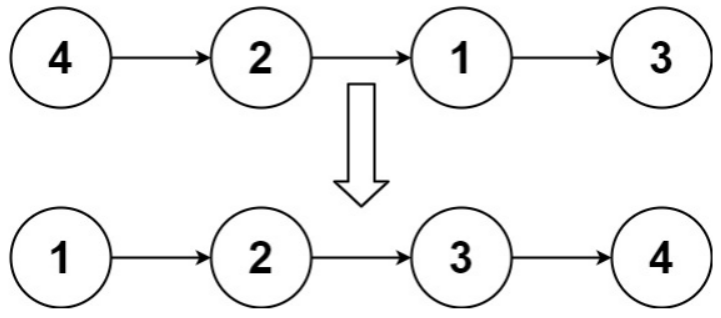
<https://leetcode.com/problems/sort-list/>

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

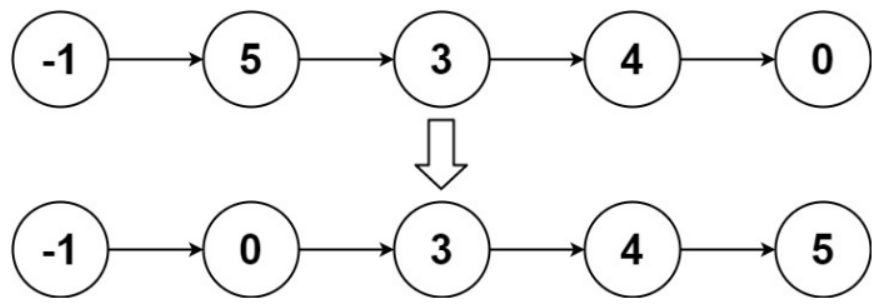
**Example 1:**



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 \* 104].
* -105 <= Node.val <= 105

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

**Attempt 1: 2023-02-10**

**Solution 1: Merge Sort (60 min)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode sortList(ListNode head) {

// Split list into two halves

if(head == null || head.next == null) {

return head;

}

ListNode slow = head;

ListNode fast = head;

while(fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

}

ListNode firstHalf = head;

ListNode secondHalf = slow.next;

slow.next = null;

// Merge sort

ListNode l1 = sortList(firstHalf);

ListNode l2 = sortList(secondHalf);

return merge(l1, l2);

}

private ListNode merge(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode();

ListNode iter = dummy;

while(l1 != null || l2 != null) {

if(l1 != null && l2 != null) {

if(l1.val < l2.val) {

iter.next = l1;

l1 = l1.next;

} else {

iter.next = l2;

l2 = l2.next;

}

} else if(l1 == null) {

iter.next = l2;

l2 = l2.next;

} else {

iter.next = l1;

l1 = l1.next;

}

iter = iter.next;

}

return dummy.next;

}

}

Time Complexity: O(nlog⁡n), where n is the number of nodes in linked list. The algorithm can be split into 2 phases, Split and Merge

Space Complexity: O(log⁡n), where n is the number of nodes in linked list. Since the problem is recursive, we need additional space to store the recursive call stack. The maximum depth of the recursion tree is log⁡n

**Note: There is a slightly difference between L148. Sort List vs L876. Middle of the Linked List**

**In L876 use below 'slow' and 'fast' pointer (fast != null && [fast.next](http://fast.next) != null) to find the actual middle node, 'slow' directly point to it**

**Test out by 2 cases:**

**1. {1,2,3,4,5} -> slow = 3, actual middle node = 3**

**2. {1,2,3,4,5,6} -> slow = 4, actual middle node = 4**

while(fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

}

return slow;

**But in L148 use another style ([fast.next](http://fast.next) != null && [fast.next.next](http://fast.next.next) != null) because the purpose is cut the list into two halves, the cutting point is after the end of first half and before the start of second half, instead of find actual middle node, 'slow' should point to last node of first half, in future we can cut list into two halves by reserve head of second half as 'ListNode secondHalf = slow.next' and cut list by removing connection between two sub-lists as 'slow.next=null'**

**Test out by 2 cases:**

**1.{1,2,3,4,5} -> slow = 2, [slow.next](http://slow.next) = 3, second half start node = 3**

**2.{1,2,3,4,5,6} -> slow = 3, [slow.next](http://slow.next) = 4, second half start node = 4**

while(fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

}

ListNode firstHalf = head;

ListNode secondHalf = slow.next;

slow.next = null;

**Refer to**

<https://leetcode.com/problems/sort-list/solutions/840381/sort-list/>

#### **Overview**

The problem is to sort the linked list in O(nlog⁡n) time and using only constant extra space. If we look at various sorting algorithms, [Merge Sort](https://en.wikipedia.org/wiki/Merge_sort) is one of the efficient sorting algorithms that is popularly used for sorting the linked list. The merge sort algorithm runs in O(nlog⁡n) time in all the cases. Let's discuss approaches to sort linked list using merge sort.

[Quicksort](https://en.wikipedia.org/wiki/Quicksort) is also one of the efficient algorithms with the average time complexity of O(nlog⁡n). But the worst-case time complexity is O(n^2). Also, variations of the quick sort like randomized quicksort are not efficient for the linked list because unlike arrays, random access in the linked list is not possible in O(1) time. If we sort the linked list using quicksort, we would end up using the head as a pivot element which may not be efficient in all scenarios.

#### **Approach 1: Top Down Merge Sort**

**Intuition**

Merge sort is a popularly known algorithm that follows the [Divide and Conquer Strategy](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm). The divide and conquer strategy can be split into 2 phases:

*Divide phase*: Divide the problem into subproblems.

*Conquer phase*: Repeatedly solve each subproblem independently and combine the result to form the original problem.

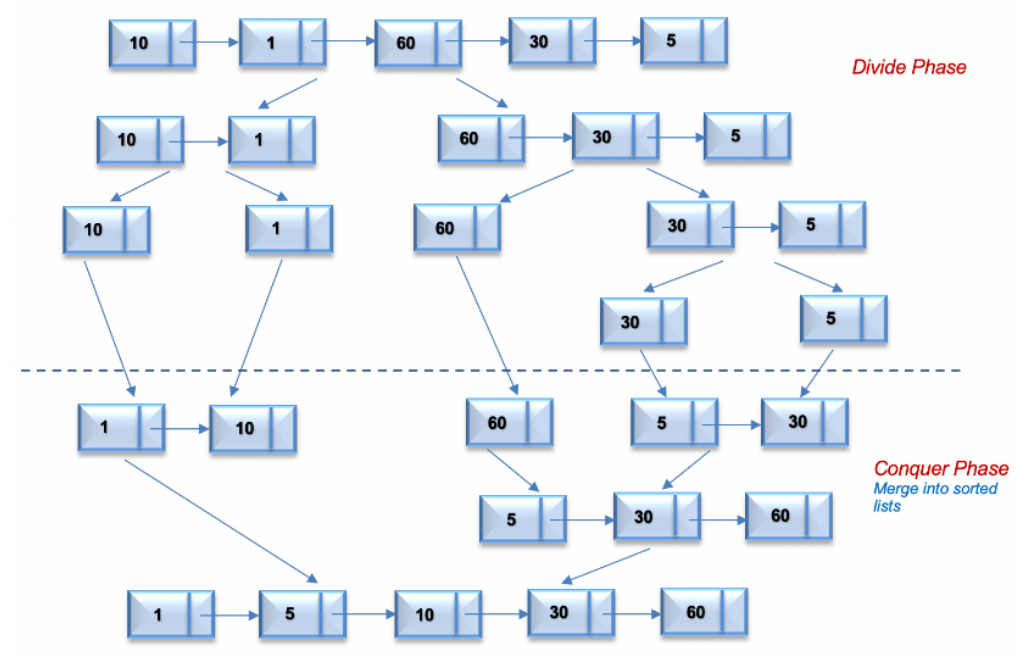
The Top Down approach for merge sort recursively splits the original list into sublists of equal sizes, sorts each sublist independently, and eventually merge the sorted lists. Let's look at the algorithm to implement merge sort in Top Down Fashion.

**Algorithm**

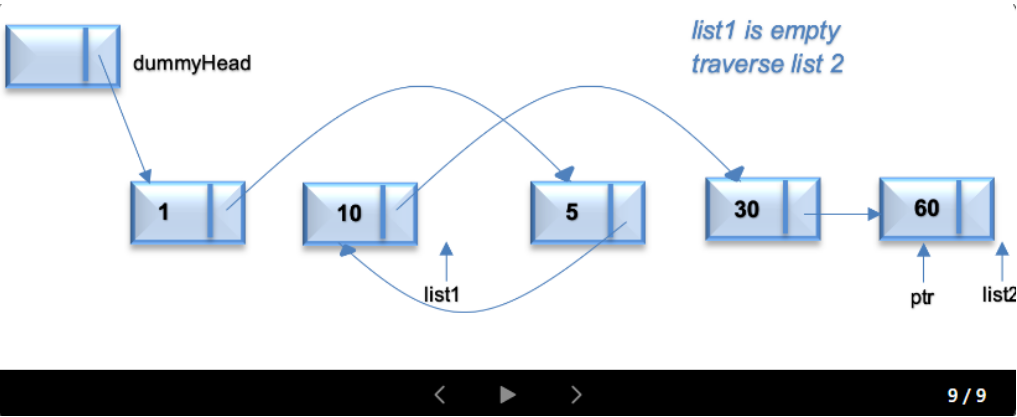
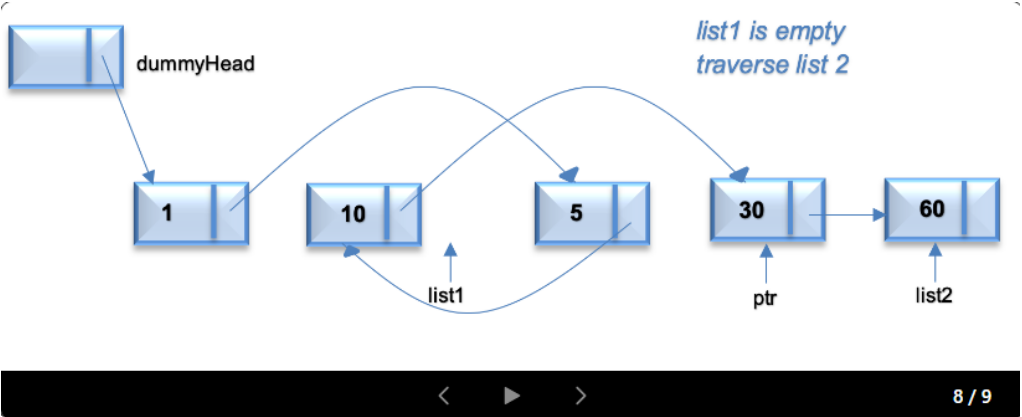
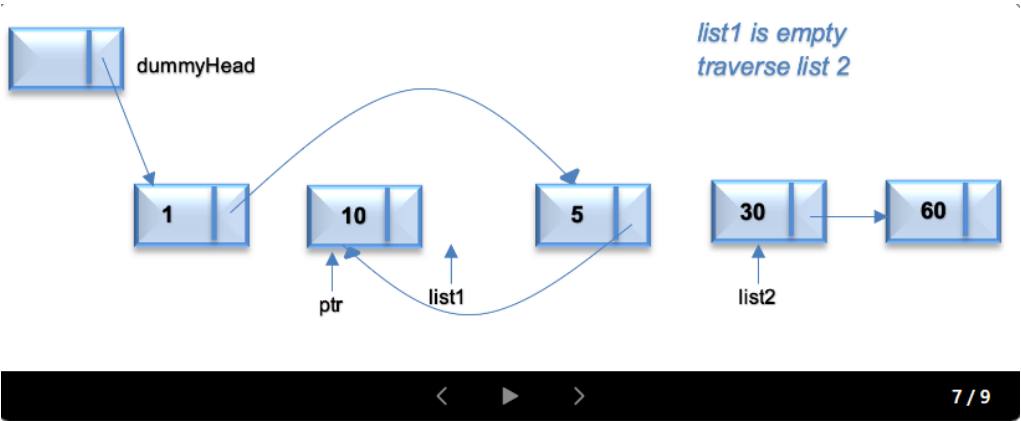
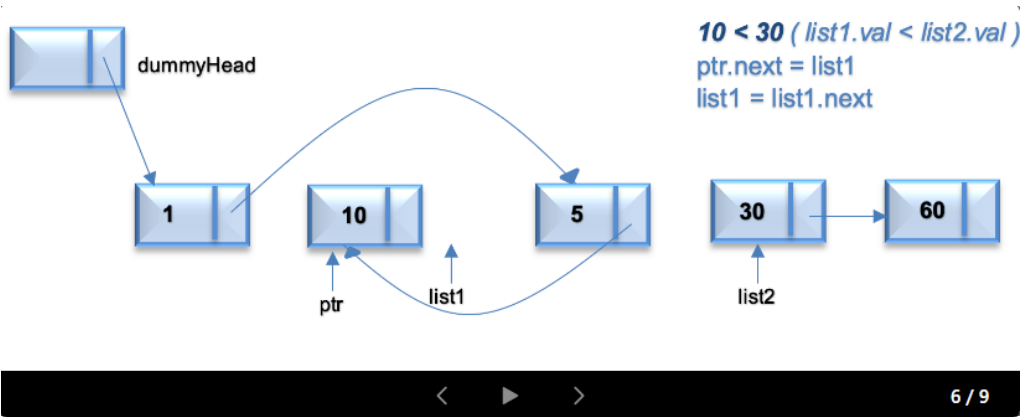
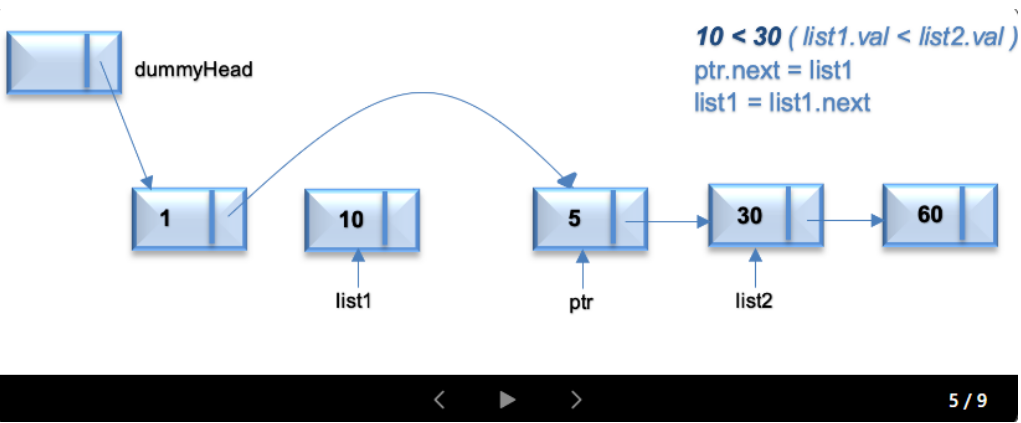
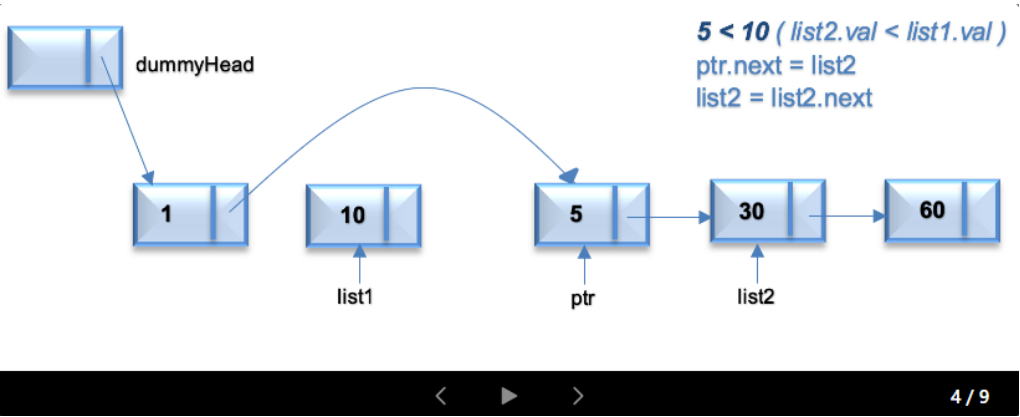
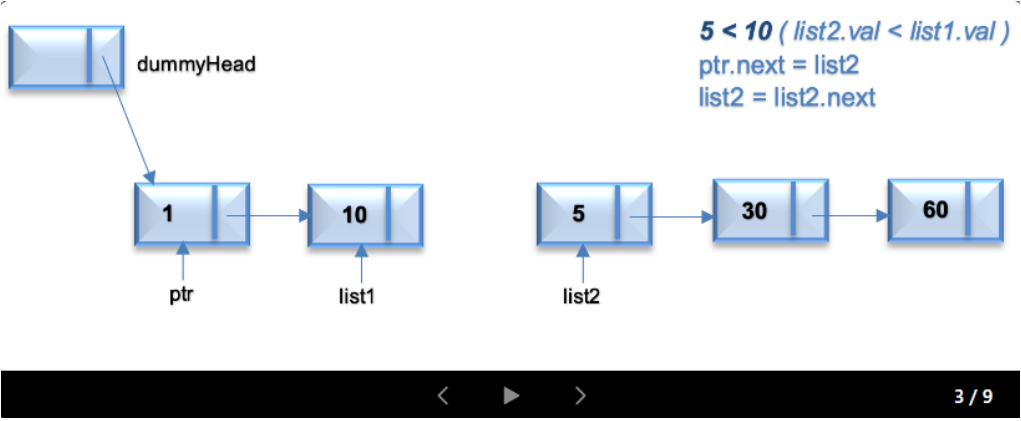
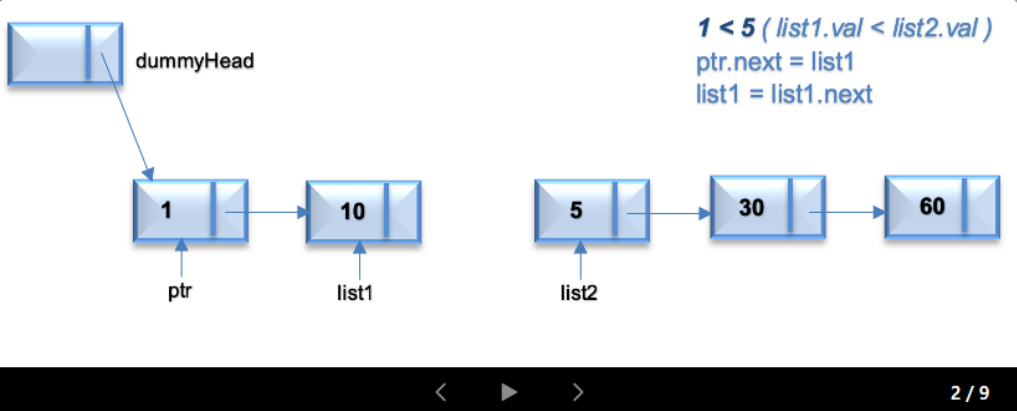
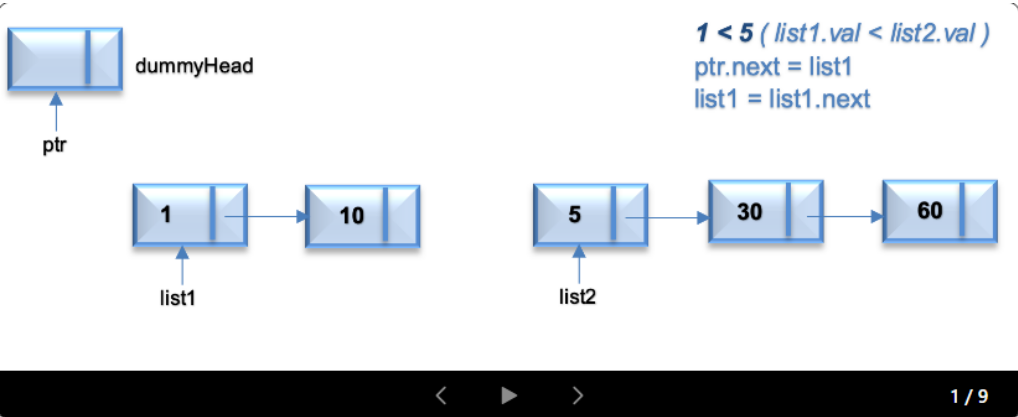
* Recursively split the original list into two halves. The split continues until there is only one node in the linked list (Divide phase). To split the list into two halves, we find the middle of the linked list using the Fast and Slow pointer approach as mentioned in [Find Middle Of Linked List](https://leetcode.com/problems/middle-of-the-linked-list/).
* Recursively sort each sublist and combine it into a single sorted list. (Merge Phase). This is similar to the problem [Merge two sorted linked lists](https://leetcode.com/problems/merge-two-sorted-lists/)

The process continues until we get the original list in sorted order.

For the linked list = [10,1,60,30,5], the following figure illustrates the merge sort process using a top down approach.



If we have sorted lists, list1 = [1,10] and list2 = [5,30,60]. The following animation illustrates the merge process of both lists into a single sorted list.



class Solution {

public ListNode sortList(ListNode head) {

if (head == null || head.next == null)

return head;

ListNode mid = getMid(head);

ListNode left = sortList(head);

ListNode right = sortList(mid);

return merge(left, right);

}

ListNode merge(ListNode list1, ListNode list2) {

ListNode dummyHead = new ListNode();

ListNode tail = dummyHead;

while (list1 != null && list2 != null) {

if (list1.val < list2.val) {

tail.next = list1;

list1 = list1.next;

tail = tail.next;

} else {

tail.next = list2;

list2 = list2.next;

tail = tail.next;

}

}

tail.next = (list1 != null) ? list1 : list2;

return dummyHead.next;

}

ListNode getMid(ListNode head) {

ListNode midPrev = null;

while (head != null && head.next != null) {

midPrev = (midPrev == null) ? head : midPrev.next;

head = head.next.next;

}

ListNode mid = midPrev.next;

midPrev.next = null;

return mid;

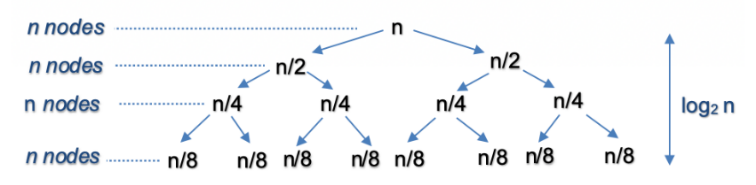
}

}

**Complexity Analysis**

* **Time Complexity:** O(nlog⁡n), where n is the number of nodes in linked list. The algorithm can be split into 2 phases, Split and Merge.

Let's assume that nis power of 2. For n = 16, the split and merge operation in Top Down fashion can be visualized as follows



***Split***

The recursion tree expands in form of a complete binary tree, splitting the list into two halves recursively. The number of levels in a complete binary tree is given by log⁡2{n}. For n=16, number of splits = log⁡2{16}=4

***Merge***

At each level, we merge n nodes which takes O(n) time. For n=166, we perform merge operation on 16 nodes in each of the 4 levels.

So the time complexity for split and merge operation is O(nlog⁡n)

* **Space Complexity:** O(log⁡n), where n is the number of nodes in linked list. Since the problem is recursive, we need additional space to store the recursive call stack. The maximum depth of the recursion tree is log⁡n