21. Merge Two Sorted Lists

Approach 1: Recursion

Intuition

We can recursively define the result of a `merge` operation on two lists as the following (avoiding the corner case logic surrounding empty lists):

$$\begin{cases} list1[0] + merge(list1[1:], list2) & list1[0] < list2[0] \\ list2[0] + merge(list1, list2[1:]) & otherwise \end{cases}$$

Namely, the smaller of the two lists' heads plus the result of a `merge` on the rest of the elements.

Algorithm

We model the above recurrence directly, first accounting for edge cases. Specifically, if either of `l1` or `l2` is initially `null`, there is no merge to perform, so we simply return the non-`null` list. Otherwise, we determine which of `l1` and `l2` has a smaller head, and recursively set the `next` value for that head to the next merge result. Given that both lists are `null`-terminated, the recursion will eventually terminate.

Java Python3 Copy

Complexity Analysis

- Time complexity : $O(n + m)$

  Because each recursive call increments the pointer to `l1` or `l2` by one (approaching the dangling `null` at the end of each list), there will be exactly one call to `mergeTwoLists` per element in each list. Therefore, the time complexity is linear in the combined size of the lists.

- Space complexity : $O(n + m)$

  The first call to `mergeTwoLists` does not return until the ends of both `l1` and `l2` have been reached, so $n + m$ stack frames consume $O(n + m)$ space.

Approach 2: Iteration

Intuition

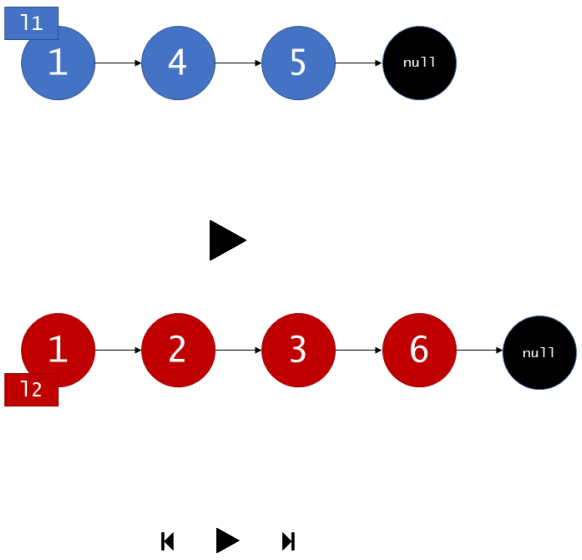
We can achieve the same idea via iteration by assuming that `l1` is entirely less than `l2` and processing the elements one-by-one, inserting elements of `l2` in the necessary places in `l1`.

Algorithm

First, we set up a false " `prehead` " node that allows us to easily return the head of the merged list later. We also maintain a `prev` pointer, which points to the current node for which we are considering adjusting its `next` pointer. Then, we do the following until at least one of `l1` and `l2` points to `null`: if the value at `l1` is less than or equal to the value at `l2`, then we connect `l1` to the previous node and increment `l1`. Otherwise, we do the same, but for `l2`. Then, regardless of which list we connected, we increment `prev` to keep it one step behind one of our list heads.

After the loop terminates, at most one of `l1` and `l2` is non-`null`. Therefore (because the input lists were in sorted order), if either list is non-`null`, it contains only elements greater than all of the previously-merged elements. This means that we can simply connect the non-`null` list to the merged list and return it.

To see this in action on an example, check out the animation below:



1 / 35

Java | Python3 Copy

```java
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // maintain an unchanging reference to node ahead of the return node.
        ListNode prehead = new ListNode(-1);

        ListNode prev = prehead;
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                prev.next = l1;
                l1 = l1.next;
            } else {
                prev.next = l2;
                l2 = l2.next;
            }
            prev = prev.next;
        }

        // exactly one of l1 and l2 can be non-null at this point, so connect
        // the non-null list to the end of the merged list.
        prev.next = l1 == null ? l2 : l1;

        return prehead.next;
    }
}
```

Quick Navigation View in Article (/articles/merged-two-sorted-lists/)

Notes

Complexity Analysis

- Time complexity : $O(n + m)$

  Because exactly one of `l1` and `l2` is incremented on each loop iteration, the `while` loop runs for a number of iterations equal to the sum of the lengths of the two lists. All other work is constant, so the overall complexity is linear.

- Space complexity : $O(1)$

  The iterative approach only allocates a few pointers, so it has a constant overall memory footprint.

Comments: 2

Sort By

Type comment here... (Markdown is supported)

Preview Post

sfdye (/sfdye) ★ 45 September 1, 2018 12:28 PM

For Approach 2 Python 3 solutions, the second last line can be just `prev.next = l1 or l2`

1 Share Reply

Hai_dee (/hai_dee) ★ 56 August 11, 2018 5:15 AM

I'm being somewhat pedantic, however, this comment is somewhat incorrect.

```
# exactly one of l1 and l2 can be non-null at this point, so connect
```

Read More

2 Share Reply

SHOW 2 REPLIES