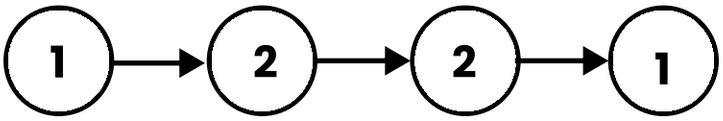
**Time complexity:** O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once, performing constant-time operations in each iteration.

**Space complexity:** O(1). The algorithm uses a constant amount of extra space for the three-pointers (prev, current, next), regardless of the size of the input linked list.

# Palindrome Linked List (Leetcode-234)

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

**Input:** head = [1,2,2,1]
**Output:** true




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
self.val = val
        self.next = next

class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        reverse_head = self.reverse(self.copy(head))

        while head and reverse_head:
            if head.val ≠ reverse_head.val:
                return False
            head = head.next
            reverse_head = reverse_head.next

        return True

    def reverse(self, node: ListNode) -> ListNode:
        prev = None
        current = node

while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        return prev

    def copy(self, node: ListNode) -> ListNode:
        new_head = ListNode(node.val)
        current = new_head
        node = node.next

        while node:
            current.next = ListNode(node.val)
            node = node.next
            current = current.next
 return new_head
```

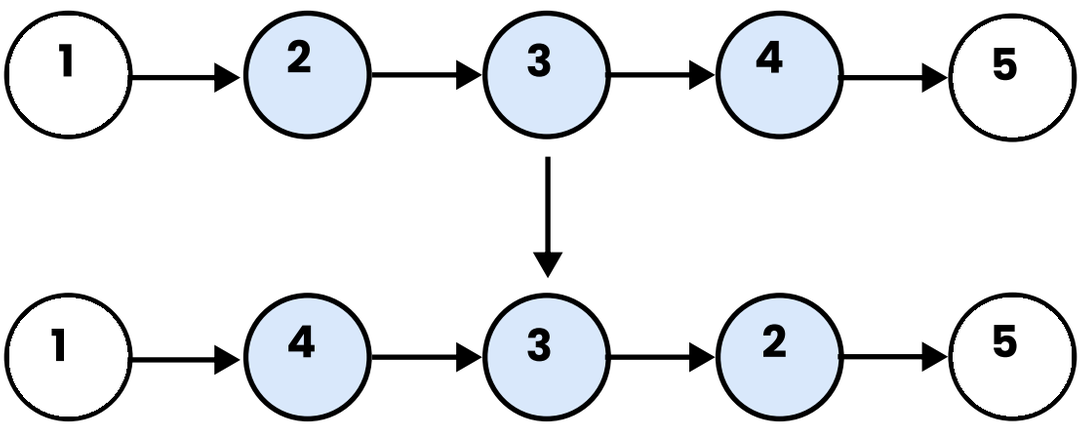
**Time complexity:** O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list twice—once to find the middle and reverse the second half, and once to compare the reversed second half with the first half.

**Space complexity:** O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

# Reverse Linked List II (Leetcode-92)

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.

**Input:** head = [1,2,3,4,5], left = 2, right = 4
**Output**: [1,4,3,2,5]




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseBetween(self, head: ListNode, left: int, right:
int) → ListNode:
        if not head or left == right:
            return head

        dummy = ListNode(0)
        dummy.next = head
        prev = dummy
for _ in range(left - 1):
            prev = prev.next

        current = prev.next
        for _ in range(right - left):
            next_node = current.next
            current.next = next_node.next
            next_node.next = prev.next
            prev.next = next_node
return dummy.next
```

**Time complexity:** O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list once, reversing the specified portion of the list.

**Space complexity:** O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

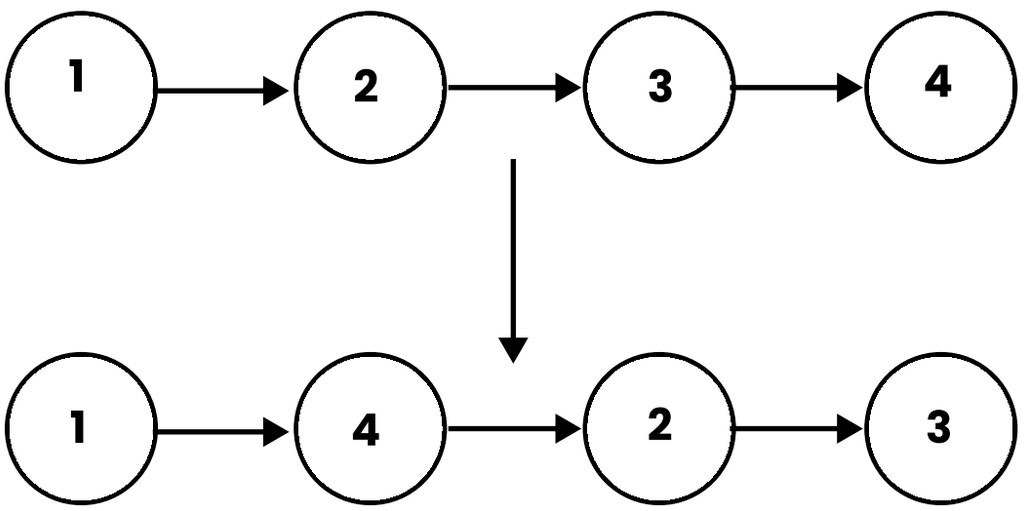
# Reorder List (Leetcode-143)

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

L0 → L1 → … → Ln - 1 → Ln
Reorder the list to be on the following form:

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

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




**Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head:
            return

        # Find the middle of the list

 slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
```

```
# Reverse the second half of the list
        prev, current = None, slow
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        # Merge the two halves
        first, second = head, prev
        while second.next:
            temp1, temp2 = first.next, second.next
            first.next = second
            second.next = temp1
            first, second = temp1, temp2
```