Sort List

Option A: Bottom-up merge sort (O(1) extra space)

Idea:

- Repeatedly merge runs (sublists) of size 1, 2, 4, 8, ... until the whole list is one run.
- You'll need three tiny helpers:
  - length(head) → number of nodes
  - split(head, k) → cut off the first k nodes and return the head of the remainder (or nullptr).
  - merge(l1, l2) → merge two sorted lists; return (mergeHead, mergedTail).

Outline:

```
1. Compute n = length(head). Use a dummy prehead (0, head) to simplify links.
2. For step = 1; step < n; step <<= 1:
     - prev = &dummy, cur = dummy.next
     - while cur:
         - left = cur
         - right = split(left, step) // detaches left run of size ≤ step
         - next = split(right, step) // detaches right run of size ≤ step
         - (mergedHead, mergedTail) = merge(left, right)
         - prev → next = mergedHead
         - prev = mergedTail
         - cur = next
3. Return dummy.next.
```

Option B: Top-down (recursive) merge sort (simpler but O(log n) stack)

Idea:

- Use slow/fast pointers to find the middle, split into two halves, recursively sort, then merge.

Outline:

1. Base: if head == nullptr || head → next == nullptr return head.
2. Find mid with slow/fast, split into [head... mid] and [midNext.. end].
3. L = sortList (left), R = sortList (right)
4. Return merge (L, R).

Helper behavior (for either option):

merge(l1, l2):

- Classic two-pointer merge for sorted list.
- Keep a dummy node, append the smaller node each time.
- Return (dummy.next, tailPointer).

split(head k) (bottom-up only):

- If k==0 return head.
- Walk k-1 steps or until end; let tail be last node of first part.
- rest = tail ? tail → next : nullptr
- If tail exists, set tail → next = nullptr
- Return reset.

Complexity:

- Time: O(n logn)
- Space: O(1) extra for bottom-up; O(logn) stack for recursive.