1. Problem Statement

I chose problem 23 on leetcode for my final project. Merge *k* sorted lists. In this problem, "you are given an array of *k* linked-lists *lists*, each linked-list is sorted in ascending order." The goal is to "merge all the linked-lists into one sorted linked-list and return it."

This particular problem relates to the linked-list data structure and can be solved in numerous ways. Related topics include linked lists, divide and conquer recursion, heaps and priority queues, and merge sort.

The head nodes of the linked-lists are all in the *lists* array, so it's the programmer's job to figure out how to quickly merge all the linked-lists. An example of an input is: [[1->4->5], [1->3->4], [2->6]], the *lists* array would be [1, 1, 2], where each integer also has a pointer to its linked-list's next value.

2. Solutions and Analysis

My first idea for this problem was to iterate through the whole array and find the minimum value of the current nodes, then make that node the next node of the last node in a sorted global linked list, then repeat until all linked lists in the array have been merged. With this method, I was attempting to brute force the array, not using any particularly complex algorithms or other data structures. However, this first submission for the problem was a failure as the time limit for the code was reached, though it did complete all but one of the test cases. The time complexity of this first approach is O(*nk*), where *n* is the total number of nodes in all linked lists, and *k* is the number of linked lists in the array. This is because the code was run *n* number of times and had to iterate through *k* nodes each time to find a new minimum. (see code 1)

The next iteration of my code took a similar approach to the problem, with a single modification. This time, when a linked list was fully iterated through, the list was deleted from the array. This way, with every fully emptied linked list, the iterations through the array would gradually speed up. This version of my code resulted in my first accepted submission. This submission had a runtime of 5347ms (top 95%) and a memory usage of 19.3mb (top 5%). Given that the code iterates through the entire array every time, it's no surprise that the code performed as slow as it did. Though, now that I had a successful submission, I began to look to begin optimizing the code with a focus on runtime. The time complexity of this approach is also O(*nk*), where *n* is the total number of nodes in all linked lists, and *k* is the number of linked lists in the array. The reason this iteration of my code and the last have the same time complexity is because the loop would still have to check all the lists in the array to see if they were empty. (see code 2)

The final iteration of my code took a new approach to the problem. Rather than iterating through the entire array, the code would now take a divide and conquer approach. It splits the array in half, then recursively calls the merge function on both halves until the arrays reach a size of 1 or 2 linked lists. Then, the code merges those 2 linked lists and works its way back up the recursion until finally merging 2 large linked lists to get the final list. This submission had a runtime of 95ms (top 50%) and a memory usage of 19.4mb (top 6%). The new divide and conquer approach broke the problem down into much smaller, easier to handle pieces, unlike my previous attempts which iterated through the whole array each time to find a new minimum. The time complexity of this final entry is O(nlog(k)), where, again, *n* is the total number of nodes in all linked lists, and *k* is the number of linked lists in the array. This is because the code would still have to compare *n* number of nodes but would only have to run log(*k*) times due to the recursion. (see code 3)

3. Comments and Discoveries

My thought process going into this problem was to first get a working submission, then use some of the algorithms and data structures taught in class to make the code faster. When my submission was first accepted, I was not expecting it to be as slow as it was. This ridiculously slow runtime really emphasized the importance of what was taught in class; it really changed my perception of some of these algorithms and opened an eye.

After my first successful submission, I realized that, especially in this case of an array of linked lists, it's much faster to break the whole list down into parts to recursively build new linked lists from the smaller parts than to iterate through the entire array, adding a single node to the end of a single linked list per iteration. No matter the problem, there is always going to be a faster way of completing it, and I know for a fact that my solution is definitely not the fastest, though it is very average. Moreover, one thing I did not mention were the numerous insignificant submissions I made between my successful submissions; in each one, I only tweaked a small piece of logic in my code with a goal of reducing runtime, but in each case, there was not a significant change.

I also realized that this problem did not really test my coding abilities, but rather my logical thinking abilities. Much of programming does this; it was not hard to get the input and manipulate it, but the logic and process behind manipulating the input the way you want can be pretty difficult to get right.

Additionally, thanks to this project, I finally began to use leetcode. Previously, I was hesitant to create an account and start solving problems, but this project led me to discover that the platform is not as intimidating as I once thought it to be. Leading up to graduation, I will try to take advantage of the website and flush out my programming and logical skills, applying what I've learned here in CSCI 3320.

Overall, I really enjoyed this experience. I do feel like I've learned something new.

4. Code

All submissions include the following piece of code meant to help the programmer:

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
```

Below is the code used in my first submission.

```
class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        if lists:
            if len(lists) == 1:
                return lists[0]
            else:
                return self.mergeLists(lists)

    def mergeLists(self, allLists):
        currNodes = []
        for headNode in allLists:
            currNodes.append(headNode)

        headNode = None
        prevNode = None
        while currNodes.count(None) != len(currNodes):
            minNode = None
            minNodeIndex = None
            for index, node in enumerate(currNodes):
                if not minNode or (node and node.val <= minNode.val):
                    minNode = node
                    minNodeIndex = index
            if not headNode:
                headNode = minNode
            if prevNode:
                prevNode.next = minNode
            currNodes[minNodeIndex] = currNodes[minNodeIndex].next
            prevNode = minNode
        return headNode
```

Below is the code used in my first successful submission.

```
class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        if lists:
            if len(lists) == 1:
                return lists[0]
            else:
                return self.mergeLists(lists)

    def mergeLists(self, allLists):
        currNodes = []
        for headNode in allLists:
            currNodes.append(headNode)

        headNode = None
        prevNode = None
        while currNodes.count(None) != len(currNodes):
            minNode = None
            minNodeIndex = None
            for index, node in enumerate(currNodes):
                if not minNode or (node and node.val <= minNode.val):
                    minNode = node
                    minNodeIndex = index
            if not headNode:
                headNode = minNode
            if prevNode:
                prevNode.next = minNode
            if currNodes[minNodeIndex].next:
                currNodes[minNodeIndex] = currNodes[minNodeIndex].next
            else:
                del currNodes[minNodeIndex]
            prevNode = minNode
        return headNode
```

Below is the code for my most recent successful submission.

```
class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        if lists:
            if len(lists) == 1:
                return lists[0]
            else:
                mid = len(lists) // 2
                left, right = self.mergeKLists(lists[mid:]),
self.mergeKLists(lists[:mid])
                return self.mergeLists(left, right)

    def mergeLists(self, left, right):
        headNode = None
        prevNode = None
        currNode = None
        while left or right:
            if left and (not right or left.val <= right.val):
                currNode = left
                left = left.next
            else:
                currNode = right
                right = right.next
            if not headNode:
                headNode = currNode
            if prevNode:
                prevNode.next = currNode
            prevNode = currNode
        return headNode
```

References

“23. Merge k Sorted Lists.” Leetcode. https://leetcode.com/problems/merge-k-sorted-lists/