Summary

Time complexity: O(N log k) The heap has size k at any point of time and we pop and push exactly N times where N is the total number of nodes (including all lists). Since each pop/push operation takes O(log k) time, the overall time complexity of this solution is O(N log k).
Space complexity: O(k) to store the nodes on the heap.

Problem Pattern

1. You are given K linked lists of different sizes and you need to merge these into 1 linked list such that the resulting list is also sorted.
2. There can be many brute force approaches for this, such as adding all elements to an array first and sort and iterate over this array to get the proper value of nodes. Then you can create a new sorted linked list and extend it with the new nodes.
3. But this method is very inefficient. Since all lists are sorted, we know that the smallest element of each list is the first element of this list (total k elements). So, wouldn't the first element of our result list be one of these k elements?
4. So at any time, we need to compare only K elements. Now after picking one of these, how will you decide which linkedlist you'll add the next element from?
5. What if you create a min-heap? What will be the size of this min heap?

Problem Approach

1. The idea is to construct a min-heap of size K and insert the first node of each list into it.
2. Then we pop the root node (having minimum value) from the heap and insert the next node from the same list as the popped node. We repeat this process until the heap is exhausted.
3. We keep forming our resultant linked list with these popped nodes.

Problem Pseudocode

```
ListNode mergeKLists(ListNode[] lists) {

    new PriorityQueue<Node> pq


    for (i=0 to k)

        pq.add(list[i])


    head = null, last = null

    while (!pq.isEmpty()) {

        Node min = pq.poll()


        if (head == null)

            head = last = min

        else

            last.next = min

            last = min



        if (min.next != null)

            pq.add(min.next)

    }

    return head

}
```

Alternate Approaches

1. A simple solution would be to connect all linked lists into one list (order doesn't matter). Then use the merge sort algorithm for linked list to sort the list in ascending order. The worst case time complexity of this approach will be O(N log N) where N is the total number of nodes present in all lists. Also, this approach does not take advantage of the fact that each list is already sorted.
2. Above approach reduces the time complexity to O(N log K) but takes O(K) extra space for heap. We can solve this problem in constant space using Divide and Conquer.

We already know that merging of two linked lists can be done in O(n) time and O(1) space (For arrays O(n) space is required). The idea is to pair up K lists and merge each pair in linear time using O(1) space. After the first cycle, K/2 lists are left. After the second cycle, K/4 lists are left and so on. We repeat the procedure until we have only one list left.