leetcode.com

Sort List - LeetCode

6-7 minutes

Solution

Overview

The problem is to sort the linked list in $\mathcal{O}(n \log n)$ time and using only constant extra space. If we look at various sorting algorithms, Merge Sort is one of the efficient sorting algorithms that is popularly used for sorting the linked list. The merge sort algorithm runs in $\mathcal{O}(n \log n)$ time in all the cases. Let's discuss approaches to sort linked list using merge sort.

> Quicksort is also one of the efficient algorithms with the average time complexity of $\mathcal{O}(n \log n)$. But the worst-case time complexity is $\mathcal{O}(n^2)$. 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 $\mathcal{O}(1)$ 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. 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.

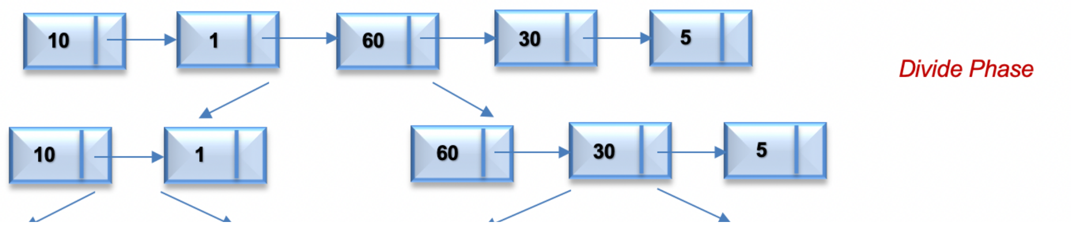
The Top Down approach for merge sort recursively splits the original list into sublists of equal sizes, sorts each sublist independently, and eventually merge the sorted lists. Let's look at the algorithm to implement merge sort in Top Down Fashion.

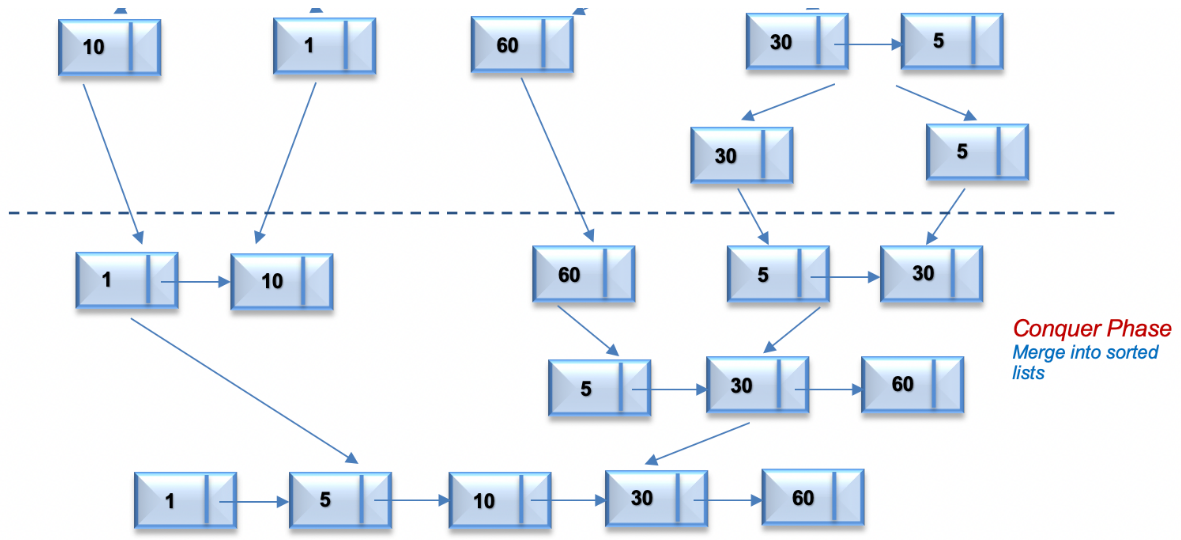
Algorithm

- 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.
- 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

  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.





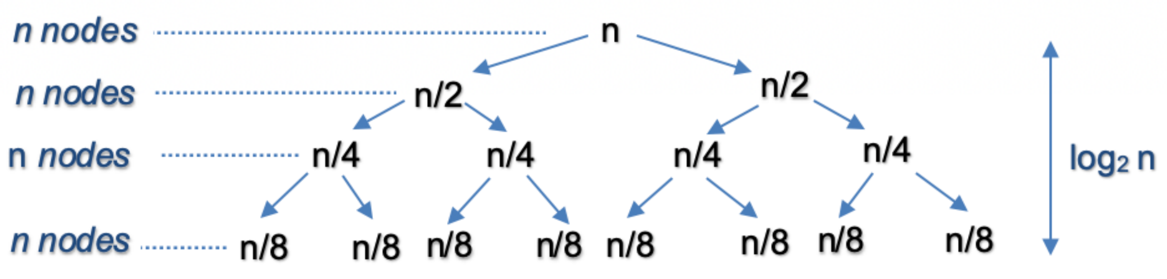
If we have sorted lists, list1 = `[1,10]` and list2 = `[5,30,60]`. The following animation illustrates the merge process of both lists into a single sorted list.

Current

Complexity Analysis

- Time Complexity: $\mathcal{O}(n \log n)$, 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 n is 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 invalid-markup$n$. For $n = 16$, number of splits = invalid-markup$16 = 4$

Merge

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

So the time complexity for split and merge operation is $\mathcal{O}(n \log n)$

- Space Complexity: $\mathcal{O}(\log n)$, 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 $\log n$

Approach 2: Bottom Up Merge Sort

Intuition

The Top Down Approach for merge sort uses $\mathcal{O}(\log n)$ extra space due to recursive call stack. Let's understand how we can implement merge sort using constant extra space using Bottom Up Approach.

The Bottom Up approach for merge sort starts by splitting the problem into the smallest subproblem and iteratively merge the result to solve the original problem.

- First, the list is split into sublists of size 1 and merged iteratively in sorted order. The merged list is solved similarly.
- The process continues until we sort the entire list.

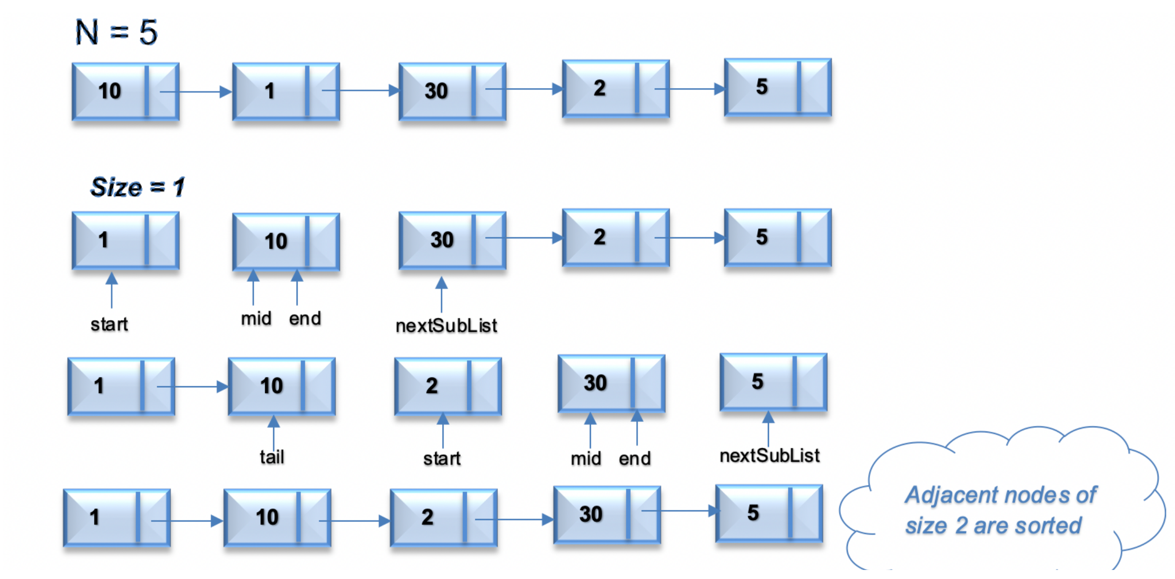
This approach is solved iteratively and can be implemented using constant extra space. Let's look at the algorithm to implement merge sort in Bottom Up Fashion.

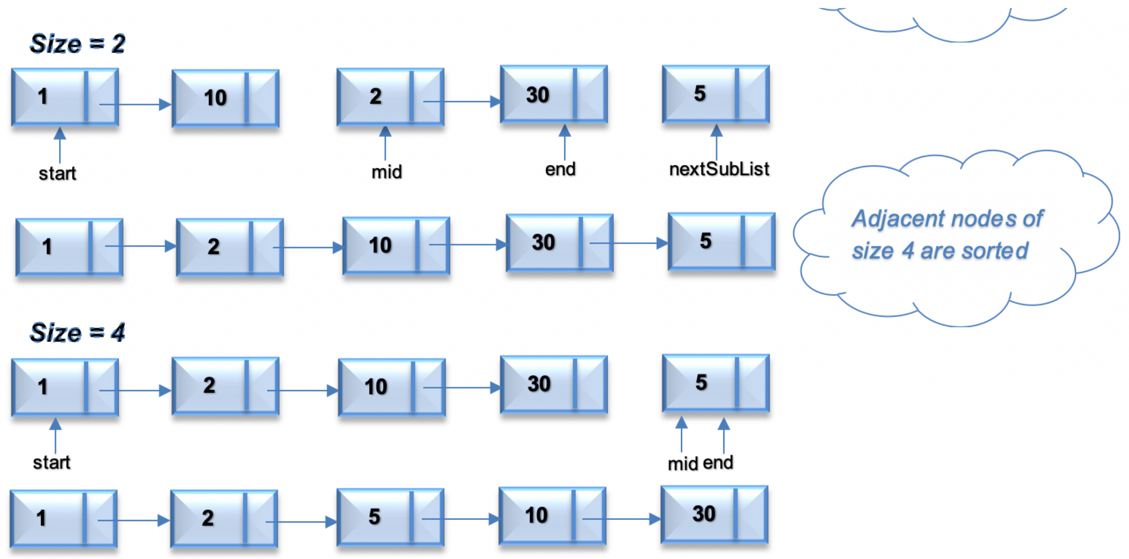
Algorithm

Assume, n is the number of nodes in the linked list.

- Start with splitting the list into sublists of size 1. Each adjacent pair of sublists of size 1 is merged in sorted order. After the first iteration, we get the sorted lists of size 2. A similar process is repeated for a sublist of size 2. In this way, we iteratively split the list into sublists of size $1, 2, 4, 8$.. and so on until we reach n.
- To split the list into two sublists of given size beginning from start, we use two pointers, mid and end that references to the start and end of second linked list respectively. The split process finds the middle of linked lists for the given size.
- Merge the lists in sorted order as discussed in *Approach 1*
- As we iteratively split the list and merge, we have to keep track of the previous merged list using pointer tail and the next sublist to be sorted using pointer nextSubList.

For the linked list = `[10,1,30,2,5]`, the following figure illustrates the merge sort process using a Bottom Up approach.

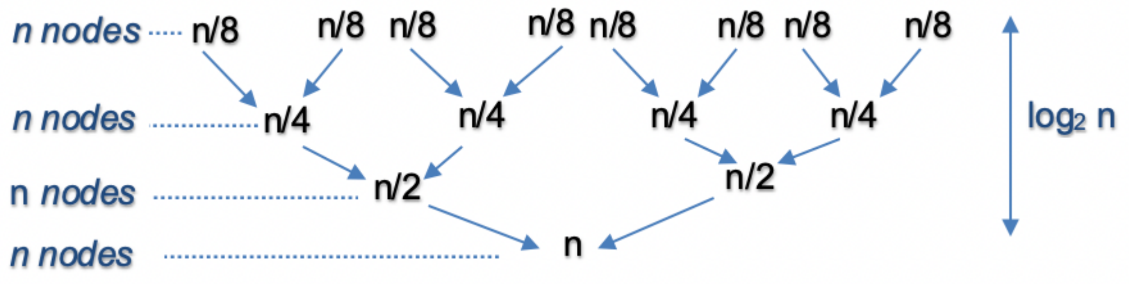




Complexity Analysis

- Time Complexity: $\mathcal{O}(n \log n)$, where n is the number of nodes in linked list. Let's analyze the time complexity of each step:

1. Count Nodes - Get the count of number nodes in the linked list requires $\mathcal{O}(n)$ time.
2. Split and Merge - This operation is similar to *Approach 1* and takes $\mathcal{O}(n \log n)$ time. For `n = 16`, the split and merge operation in Bottom Up fashion can be visualized as follows



This gives us total time complexity as $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

- Space Complexity: $\mathcal{O}(1)$ We use only constant space for storing the reference pointers `tail` , `nextSubList` etc.