Topic Introduction

Welcome to today’s coding interview prep!

Today, we’ll be diving into a timeless classic in algorithm interviews: **merging sorted data structures**. Whether you’re working with arrays, linked lists, or even heaps, merging sorted sequences is a fundamental skill that pops up everywhere, from searching in databases to handling streaming data.

What’s the pattern?
Whenever you have multiple ordered sequences and need to create a single sorted sequence, there’s a very efficient way to do it: **the two-pointer technique** (or its cousins, like the min-heap for multiple lists). Instead of sorting everything from scratch, you cleverly walk through the sequences, comparing and choosing the smallest/largest element at each step. This saves tons of time!

When should you use it?

- When you’re given two or more already sorted lists/arrays and need a single sorted result.
- When space or time efficiency is crucial, especially if in-place merging is required.

A simple example:
Suppose you have two sorted arrays:
A = [1, 3, 5]
B = [2, 4, 6]

You want to merge them into one sorted array.
Start with pointers at the beginning of both arrays. Compare the pointed values, pick the smaller one, move the pointer. Repeat!

Result: [1, 2, 3, 4, 5, 6]

Let’s see how this plays out in three popular LeetCode problems, starting from the basics and ramping up the challenge!

Problem 1: Merge Two Sorted Lists

LeetCode #21: Merge Two Sorted Lists

Problem statement (in plain English):
Given the heads of two sorted linked lists, combine them into a single sorted linked list, and return its head. Your new list should use the original nodes (no creating new nodes).

Example

Input:
List 1: 1 -> 3 -> 5
List 2: 2 -> 4 -> 6

Output:
1 -> 2 -> 3 -> 4 -> 5 -> 6

Thought process:

- Both lists are already sorted.
- At each step, compare the current nodes of both lists.
- Attach the node with the smaller value to the merged list, and move that pointer forward.
- Repeat until you've processed both lists.

Try it yourself!

Before moving on, grab pen and paper and try merging these two lists:

List 1: 1 -> 2 -> 4

List 2: 1 -> 3 -> 4

What would the merged list look like?

Take a moment to try solving this on your own before reading the solution.

Solution (Two-Pointer Merge)

We'll use a dummy node to simplify the process, and a pointer `current` to track where we're building the merged list.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1, l2):
    # Create a dummy node to simplify edge cases
    dummy = ListNode()
    current = dummy

    # Traverse both lists
    while l1 and l2:
        # Compare values and attach the smaller node
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # Attach any remaining nodes
    if l1:
        current.next = l1
    else:
        current.next = l2
    # Return the merged list (skip dummy)
    return dummy.next
```

Time and Space Complexity

- **Time:** O(n + m), where n and m are the lengths of the two lists.
- **Space:** O(1) (in-place, only using pointers).

Step-by-Step Example

Suppose List 1: 1 -> 4, List 2: 2 -> 3.

- 1 vs 2: take 1
- 4 vs 2: take 2
- 4 vs 3: take 3
- 4 left: append 4

Result: 1 -> 2 -> 3 -> 4

Your turn:

Try merging List 1: 1 -> 3 -> 5, List 2: 2 -> 4. Do a dry-run on the code above!

Problem 2: Merge Sorted Array

LeetCode #88: Merge Sorted Array

Problem statement (in plain English):

You have two integer arrays, nums1 and nums2, both sorted in non-decreasing order.

- nums1 has enough space at the end to hold all elements of nums2.
- You're given the number of initialized elements in nums1 (`m`) and nums2 (`n`).
- Merge nums2 into nums1 as one sorted array, in-place.

Example

Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6], n = 3

Output:
nums1 = [1,2,2,3,5,6]

Thought process:

- Unlike the previous problem, this one asks for **in-place** merging.
- If we start merging from the beginning, we'll overwrite elements in nums1.
- Instead, merge from the **end** of both arrays, placing the largest values at the end of nums1.

Where does this differ from Problem 1?

- Instead of linked lists, we work with arrays and need to do the merging in-place, without extra space.
- The merge logic is similar, but we need to be careful with array indices.

Try this test case yourself:

nums1 = [4,5,6,0,0,0], m = 3

nums2 = [1,2,3], n = 3

What should nums1 look like after merging?

Take a moment to try solving this on your own before reading the solution.

Solution (Two-Pointer from the End)

We use three pointers:

- `p1` points to the last element in the initialized part of nums1.
- `p2` points to the last element in nums2.
- `p` points to the last index of nums1 (where we'll write the next largest element).

```
def merge(nums1, m, nums2, n):
    p1 = m - 1  # Last initialized element in nums1
    p2 = n - 1  # Last element in nums2
    p = m + n - 1  # Last index in nums1

    # Merge in reverse order
    while p1 >= 0 and p2 >= 0:
        if nums1[p1] > nums2[p2]:
            nums1[p] = nums1[p1]
            p1 -= 1
        else:
            nums1[p] = nums2[p2]
            p2 -= 1
        p -= 1

    # If nums2 still has elements, copy them
    while p2 >= 0:
        nums1[p] = nums2[p2]
        p2 -= 1
        p -= 1
```

Time and Space Complexity

- **Time:** O(m + n)
- **Space:** O(1) (in-place)

Step-by-Step Example

nums1 = [4,5,6,0,0,0], m = 3
nums2 = [1,2,3], n = 3

- Compare 6 and 3: place 6 at position 5
- Compare 5 and 3: place 5 at position 4
- Compare 4 and 3: place 4 at position 3
- nums2 still has 1,2,3: copy them to nums1[0:2]

Result: [1,2,3,4,5,6]

Your turn:
Try nums1 = [2,0], m = 1; nums2 = [1], n = 1. Dry-run the code!

Did you know?
You could also solve this by creating a new array, but that would use extra space. For interviews, always ask if in-place is required!

Problem 3: Merge k Sorted Lists

LeetCode #23: Merge k Sorted Lists

Problem statement (in plain English):
Given an array of k sorted linked lists, merge them all into one sorted linked list and return its head.

Example

```
Input:
lists = [
  1 -> 4 -> 5,
  1 -> 3 -> 4,
  2 -> 6
]
```

```
Output:
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6
```

Thought process:
- This is an extension of Problem 1, but now with **k** lists!
- If you merge one by one, it’s not efficient for large k.
- Instead, use a **min-heap (priority queue)** to always get the smallest current node among all lists.

How is this different from earlier problems?
- We now deal with more than two lists.
- The two-pointer merge approach doesn’t scale well for k lists; instead, we use a heap to efficiently pick the smallest node at each step.

Try this test case:

lists = [
  2 -> 5,
  1 -> 9,
  3 -> 4 -> 6
]

What would the merged list look like?

Take a moment to try solving this on your own before reading the solution.

Solution (Min-Heap / Priority Queue)

We use a heap to keep track of the smallest head among all lists.

```python
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    # Needed to compare ListNode objects in the heap
    def __lt__(self, other):
        return self.val < other.val

def mergeKLists(lists):
    heap = []
    # Initialize the heap with the first node of each list
    for i, l in enumerate(lists):
        if l:
            heapq.heappush(heap, (l.val, i, l))

    dummy = ListNode()
    current = dummy

    while heap:
        val, i, node = heapq.heappop(heap)
        current.next = node
        current = current.next
        if node.next:
            # Push the next node from the same list into the heap
            heapq.heappush(heap, (node.next.val, i, node.next))

    return dummy.next
```

Explanation:

- We keep the heap size to k.
- The tuple in the heap is (node value, list index, node itself) to avoid errors if node values are equal.
- At each step, pop the smallest node, add it to the result, and push its next node if it exists.

Time and Space Complexity

- **Time:** O(N log k), where N is the total number of nodes.
- **Space:** O(k) for the heap.

Walkthrough Example

Suppose lists = [1->4, 2->3, 0->5]:

- Heap starts with (0, 2), (1, 0), (2, 1)
- Pop 0, add 5 from list 2
- Heap: (1, 0), (2, 1), (5, 2)
- Pop 1, add 4 from list 0
- Continue...

Your turn:

Try merging these lists:

[1 -> 2 -> 2], [1 -> 1 -> 2], [2 -> 2].

Write out the steps or dry-run the code!

Alternate approaches:

You could also use a divide-and-conquer method (merge pairs of lists recursively, like merging in merge sort), which also gives O(N log k) time. Give that a try after you're done!

Summary and Next Steps

Let's recap:

- All three problems are about **merging sorted data structures**, but each adds a layer of challenge.
    - Merge Two Sorted Lists: foundation, two-pointer technique.
    - Merge Sorted Array: in-place merging, careful index management.
    - Merge k Sorted Lists: scaling up, using a heap to efficiently merge many lists.

Key patterns to remember:

- Use two pointers for merging two sorted sequences.
- For k sequences, use a heap or divide-and-conquer (don't try pairwise merges in a brute-force way).
- Always pay attention to constraints: in-place? arrays or linked lists? time/space limits?

Common pitfalls:

- Forgetting to handle leftover elements in either list/array.
- Overwriting data when merging arrays in-place.

- Not handling empty lists or arrays.
- Not updating pointers correctly in linked list merges.
- For heaps, forgetting to handle duplicate values/heap tie-breakers.

Action List:

- Solve all three problems yourself to reinforce the techniques.
- Try other problems involving merging, like “Sorted Merge of Intervals” or “Find Median of Two Sorted Arrays.”
- Read other people’s solutions to see different (and sometimes more elegant) approaches.
- If you find the heap or divide-and-conquer approach tricky, don’t worry! Focus first on mastering the two-pointer basics.
- Practice writing code on paper or a whiteboard, as you’ll do in interviews.

Remember, not every new problem will look exactly like these, but if you spot sorted sequences and “merge” in the description, these patterns will serve you well. Keep practicing—mastery comes with time!

Happy coding, and see you in the next PrepLetter!