Linked List

1. Merge Two Sorted Lists

Pattern: Linked Lists + Two Pointers

Problem Statement

> 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.

Sample Input & Output

```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
Explanation: Merge by comparing nodes and linking smaller one.
```

```
Input: list1 = [], list2 = []
Output: []
Explanation: Both lists empty → return null.
```

```
Input: list1 = [], list2 = [0]
Output: [0]
Explanation: One list empty → return the other as-is.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(
        self,
        list1: Optional[ListNode],
        list2: Optional[ListNode]
    ) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - Use dummy head to simplify edge cases (empty lists)
        #   - `current` tracks the tail of the merged list
        dummy = ListNode()
        current = dummy

        # STEP 2: Main loop / recursion
        #   - While both lists have nodes, compare values
        #   - Link the smaller node to `current`
        while list1 and list2:
            if list1.val <= list2.val:
                current.next = list1
                list1 = list1.next
            else:
                current.next = list2
                list2 = list2.next
            current = current.next  # move tail forward

        # STEP 3: Update state / bookkeeping
        #   - One list may remain; attach it directly
        #   - No need to iterate-already sorted
        current.next = list1 or list2

        # STEP 4: Return result
        #   - Skip dummy node; return real head
        return dummy.next
```

```python
# -------------------- INLINE TESTS --------------------
def to_list(head: Optional[ListNode]) -> list:
    """Helper: convert linked list to Python list for easy testing"""
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

def from_list(vals: list) -> Optional[ListNode]:
    """Helper: convert Python list to linked list"""
    if not vals:
        return None
    head = ListNode(vals[0])
    current = head
    for val in vals[1:]:
        current.next = ListNode(val)
        current = current.next
    return head

if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    l1 = from_list([1, 2, 4])
    l2 = from_list([1, 3, 4])
    merged = sol.mergeTwoLists(l1, l2)
    print(to_list(merged))  # Expected: [1, 1, 2, 3, 4, 4]

    #  Test 2: Edge case - both empty
    l1 = from_list([])
    l2 = from_list([])
    merged = sol.mergeTwoLists(l1, l2)
    print(to_list(merged))  # Expected: []

    #  Test 3: Tricky/negative - one empty
    l1 = from_list([])
    l2 = from_list([0])
    merged = sol.mergeTwoLists(l1, l2)
    print(to_list(merged))  # Expected: [0]
```

How to use: Copy-paste this block into .py or Quarto cell → run directly →

instant feedback.

Example Walkthrough

We'll trace **Test 1**: `list1 = [1,2,4]`, `list2 = [1,3,4]`.

1. **Initialize**:
   - `dummy = ListNode(0)` → dummy node with val=0
   - `current = dummy` → points to dummy
   - `list1` points to node(1), `list2` points to node(1)
2. **First loop iteration**:
   - Compare `list1.val = 1` vs `list2.val = 1` → equal, pick `list1`
   - `current.next = list1` → dummy → node(1)
   - Move `list1 = list1.next` → now points to node(2)
   - Move `current = current.next` → now at node(1)
   - State: merged so far = [1]
3. **Second iteration**:
   - `list1.val = 2`, `list2.val = 1` → pick `list2`
   - `current.next = list2` → node(1) → node(1)
   - Move `list2 = list2.next` → node(3)
   - Move `current` → now at second node(1)
   - State: [1, 1]
4. **Third iteration**:
   - `list1.val = 2`, `list2.val = 3` → pick `list1`

- Link node(2), advance `list1` to node(4), `current` to node(2)
- State: [1, 1, 2]

5. **Fourth iteration**:
   - `list1.val = 4`, `list2.val = 3` → pick `list2`
   - Link node(3), advance `list2` to node(4), `current` to node(3)
   - State: [1, 1, 2, 3]
6. **Fifth iteration**:
   - `list1.val = 4`, `list2.val = 4` → pick `list1`
   - Link node(4), advance `list1` to `None`, `current` to node(4)
   - State: [1, 1, 2, 3, 4]
7. **Exit loop**:
   - `list1` is `None`, `list2` is node(4)
   - `current.next = list2` → attach remaining [4]
   - Final list: [1, 1, 2, 3, 4, 4]
8. **Return**: `dummy.next` → skips dummy, returns first real node.

Final output: `[1, 1, 2, 3, 4, 4]`

Complexity Analysis

- **Time Complexity**: `O(m + n)`

  We traverse each node in `list1` (length `m`) and `list2` (length `n`) exactly once. Total steps = `m + n`.

- **Space Complexity**: `O(1)`

  Only constant extra space used (`dummy`, `current`). We reuse existing nodes—no new list allocated.

2. Reverse Linked List

Pattern: Linked List Reversal (Iterative)

Problem Statement

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

You are given the head of a singly linked list. Each node contains an integer value and a pointer to the next node. Your task is to reverse the direction of the links so that the last node becomes the first, and return the new head.

Sample Input & Output

```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
Explanation: The links between nodes are reversed.
```

```
Input: head = [1,2]
Output: [2,1]
Explanation: Two-node list is swapped.
```

```
Input: head = []
Output: []
Explanation: Empty list remains empty.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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]:
        # STEP 1: Initialize structures
        #   - prev tracks the new reversed list's head (starts as None)
        #   - curr is the current node being processed (starts at head)
        prev = None
        curr = head

        # STEP 2: Main loop / recursion
        #   - Process each node until curr becomes None
        #   - Invariant: all nodes before curr are already reversed
        while curr:
            # STEP 3: Update state / bookkeeping
            #   - Save next node before breaking the link
            #   - Reverse the current node's pointer to prev
            #   - Move prev and curr forward
            next_temp = curr.next
            curr.next = prev
            prev = curr
            curr = next_temp

        # STEP 4: Return result
        #   - prev is now the new head of the reversed list
        return prev

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    # Helper to convert list to linked list
    def list_to_linked(lst):
        dummy = ListNode()
        curr = dummy
        for val in lst:
```

```python
        curr.next = ListNode(val)
        curr = curr.next
    return dummy.next

# Helper to convert linked list to list
def linked_to_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

#   Test 1: Normal case
head1 = list_to_linked([1, 2, 3, 4, 5])
rev1 = sol.reverseList(head1)
assert linked_to_list(rev1) == [5, 4, 3, 2, 1]
print("  Test 1 passed: [1,2,3,4,5] → [5,4,3,2,1]")

#   Test 2: Edge case - single node
head2 = list_to_linked([1])
rev2 = sol.reverseList(head2)
assert linked_to_list(rev2) == [1]
print("  Test 2 passed: [1] → [1]")

#   Test 3: Tricky/negative - empty list
head3 = list_to_linked([])
rev3 = sol.reverseList(head3)
assert linked_to_list(rev3) == []
print("  Test 3 passed: [] → []")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `reverseList` on input `[1,2,3]`.

Initial state:
- `head` → `1 → 2 → 3 → None`
- `prev = None`, `curr = node(1)`

Step 1: `curr` is not `None`
- `next_temp = curr.next` → `node(2)`
- `curr.next = prev` → `node(1).next = None`
- `prev = curr` → `prev = node(1)`
- `curr = next_temp` → `curr = node(2)`
→ List so far: `1 → None` (reversed part), `2 → 3 → None` (remaining)

Step 2: `curr = node(2)`
- `next_temp = node(3)`
- `node(2).next = prev (node(1))`
- `prev = node(2)`
- `curr = node(3)`
→ Reversed: `2 → 1 → None`

Step 3: `curr = node(3)`
- `next_temp = None`
- `node(3).next = node(2)`
- `prev = node(3)`
- `curr = None`
→ Reversed: `3 → 2 → 1 → None`

Loop ends → return `prev = node(3)` → `[3,2,1]`

Key insight: We never lose the rest of the list because we save `next_temp` **before** reversing the link.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit each node exactly once in a single `while` loop.

- **Space Complexity**: `O(1)`

  Only three pointers (`prev`, `curr`, `next_temp`) are used — constant extra space.

3. Middle of the Linked List

Pattern: Fast & Slow Pointers (Two Pointers)

Problem Statement

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.

Sample Input & Output

```
Input: [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node is 3 (1-based index 3 of 5).
```

```
Input: [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Two middles (3 and 4); return second → 4.
```

```
Input: [1]
Output: [1]
Explanation: Single node is the middle.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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]:
        # STEP 1: Initialize two pointers
        #   - slow moves 1 step at a time
        #   - fast moves 2 steps at a time
        slow = fast = head

        # STEP 2: Main loop - move pointers until fast reaches end
        #   - Invariant: when fast can't move 2 steps, slow is at middle
        #   - For even length: stops at second middle (as required)
        while fast and fast.next:
            slow = slow.next          # Move slow by 1
            fast = fast.next.next     # Move fast by 2

        # STEP 3: No extra bookkeeping needed - pointer logic handles it

        # STEP 4: Return slow - it's at the middle
        return slow

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - odd length
    n1 = ListNode(1)
    n2 = ListNode(2)
    n3 = ListNode(3)
    n4 = ListNode(4)
    n5 = ListNode(5)
    n1.next, n2.next, n3.next, n4.next = n2, n3, n4, n5
    res = sol.middleNode(n1)
    assert [res.val, res.next.val, res.next.next.val] == [3,4,5]

    #  Test 2: Edge case - single node
    single = ListNode(1)
    res = sol.middleNode(single)
    assert res.val == 1 and res.next is None

    #  Test 3: Tricky case - even length (return second middle)
    e1 = ListNode(1)
    e2 = ListNode(2)
    e3 = ListNode(3)
    e4 = ListNode(4)
    e5 = ListNode(5)
    e6 = ListNode(6)
    e1.next, e2.next, e3.next = e2, e3, e4
    e4.next, e5.next = e5, e6
    res = sol.middleNode(e1)
    assert [res.val, res.next.val, res.next.next.val] == [4,5,6]

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `[1,2,3,4,5]`

1. **Initialize**:
   `slow = head` → points to node `1`
   `fast = head` → also points to node `1`
2. **First loop iteration**:
   - Condition: `fast (1)` and `fast.next (2)` exist → enter loop
   - `slow = slow.next` → now points to `2`
   - `fast = fast.next.next` → from `1` → `2` → `3`, so points to `3`
3. **Second loop iteration**:
   - Condition: `fast (3)` and `fast.next (4)` exist → enter loop
   - `slow = slow.next` → from `2` → `3`
   - `fast = fast.next.next` → from `3` → `4` → `5`, so points to `5`
4. **Third loop check**:
   - `fast = 5` → not `None`, but `fast.next` is `None`
   - Loop condition `fast and fast.next` fails → exit
5. **Return**: `slow` points to node `3` → correct middle

For even-length list `[1,2,3,4,5,6]`: - After 1st iter: slow=2, fast=3
- After 2nd iter: slow=3, fast=5
- After 3rd iter: slow=4, fast=None (since 5→6→None)
- Loop stops → return `4` (second middle)

This works because **fast moves twice as fast**, so when it reaches the end, slow has covered exactly half the distance.

Complexity Analysis

- **Time Complexity**: `O(n)`

  The `fast` pointer traverses the list once (n/2 iterations), so total steps n/2 → linear in input size.

- **Space Complexity**: `O(1)`

  Only two pointers (`slow`, `fast`) used — constant extra space, regardless of list length.

4. Remove Nth Node From End of List

Pattern: Two Pointers (Fast & Slow)

Problem Statement

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

Sample Input & Output

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
Explanation: The 2nd node from the end is 4 → remove it.
```

```
Input: head = [1], n = 1
Output: []
Explanation: Only one node → remove it → empty list.
```

```
Input: head = [1,2], n = 2
Output: [2]
Explanation: Remove the 2nd from end (i.e., the first node).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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]:
        # STEP 1: Initialize dummy node and two pointers
        #   - Dummy simplifies edge case (removing head)
        #   - Fast and slow both start at dummy
        dummy = ListNode(0)
        dummy.next = head
        fast = slow = dummy

        # STEP 2: Move fast pointer n+1 steps ahead
        #   - Ensures gap of n+1 between fast and slow
        #   - So when fast reaches end, slow is just before target
        for _ in range(n + 1):
            fast = fast.next

        # STEP 3: Move both pointers until fast hits None
        #   - Maintains fixed gap → slow lands before nth-from-end
        while fast:
            fast = fast.next
            slow = slow.next

        # STEP 4: Remove target node
        #   - slow.next is the node to remove
        #   - Bypass it by linking slow to slow.next.next
        slow.next = slow.next.next

        # Return dummy.next (handles head removal cleanly)
        return dummy.next

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - [1,2,3,4,5], n=2 → [1,2,3,5]
    head1 = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
    res1 = sol.removeNthFromEnd(head1, 2)
    out1 = []
    curr = res1
    while curr:
        out1.append(curr.val)
        curr = curr.next
    print("Test 1:", out1)  # Expected: [1, 2, 3, 5]

    #  Test 2: Edge case - [1], n=1 → []
    head2 = ListNode(1)
    res2 = sol.removeNthFromEnd(head2, 1)
    out2 = []
    curr = res2
    while curr:
        out2.append(curr.val)
        curr = curr.next
    print("Test 2:", out2)  # Expected: []
```

```python
#   Test 3: Tricky case - [1,2], n=2 → [2]
head3 = ListNode(1, ListNode(2))
res3 = sol.removeNthFromEnd(head3, 2)
out3 = []
curr = res3
while curr:
    out3.append(curr.val)
    curr = curr.next
print("Test 3:", out3)  # Expected: [2]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**: `head = [1,2]`, `n = 2`.

1. **Create dummy node**:
   - `dummy = ListNode(0)` → `dummy.next = head` → list: `[0 → 1 → 2]`
   - `fast = slow = dummy` → both point to node `0`.
2. **Move `fast` `n+1 = 3` steps**:
   - Step 1: `fast = fast.next` → points to `1`
   - Step 2: `fast = fast.next` → points to `2`
   - Step 3: `fast = fast.next` → points to `None`
   - Now: `fast = None`, `slow = node 0`
3. **Enter `while fast:` loop**:
   - `fast` is `None` → loop **does not run**
   - So `slow` remains at `node 0`
4. **Remove target node**:
   - `slow.next` is `node 1` (the 2nd from end)

    - `slow.next = slow.next.next` → `node 0` now points to `node 2`
    - Resulting list: `[0 → 2]`
5. **Return `dummy.next`**:
    - `dummy.next` is `node 2` → final list: `[2]`

Output: `[2]` — correct!

Key insight: The **dummy node** lets us treat head removal like any other node, and the **n+1 gap** ensures `slow` stops *just before* the node to delete.

Complexity Analysis

- **Time Complexity**: `O(L)`

    We traverse the list at most twice: once to advance `fast`, once to move both pointers. `L` = length of list.

- **Space Complexity**: `O(1)`

    Only a few pointers (`dummy`, `fast`, `slow`) — no extra space proportional to input.

5. Swap Nodes in Pairs

Pattern: Linked List Manipulation (Iterative Pointer Rewiring)

Problem Statement

> Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed).

The number of nodes in the list is in the range `[0, 100]`.
`0 <= Node.val <= 100`

Sample Input & Output

```
Input: head = [1,2,3,4]
Output: [2,1,4,3]
Explanation: Nodes 1 2 and 3 4 are swapped pairwise.
```

```
Input: head = []
Output: []
Explanation: Empty list remains empty.
```

```
Input: head = [1]
Output: [1]
Explanation: Single node has no pair to swap with.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - Use a dummy node to simplify edge handling at head
        #   - prev points to node before the current pair
        dummy = ListNode(0)
        dummy.next = head
        prev = dummy

        # STEP 2: Main loop / recursion
        #   - Loop while at least two nodes remain to swap
        #   - first = first node in pair, second = second
        while prev.next and prev.next.next:
            first = prev.next
            second = prev.next.next

            # STEP 3: Update state / bookkeeping
            #   - Rewire pointers to swap the pair
            #   - Order matters: avoid losing references
            first.next = second.next
            second.next = first
            prev.next = second

            # Move prev to end of swapped pair for next iteration
            prev = first

        # STEP 4: Return result
        #   - dummy.next is new head (handles empty/single cases)
        return dummy.next

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case [1,2,3,4] → [2,1,4,3]
    n4 = ListNode(4)
    n3 = ListNode(3, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    res = sol.swapPairs(n1)
    out = []
    while res:
        out.append(res.val)
        res = res.next
    print("Test 1:", out)  # Expected: [2, 1, 4, 3]

    #  Test 2: Edge case [] → []
    res = sol.swapPairs(None)
    print("Test 2:", [] if not res else "ERROR")  # Expected: []

    #  Test 3: Tricky/negative [1] → [1]
    single = ListNode(1)
    res = sol.swapPairs(single)
    out = [res.val] if res else []
    print("Test 3:", out)  # Expected: [1]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: input `[1,2,3,4]`.

Initial setup: - `dummy = ListNode(0)` - `dummy.next = n1` → list: `0 → 1 → 2 → 3 → 4` - `prev = dummy` (points to node `0`)

First iteration: - `prev.next = n1`, `prev.next.next = n2` → both exist → enter loop - `first = n1`, `second = n2` - Rewiring: - `first.next = second.next` → `n1.next = n3` - `second.next = first` → `n2.next = n1` - `prev.next = second` → `dummy.next = n2` - Now: `0 → 2 → 1 → 3 → 4` - Update `prev = first` → `prev = n1`

Second iteration: - `prev = n1`, so `prev.next = n3`, `prev.next.next = n4` → enter loop - `first = n3`, `second = n4` - Rewiring: - `n3.next = n4.next` → `n3.next = None` - `n4.next = n3` - `n1.next = n4` - Now: `0 → 2 → 1 → 4 → 3` - Update `prev = n3`

Next check: - `prev.next = None` → loop ends

Return `dummy.next` → node `2`, which starts `2 → 1 → 4 → 3`

Final output: `[2, 1, 4, 3]`

Key insight: **dummy node** avoids special-casing the head swap, and **prev always trails the last swapped node**, enabling clean chaining.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit each node exactly once. Each pair is processed in constant time, and there are `n/2` pairs → linear overall.

- **Space Complexity**: `O(1)`

  Only a fixed number of pointers (`dummy`, `prev`, `first`, `second`) are used — no recursion or extra data structures scaling with input.

6. Rotate List

Pattern: Linked List + Two Pointers + Modular Arithmetic

Problem Statement

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

Clarifications:
- Rotation means moving the last `k` nodes to the front.
- If `k` is larger than the list length, use `k % length` (effective rotation).
- Edge cases: empty list, single node, or `k = 0`.

Sample Input & Output

```
Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]
Explanation: Rotate right twice: [5,1,2,3,4] → [4,5,1,2,3]
```

```
Input: head = [0,1,2], k = 4
Output: [2,0,1]
Explanation: k=4 → effective k = 4 % 3 = 1 → rotate once
```

```
Input: head = [1], k = 0
Output: [1]
Explanation: No rotation needed
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def rotateRight(self, head: Optional[ListNode],
                    k: int) -> Optional[ListNode]:
        # STEP 1: Handle empty or single-node list
        if not head or not head.next:
            return head

        # STEP 2: Compute length and get tail
        #   - Traverse once to find length and tail node
        length = 1
        tail = head
        while tail.next:
            tail = tail.next
            length += 1

        # STEP 3: Normalize k using modulo
        #   - Avoid unnecessary full rotations
        k = k % length
        if k == 0:
            return head  # no effective rotation

        # STEP 4: Find new tail (at position length - k - 1)
        #   - New head will be new_tail.next
        new_tail = head
        for _ in range(length - k - 1):
            new_tail = new_tail.next

        new_head = new_tail.next
        new_tail.next = None  # break the list
        tail.next = head      # connect old tail to old head

        return new_head

# ------------------- INLINE TESTS -------------------
```

```
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    # Build [1,2,3,4,5]
    n5 = ListNode(5)
    n4 = ListNode(4, n5)
    n3 = ListNode(3, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    res = sol.rotateRight(n1, 2)
    # Expected: [4,5,1,2,3]
    out = []
    curr = res
    while curr:
        out.append(curr.val)
        curr = curr.next
    print("Test 1:", out)  # [4, 5, 1, 2, 3]

    #  Test 2: Edge case - k > length
    m2 = ListNode(2)
    m1 = ListNode(0, m2)
    res2 = sol.rotateRight(m1, 4)  # len=2 → k=0 → but wait: len=2? No!
    # Correction: [0,1,2] → len=3
    c2 = ListNode(2)
    c1 = ListNode(1, c2)
    c0 = ListNode(0, c1)
    res2 = sol.rotateRight(c0, 4)
    out2 = []
    curr = res2
    while curr:
        out2.append(curr.val)
        curr = curr.next
    print("Test 2:", out2)  # [2, 0, 1]

    #  Test 3: Tricky/negative - k=0 or single node
    single = ListNode(1)
    res3 = sol.rotateRight(single, 0)
    out3 = []
    curr = res3
    while curr:
        out3.append(curr.val)
        curr = curr.next
    print("Test 3:", out3)  # [1]
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `head = [1,2,3,4,5]`, `k = 2`.

Initial state:
- Linked list: `1 → 2 → 3 → 4 → 5 → None`
- `head` points to node `1`

Step-by-step:

1. **Check empty/single**: `head` exists and has next → proceed.
2. **Compute length & find tail**:
   - Start: `tail = node(1)`, `length = 1`
   - Iter 1: `tail = node(2)`, `length = 2`
   - Iter 2: `tail = node(3)`, `length = 3`
   - Iter 3: `tail = node(4)`, `length = 4`
   - Iter 4: `tail = node(5)`, `length = 5` → stop
   - Now: `tail = node(5)`, `length = 5`
3. **Normalize k**:
   - `k = 2 % 5 = 2` → not zero → continue
4. **Find new_tail**:
   - Need to move `length - k - 1 = 5 - 2 - 1 = 2` steps from head
   - Start: `new_tail = node(1)`

- Step 1: `new_tail = node(2)`
- Step 2: `new_tail = node(3)`
- Now: `new_tail = node(3)`

5. **Break and reconnect**:
   - `new_head = new_tail.next = node(4)`
   - Set `new_tail.next = None` → list becomes: `1→2→3→None`
   - Set `tail.next = head` → `5 → 1`
   - Final list: `4 → 5 → 1 → 2 → 3 → None`
6. **Return `new_head` (node 4)** → output `[4,5,1,2,3]`

Final output matches expectation.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse the list **twice**: once to get length/tail, once to find new tail. Each is O(n), so total O(n).

- **Space Complexity**: `O(1)`

  Only a few pointers (`tail`, `new_tail`, `new_head`) are used. No recursion or extra data structures that scale with input.

7. Linked List Cycle

Pattern: Fast & Slow Pointers (Floyd's Cycle Detection)

Problem Statement

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`.

Sample Input & Output

```
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
```

```
Input: head = [1,2], pos = 0
Output: true
Explanation: The tail connects to the 0th node, forming a cycle.
```

```
Input: head = [1], pos = -1
Output: false
Explanation: No cycle exists; the list has only one node pointing to null.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
```

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        # STEP 1: Initialize structures
        #   - Use two pointers: slow (1x speed) and fast (2x speed)
        #   - If there's a cycle, fast will eventually catch up to slow

        if not head or not head.next:
            return False  # Empty or single node → no cycle possible

        slow = head
        fast = head.next

        # STEP 2: Main loop / recursion
        #   - Move slow by 1 step, fast by 2 steps
        #   - Invariant: if cycle exists, fast and slow will meet
        while fast and fast.next:
            if slow == fast:
                return True  # Cycle detected

            slow = slow.next
            fast = fast.next.next

        # STEP 3: Update state / bookkeeping
        #   - Handled implicitly via pointer movement in loop

        # STEP 4: Return result
        #   - If loop exits, fast reached null → no cycle
        return False

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - cycle exists
    n1 = ListNode(3)
    n2 = ListNode(2)
    n3 = ListNode(0)
    n4 = ListNode(-4)
    n1.next = n2
    n2.next = n3
    n3.next = n4
    n4.next = n2  # cycle: -4 → 2
    assert sol.hasCycle(n1) == True
    print("  Test 1 passed: Cycle detected in [3,2,0,-4]")

    #   Test 2: Edge case - single node, no cycle
    n5 = ListNode(1)
    assert sol.hasCycle(n5) == False
    print("  Test 2 passed: Single node, no cycle")

    #   Test 3: Tricky/negative - two nodes, cycle
    n6 = ListNode(1)
    n7 = ListNode(2)
    n6.next = n7
    n7.next = n6  # cycle: 2 → 1
    assert sol.hasCycle(n6) == True
    print("  Test 3 passed: Two-node cycle detected")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `head = [3 → 2 → 0 → -4   2]`

1. **Initialization**:
   - `head` is not null and has `next` → skip early return.
   - `slow = n1 (3)`, `fast = n1.next = n2 (2)`
2. **First loop iteration**:
   - Check: `slow (3) != fast (2)` → continue
   - Update:
     `slow = slow.next → n2 (2)`
     `fast = fast.next.next → n2.next = n3 → n3.next = n4 → so fast = n4 (-4)`
3. **Second loop iteration**:
   - Check: `slow (2) != fast (-4)` → continue
   - Update:
     `slow = n2.next → n3 (0)`
     `fast = n4.next → n2 → n2.next → n3 → so fast = n3 (0)`

4. **Third loop iteration**:

    - Check: `slow (0) == fast (0)` → **match!**
    - Return `True`

State snapshots: - Start: `slow=3, fast=2` - After 1st move: `slow=2, fast=-4` - After 2nd move: `slow=0, fast=0` → **cycle confirmed**

This works because in a cycle, the fast pointer laps the slow one — like two runners on a circular track.

Complexity Analysis

- **Time Complexity**: `O(n)`

    > In the worst case (no cycle), fast traverses all nodes once → ~2n steps → O(n).
    > With a cycle, detection happens in at most one full loop around the cycle.

- **Space Complexity**: `O(1)`

    > Only two pointers (`slow`, `fast`) used — constant extra space regardless of input size.

8. Palindrome Linked List

Pattern: Linked List + Two Pointers (Fast/Slow) + In-Place Reversal

---

Problem Statement

> Given the `head` of a singly linked list, return `true` if it is a palindrome or `false` otherwise.
>
> A palindrome reads the same forward and backward.
> You must solve the problem in **O(n)** time and **O(1)** space.

Sample Input & Output

```
Input: head = [1,2,2,1]
Output: true
Explanation: The list reads the same forwards and backwards.
```

```
Input: head = [1,2]
Output: false
Explanation: [1,2]   [2,1].
```

```
Input: head = [1]
Output: true
Explanation: Single-element lists are palindromes by definition.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        # STEP 1: Handle empty or single-node list
        if not head or not head.next:
            return True

        # STEP 2: Use fast/slow pointers to find middle
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        # STEP 3: Reverse second half starting at slow
        prev = None
        curr = slow
        while curr:
            next_temp = curr.next
            curr.next = prev
            prev = curr
            curr = next_temp
        # Now 'prev' is head of reversed second half

        # STEP 4: Compare first half and reversed second half
        left = head
        right = prev
        while right:  # Only compare until end of second half
            if left.val != right.val:
                return False
            left = left.next
            right = right.next

        # STEP 5: (Optional) Restore list by reversing again
        # Not required by problem, but good practice in real systems

        return True

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case [1,2,2,1]
    n4 = ListNode(1)
    n3 = ListNode(2, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    assert sol.isPalindrome(n1) == True

    #   Test 2: Edge case [1]
    single = ListNode(1)
    assert sol.isPalindrome(single) == True

    #   Test 3: Tricky/negative [1,2,3]
    n3b = ListNode(3)
    n2b = ListNode(2, n3b)
    n1b = ListNode(1, n2b)
    assert sol.isPalindrome(n1b) == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `[1,2,2,1]`

1. **Initial check**: `head` exists and has more than one node → proceed.
2. **Fast/slow pointers**:
   - Start: `slow = 1`, `fast = 1`
   - Step 1: `slow = 2`, `fast = 2` (fast moves two steps: 1→2→2)
   - Step 2: `slow = 2` (second one), `fast = None` (2→1→None)
     → `slow` now points to start of second half: `[2,1]`
3. **Reverse second half**:
   - Start: `curr = 2`, `prev = None`
   - Iter 1: `next_temp = 1`; `curr.next = None`; `prev = 2`; `curr = 1`
   - Iter 2: `next_temp = None`; `curr.next = 2`; `prev = 1`; `curr = None`
     → Reversed list: `1 → 2 → None`; `prev = 1` (new head)
4. **Compare halves**:
   - `left = 1`, `right = 1` → match
   - `left = 2`, `right = 2` → match
   - `right` becomes `None` → stop

5. **Return**: `True`

Final state: First half `[1,2]`, reversed second half `[1,2]` → identical → palindrome.

Complexity Analysis

- **Time Complexity**: `O(n)`

  One pass to find middle (`n/2` steps), one pass to reverse (`n/2`), one pass to compare (`n/2`). Total `1.5n` → `O(n)`.

- **Space Complexity**: `O(1)`

  Only a few pointers (`slow`, `fast`, `prev`, `curr`, `left`, `right`) used. No recursion or extra arrays. Reversal is in-place.

9. Reorder List

Pattern: Linked List + Two Pointers + Reversal

Problem Statement

You are given the head of a singly linked list. Reorder the list such that it follows this pattern:
`L  → L  → … → L    → L` becomes
`L  → L  → L  → L    → L  → L    → …`

You must do this **in-place** without altering the nodes' values.

Sample Input & Output

```
Input: [1,2,3,4]
Output: [1,4,2,3]
Explanation: First and last alternate: 1→4→2→3.
```

```
Input: [1,2,3,4,5]
Output: [1,5,2,4,3]
Explanation: Middle element (3) ends the sequence.
```

```
Input: [1]
Output: [1]
Explanation: Single node - no reordering needed.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reorderList(self, head: Optional[ListNode]) -> None:
        # STEP 1: Find the middle using slow/fast pointers
        #   - Fast moves 2 steps, slow moves 1 → slow ends at mid
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # STEP 2: Reverse the second half starting from slow
        #   - After reversal, second half starts at prev
        prev = None
        curr = slow
        while curr:
            next_temp = curr.next
            curr.next = prev
            prev = curr
            curr = next_temp

        # STEP 3: Merge first half and reversed second half
        #   - Alternate nodes from each half until second is done
        first = head
        second = prev
        while second.next:
            # Save next pointers
            tmp1 = first.next
            tmp2 = second.next

            # Link first → second
            first.next = second
            second.next = tmp1

            # Move pointers forward
            first = tmp1
            second = tmp2

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case [1,2,3,4] → [1,4,2,3]
    n4 = ListNode(4)
    n3 = ListNode(3, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    sol.reorderList(n1)
    # Traverse and collect values
    res = []
    curr = n1
    while curr:
        res.append(curr.val)
        curr = curr.next
    print("Test 1:", res)  # Expected: [1,4,2,3]
```

```
#   Test 2: Odd length [1,2,3,4,5] → [1,5,2,4,3]
n5 = ListNode(5)
n4 = ListNode(4, n5)
n3 = ListNode(3, n4)
n2 = ListNode(2, n3)
n1 = ListNode(1, n2)
sol.reorderList(n1)
res = []
curr = n1
while curr:
    res.append(curr.val)
    curr = curr.next
print("Test 2:", res)  # Expected: [1,5,2,4,3]

#   Test 3: Edge case [1] → [1]
n1 = ListNode(1)
sol.reorderList(n1)
res = [n1.val]
print("Test 3:", res)  # Expected: [1]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `[1,2,3,4]` → `[1,4,2,3]`.

Step 1: Find the middle

- `slow = fast = head (1)`
- Loop:
  - Iter 1: `fast=1→3`, `slow=1→2`
  - Iter 2: `fast=3→None` (since `3.next=4`, `4.next=None`) → stop
- `slow` now points to node `3` → **middle found**.

Step 2: Reverse second half (`3 → 4`)

- Start: `curr = 3`, `prev = None`
- Iter 1:
  - `next_temp = 4`
  - `3.next = None`
  - `prev = 3`, `curr = 4`
- Iter 2:
  - `next_temp = None`
  - `4.next = 3`
  - `prev = 4`, `curr = None` → stop
- Reversed list: `4 → 3 → None`
- `second = 4`

Step 3: Merge first half (`1→2→3`) and reversed second (`4→3`)

Note: Original `2.next` still points to `3`, but we'll overwrite links.

- Initial: `first = 1`, `second = 4`
- Loop condition: `second.next = 3` (not None) → enter
  - `tmp1 = first.next = 2`
  - `tmp2 = second.next = 3`
  - `first.next = 4` → `1→4`
  - `second.next = tmp1 = 2` → `4→2`
  - Update: `first = 2`, `second = 3`
- Now: `second.next = None` → loop ends

Final chain:
`1 → 4 → 2 → 3 → None`

State after each merge step:
- After 1st link: `1→4`, `4→2` (original `2→3` still exists but will be updated next)
- But since `second.next` becomes `None` after one merge (because reversed list ends at `3`), we stop.
- The `2→3` link remains, which is correct.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse the list 3 times:

  1. Find middle: `n/2` steps
  2. Reverse second half: `n/2` steps
  3. Merge: `n/2` steps
     Total `1.5n` → `O(n)`

- **Space Complexity**: `O(1)`

  Only a few pointers (`slow`, `fast`, `prev`, `curr`, `tmp1`, `tmp2`) used.
  No recursion or extra data structures proportional to input size.

10. Odd Even Linked List

Pattern: Linked List Manipulation (Two Pointers / In-Place Rearrangement)

Problem Statement

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The **first** node is considered **odd**, the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.
You must solve the problem **in O(1) extra space complexity and O(n) time complexity**.

Sample Input & Output

```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
Explanation: Odd-indexed nodes (1st, 3rd, 5th): 1→3→5. Even-indexed (2nd, 4th): 2→4. Concaten
```

```
Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]
Explanation: Odds: 2→3→6→7; Evens: 1→5→4 → combined as required.
```

```
Input: head = [1]
Output: [1]
Explanation: Single node - already satisfies condition.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - If list is empty or has 1-2 nodes, no rearrangement needed.
        if not head or not head.next:
            return head

        # odd_head tracks start of odd list (1st node)
        # even_head tracks start of even list (2nd node)
        odd = head
        even = head.next
        even_head = even  # Save head of even list for later linking

        # STEP 2: Main loop / recursion
        #   - Traverse while both odd and even have valid next pointers
        #   - Maintain invariant: odd is always at an odd-indexed node,
        #     even at an even-indexed node.
        while even and even.next:
            # STEP 3: Update state / bookkeeping
            #   - Link odd to next odd (skip one node)
            odd.next = even.next
            odd = odd.next          # Move odd pointer forward

            #   - Link even to next even (skip one node)
            even.next = odd.next
            even = even.next        # Move even pointer forward

        # STEP 4: Return result
        #   - Connect tail of odd list to head of even list
        odd.next = even_head
        return head

# ------------------- INLINE TESTS -------------------
def list_to_linkedlist(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    curr = head
    for val in arr[1:]:
        curr.next = ListNode(val)
        curr = curr.next
    return head

def linkedlist_to_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    head1 = list_to_linkedlist([1, 2, 3, 4, 5])
    result1 = sol.oddEvenList(head1)
    assert linkedlist_to_list(result1) == [1, 3, 5, 2, 4]
    print("  Test 1 passed")

    #  Test 2: Edge case - single node
    head2 = list_to_linkedlist([1])
    result2 = sol.oddEvenList(head2)
    assert linkedlist_to_list(result2) == [1]
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - two nodes
    head3 = list_to_linkedlist([2, 1])
    result3 = sol.oddEvenList(head3)
    assert linkedlist_to_list(result3) == [2, 1]
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `oddEvenList` with input `[1,2,3,4,5]`.

Initial Setup: - `head` points to node `1` - `odd = head` → node `1` - `even = head.next` → node `2` - `even_head = even` → remembers node 2 (start of evens)

First Loop Iteration (`even=2`, `even.next=3` → truthy): - `odd.next = even.next` → `1.next = 3` → now `1 → 3` - `odd = odd.next` → `odd = 3` - `even.next = odd.next` → `2.next = 3.next = 4` → now `2 → 4` - `even = even.next` → `even = 4`

State after iteration 1: - Odd list: `1 → 3` - Even list: `2 → 4` - `odd = 3`, `even = 4`

Second Loop Iteration (`even=4`, `even.next=5` → truthy): - `odd.next = even.next` → `3.next = 5` → now `3 → 5` - `odd = 5` - `even.next = odd.next` → `4.next = 5.next = None` - `even = None`

Loop ends because `even` is now `None`.

Final Step: - `odd.next = even_head` → `5.next = node 2` - Full list: `1 → 3 → 5 → 2 → 4`

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

Key insight: We never create new nodes — just rewired pointers in place.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse each node exactly once. The while loop runs ~n/2 times, but each step processes two nodes → total O(n).

- **Space Complexity**: `O(1)`

  Only a constant number of pointers (`odd`, `even`, `even_head`) are used. No recursion or auxiliary data structures.

11. Add Two Numbers

Pattern: Linked List Traversal + Elementary Math Simulation

Problem Statement

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Sample Input & Output

```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807 → stored as [7,0,8] (reversed).
```

```
Input: l1 = [0], l2 = [0]
Output: [0]
Explanation: 0 + 0 = 0.
```

```
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]
Explanation: 9999999 + 9999 = 10009998 → reversed: [8,9,9,9,0,0,0,1].
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def addTwoNumbers(
        self, l1: Optional[ListNode], l2: Optional[ListNode]
    ) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - dummy head simplifies list construction
        #   - carry tracks overflow from digit addition
        dummy = ListNode(0)
        curr = dummy
        carry = 0

        # STEP 2: Main loop / recursion
        #   - continue while either list has nodes or carry remains
        #   - invariant: curr always points to last node of result
        while l1 or l2 or carry:
            # STEP 3: Update state / bookkeeping
            #   - extract values (0 if list exhausted)
            #   - compute total and new carry
            val1 = l1.val if l1 else 0
            val2 = l2.val if l2 else 0
            total = val1 + val2 + carry
            carry = total // 10
            digit = total % 10

            # Append new node with current digit
            curr.next = ListNode(digit)
            curr = curr.next

            # Advance input lists if possible
            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next

        # STEP 4: Return result
        #   - skip dummy head; return actual start
        return dummy.next

# ------------------- HELPER FOR TESTING -------------------
def list_to_linkedlist(lst):
    dummy = ListNode(0)
    curr = dummy
    for val in lst:
```

```python
            curr.next = ListNode(val)
            curr = curr.next
        return dummy.next

def linkedlist_to_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    l1 = list_to_linkedlist([2, 4, 3])
    l2 = list_to_linkedlist([5, 6, 4])
    res = sol.addTwoNumbers(l1, l2)
    print(linkedlist_to_list(res))  # Expected: [7, 0, 8]

    #   Test 2: Edge case (both zero)
    l1 = list_to_linkedlist([0])
    l2 = list_to_linkedlist([0])
    res = sol.addTwoNumbers(l1, l2)
    print(linkedlist_to_list(res))  # Expected: [0]

    #   Test 3: Tricky/negative (carry propagates)
    l1 = list_to_linkedlist([9,9,9,9,9,9,9])
    l2 = list_to_linkedlist([9,9,9,9])
    res = sol.addTwoNumbers(l1, l2)
    print(linkedlist_to_list(res))  # Expected: [8,9,9,9,0,0,0,1]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `l1 = [2,4,3]`, `l2 = [5,6,4]`

Initial state:
- `dummy = ListNode(0)`
- `curr = dummy` → points to dummy
- `carry = 0`
- `l1` → 2 → 4 → 3
- `l2` → 5 → 6 → 4

Iteration 1:
- `val1 = 2`, `val2 = 5`
- `total = 2 + 5 + 0 = 7`
- `carry = 7 // 10 = 0`
- `digit = 7 % 10 = 7`
- Create `ListNode(7)`, attach to `curr.next`
- `curr` moves to node `7`
- Advance `l1` → 4, `l2` → 6
- **Result so far**: dummy → 7

Iteration 2:
- `val1 = 4`, `val2 = 6`
- `total = 4 + 6 + 0 = 10`
- `carry = 10 // 10 = 1`
- `digit = 10 % 10 = 0`
- Create `ListNode(0)`, attach
- `curr` moves to node `0`
- Advance `l1` → 3, `l2` → 4
- **Result so far**: dummy → 7 → 0

Iteration 3:
- `val1 = 3`, `val2 = 4`
- `total = 3 + 4 + 1 = 8`
- `carry = 8 // 10 = 0`
- `digit = 8`
- Create `ListNode(8)`, attach
- `curr` moves to node `8`
- Advance `l1 = None`, `l2 = None`
- **Result so far**: dummy → 7 → 0 → 8

Loop condition check:
- `l1 = None`, `l2 = None`, `carry = 0` → exit loop

Return: `dummy.next` → head of `[7,0,8]`

Final output: `[7, 0, 8]`

Key insight:
We simulate elementary addition **digit by digit**, propagating carry just like manual math—no need to reverse lists or convert to integers!

Complexity Analysis

- **Time Complexity**: `O(max(m, n))`

  We traverse both lists once, where `m` and `n` are their lengths. Each node is visited at most once.

- **Space Complexity**: `O(max(m, n))`

  The output list's length is at most `max(m, n) + 1` (due to final carry). No extra space beyond result.

12. Sort List

Pattern: Divide and Conquer (Merge Sort on Linked List)

Problem Statement

> Given the `head` of a linked list, return *the list after sorting it in **ascending order***. You must solve it in `O(n log n)` time complexity and `O(1)` space complexity (i.e., iterative or bottom-up merge sort preferred, but top-down with recursion is acceptable if space is `O(log n)` due to call stack).

Sample Input & Output

```
Input: head = [4,2,1,3]
Output: [1,2,3,4]
Explanation: The list is sorted in ascending order.
```

```
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
Explanation: Handles negative numbers and zero correctly.
```

```
Input: head = []
Output: []
Explanation: Empty list remains empty.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # STEP 1: Base case - empty or single-node list
        #   - Already sorted; return as-is.
        if not head or not head.next:
            return head

        # STEP 2: Split list into two halves using slow/fast pointers
        #   - Fast moves 2 steps, slow moves 1 → slow ends at mid.
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Break the list into left and right halves
        mid = slow.next
        slow.next = None  # Terminate left half

        # STEP 3: Recursively sort both halves
        #   - Divide until base case reached.
        left = self.sortList(head)
        right = self.sortList(mid)

        # STEP 4: Merge two sorted halves
        #   - Classic merge of two sorted linked lists.
        dummy = ListNode(0)
        tail = dummy

        while left and right:
            if left.val < right.val:
                tail.next = left
                left = left.next
            else:
                tail.next = right
                right = right.next
            tail = tail.next

        # Attach remaining nodes (one side is always exhausted)
        tail.next = left or right

        return dummy.next

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    # Build [4,2,1,3]
    n4 = ListNode(3)
    n3 = ListNode(1, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(4, n2)
    res = sol.sortList(n1)
    out = []
    while res:
        out.append(res.val)
        res = res.next
    print("Test 1:", out == [1,2,3,4])

    #  Test 2: Edge case - empty list
    print("Test 2:", sol.sortList(None) is None)
```

```
#   Test 3: Tricky/negative - unsorted with negatives
# Build [-1,5,3,4,0]
n5 = ListNode(0)
n4 = ListNode(4, n5)
n3 = ListNode(3, n4)
n2 = ListNode(5, n3)
n1 = ListNode(-1, n2)
res = sol.sortList(n1)
out = []
while res:
    out.append(res.val)
    res = res.next
print("Test 3:", out == [-1,0,3,4,5])
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: input `[4,2,1,3]`.

1. **Initial Call**: `sortList([4→2→1→3])`
   - Not base case → find midpoint.
   - `slow=4`, `fast=2` → then `slow=2`, `fast=1` → then `fast.next` is 3, so loop ends.
   - `mid = slow.next = 1`, and `slow.next = None` → splits into `[4→2]` and `[1→3]`.
2. **Recursive Left**: `sortList([4→2])`
   - Split: `slow=4`, `fast=2` → `fast.next` is `None`, so stop.
   - `mid = 2`, left = `[4]`, right = `[2]`.
   - Both base cases → merge: `2→4`.
3. **Recursive Right**: `sortList([1→3])`
   - Split into `[1]` and `[3]` → merge → `1→3`.
4. **Merge Left (`2→4`) and Right (`1→3`)**

- Compare `2` vs `1` → pick `1`
- Compare `2` vs `3` → pick `2`
- Compare `4` vs `3` → pick `3`
- Append remaining `4`
- Result: `1→2→3→4`

Final output: `[1,2,3,4]` → matches expected.

Complexity Analysis

- **Time Complexity**: `O(n log n)`

  The list is divided in half `log n` times (like merge sort). Each merge step processes all `n` nodes once → total `O(n log n)`.

- **Space Complexity**: `O(log n)`

  Due to recursion depth (call stack). Each recursive call uses constant extra space, but there are `log n` active calls during divide phase.
  Note: LeetCode accepts this as it's better than `O(n)` and meets typical expectations for linked list merge sort.

13. Reverse Nodes in k-Group

Pattern: Linked List Manipulation + Recursion / Iteration with Grouping

---

Problem Statement

Given the `head` of a linked list, reverse the nodes of the list `k` at a time, and return the modified list.

`k` is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of `k`, then left-out nodes, in the end, should remain as they are.

You may not alter the values in the list's nodes — only nodes themselves may be changed.

Sample Input & Output

```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
Explanation: Reverse first 2 → [2,1]; next 2 → [4,3]; last node 5 remains.
```

```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
Explanation: Reverse first 3 → [3,2,1]; remaining 2 nodes < k → unchanged.
```

```
Input: head = [1], k = 1
Output: [1]
Explanation: Single node, k=1 → reverse group of 1 (no change).
```

LeetCode Editorial Solution + Inline Tests

```
from typing import Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseKGroup(
        self, head: Optional[ListNode], k: int
    ) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - Use dummy node to simplify head handling
        #   - 'group_prev' tracks node before current group
        dummy = ListNode(0)
        dummy.next = head
        group_prev = dummy

        while True:
            # STEP 2: Check if k nodes exist ahead
            #   - 'kth' will point to kth node in current group
            kth = self._get_kth_node(group_prev, k)
            if not kth:
                break  # Not enough nodes to reverse → stop

            # Save pointers around the group
            group_next = kth.next
            prev = group_next
            curr = group_prev.next

            # STEP 3: Reverse the k-node segment
            #   - Standard reversal, but stop after k nodes
            while curr != group_next:
                tmp = curr.next
                curr.next = prev
                prev = curr
                curr = tmp

            # STEP 4: Reconnect reversed group
            #   - Update group_prev.next to new head (prev)
            #   - Move group_prev to end of reversed group
            tmp = group_prev.next
            group_prev.next = kth
            group_prev = tmp

        return dummy.next
    def _get_kth_node(
        self, node: ListNode, k: int
    ) -> Optional[ListNode]:
        # Helper: move k steps from 'node'
        while node and k > 0:
            node = node.next
            k -= 1
        return node

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    # Build [1,2,3,4,5]
    n5 = ListNode(5)
    n4 = ListNode(4, n5)
    n3 = ListNode(3, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    res = sol.reverseKGroup(n1, 2)
    out = []
    while res:
        out.append(res.val)
        res = res.next
    print("Test 1:", out)  # Expected: [2,1,4,3,5]

    #   Test 2: Edge case (k = 3)
    n5 = ListNode(5)
    n4 = ListNode(4, n5)
    n3 = ListNode(3, n4)
    n2 = ListNode(2, n3)
    n1 = ListNode(1, n2)
    res = sol.reverseKGroup(n1, 3)
    out = []
    while res:
        out.append(res.val)
        res = res.next
    print("Test 2:", out)  # Expected: [3,2,1,4,5]

    #   Test 3: Tricky/negative (k = 1)
    n1 = ListNode(1)
    res = sol.reverseKGroup(n1, 1)
    out = []
    while res:
        out.append(res.val)
        res = res.next
    print("Test 3:", out)  # Expected: [1]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `head = [1,2,3,4,5]`, `k = 2`.

Initial Setup

- Dummy node created: `dummy → 1 → 2 → 3 → 4 → 5`
- `group_prev = dummy` (points to node before group)

First Group (nodes 1,2)

1. **Find kth node**: `_get_kth_node(dummy, 2)` → returns node `2`.
2. `group_next = 2.next = 3`
3. Start reversal:

   - `prev = 3`, `curr = 1`
   - Loop:

     - `curr=1`: `1.next = 3` → `prev=1`, `curr=2`
     - `curr=2`: `2.next = 1` → `prev=2`, `curr=3` → stop (`curr == group_next`)

4. Reconnect:

   - `group_prev.next` (dummy.next) = `2` → now `dummy → 2 → 1 → 3...`
   - `group_prev = old head = 1` (now tail of reversed group)

Second Group (nodes 3,4)

1. `_get_kth_node(1, 2)` → returns node `4`
2. `group_next = 5`
3. Reverse:
    - `prev = 5`, `curr = 3`
    - `3.next = 5`, `prev=3`, `curr=4`
    - `4.next = 3`, `prev=4`, `curr=5` → stop
4. Reconnect:
    - `1.next = 4` → list: `2→1→4→3→5`
    - `group_prev = 3`

Third Group (node 5)

1. `_get_kth_node(3, 2)` → tries to move 2 steps: `3→5→None` → returns `None`
2. Break loop → return `dummy.next = 2`

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

Complexity Analysis

- **Time Complexity**: `O(n)`

    Each node is visited a constant number of times: once to check group size, once to reverse, and once during reconnection. Total ~3n → O(n).

- **Space Complexity**: `O(1)`

    Only a few pointers (`dummy`, `group_prev`, `kth`, `prev`, `curr`, etc.) are used. No recursion stack or extra data structures proportional to input size.

14. LRU Cache

Pattern: Hash Map + Doubly Linked List (Ordered Data Structure)

Problem Statement

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the `LRUCache` class: - `LRUCache(int capacity)` Initialize the LRU cache with **positive** size capacity. - `int get(int key)` Return the value of the key if it exists, otherwise return `-1`. - `void put(int key, int value)` Update the value of the key if it exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds `capacity`, evict the least recently used key.

Follow up: Could you do this in `O(1)` time complexity for both `get` and `put`?

Sample Input & Output

```
Input: ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
       [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output: [null, null, null, 1, null, -1, null, -1, 3, 4]
Explanation:
- Cache initialized with capacity 2.
- put(1,1) → cache: {1:1}
- put(2,2) → cache: {1:1, 2:2}
- get(1) → returns 1; now 1 is most recently used → order: [2, 1]
- put(3,3) → evicts 2 (least recent); cache: {1:1, 3:3}
- get(2) → not found → -1
- put(4,4) → evicts 1; cache: {3:3, 4:4}
- get(1) → -1
- get(3) → 3
- get(4) → 4
```

Edge Case:

```
Input: ["LRUCache", "get", "put", "get"]
       [[1], [1], [1, 2], [1]]
Output: [null, -1, null, 2]
```

Tricky Case (Update existing key):

```
Input: ["LRUCache", "put", "put", "put", "get"]
       [[2], [1,1], [2,2], [1,3], [1]]
Output: [null, null, null, null, 3]
Explanation: Updating key 1 should refresh its recency.
```

LeetCode Editorial Solution + Inline Tests

```python
class Node:
    def __init__(self, key: int, val: int):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        # STEP 1: Initialize structures
        #   - Use dummy head/tail to simplify edge logic
        #   - Hash map for O(1) key lookup
        #   - DLL maintains recency order (head = MRU, tail = LRU)
        self.capacity = capacity
        self.cache = {}
        self.head = Node(0, 0)  # dummy head
        self.tail = Node(0, 0)  # dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node: Node) -> None:
        # Remove node from DLL
        prev_node = node.prev
        next_node = node.next
        prev_node.next = next_node
        next_node.prev = prev_node

    def _add_to_head(self, node: Node) -> None:
        # Add node right after head (MRU position)
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        # STEP 2: Main logic for get
        #   - If key exists, move node to MRU and return val
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add_to_head(node)
            return node.val
        return -1

    def put(self, key: int, value: int) -> None:
        # STEP 3: Main logic for put
        #   - If key exists: update and move to MRU
        #   - Else: add new node; evict LRU if over capacity
        if key in self.cache:
            node = self.cache[key]
            node.val = value
            self._remove(node)
            self._add_to_head(node)
        else:
            new_node = Node(key, value)
            self.cache[key] = new_node
            self._add_to_head(new_node)
            if len(self.cache) > self.capacity:
                # Evict LRU (node before dummy tail)
                lru_node = self.tail.prev
                self._remove(lru_node)
                del self.cache[lru_node.key]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = None

    #  Test 1: Normal case
    sol = LRUCache(2)
    sol.put(1, 1)
    sol.put(2, 2)
    assert sol.get(1) == 1
    sol.put(3, 3)
    assert sol.get(2) == -1
    sol.put(4, 4)
    assert sol.get(1) == -1
    assert sol.get(3) == 3
    assert sol.get(4) == 4
    print("  Test 1 passed")

    #   Test 2: Edge case (capacity = 1)
    sol = LRUCache(1)
    assert sol.get(1) == -1
    sol.put(1, 2)
    assert sol.get(1) == 2
    print("  Test 2 passed")

    #   Test 3: Tricky case (update existing key)
    sol = LRUCache(2)
    sol.put(1, 1)
    sol.put(2, 2)
    sol.put(1, 3)  # update key 1
    assert sol.get(1) == 3
    assert len(sol.cache) == 2
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3** step by step:

1. **`sol = LRUCache(2)`**
   - Creates cache with capacity 2.
   - `head`   `tail` (dummy nodes).
   - `cache = {}`.
2. **`sol.put(1, 1)`**

    - Key `1` not in cache → create `Node(1,1)`.
    - Add to head: `head   [1]   tail`.
    - `cache = {1: node1}`.

3. **`sol.put(2, 2)`**
    - Key `2` not in cache → create `Node(2,2)`.
    - Add to head: `head   [2]   [1]   tail`.
    - `cache = {1: node1, 2: node2}`.
4. **`sol.put(1, 3)`**
    - Key `1` **exists** → update `node1.val = 3`.
    - Remove **`node1`** from current position:
      `head   [2]   tail` (temporarily).
    - Add **`node1`** to head: `head   [1]   [2]   tail`.
    - Now `1` is MRU, `2` is LRU.
    - `cache` still has 2 keys.
5. **`sol.get(1)`**
    - Found in cache → return `3`.
    - Already at head → no structural change.

Final state:
- DLL: `head   [1]   [2]   tail`
- `cache = {1: Node(1,3), 2: Node(2,2)}`
- Output: `3`

Key takeaway: **Updating or accessing a key moves it to front**, ensuring LRU eviction works correctly.

Complexity Analysis

- **Time Complexity**: `O(1)`

  Both `get` and `put` use hash map lookup (`O(1)`) and doubly linked list insert/remove (`O(1)` due to direct node access via pointers).

- **Space Complexity**: `O(capacity)`

  The cache stores at most `capacity` key-node pairs. The DLL and hash map scale linearly with the number of stored keys.