1b. Especially if strings got very long, the amount of time it would take to return all the potential encodings would increase exponentially per character addition, and create an extremely inefficient time complexity.

1c. Sets would be the most useful data structure when solving this problem. Because they store no duplicate values, they would only store the unique value of the strings inputted into them. Then their lengths could easily be compared to show if the encodings are equal without having to use any loops.

2b. The naive solution would be to go through every element of the nums list and see if every sum combination would equal the target value. This would have a O(n^2) time complexity, since it would likely need a nested for loop to go through every single index combination.

2c. After reading 9 in index 0, I could finish if I had 2, After reading 4 in index 1, I could finish if I had 7.

These could be captured in a dictionary that corresponds the values of nums to the amount needed to equal the target. A condition could be made so if that remainder value exists, the indices of the original value and the remainder value could be returned in the form of a set. This ensures that the whole list does not need to be traversed for every combination of sums.

2e. The naive solution space complexity is constant (O(1)) because it would only need a constant amount of memory space for the looping vars and return set value. Meanwhile, the efficient solution has a higher space complexity of O(n) which on paper is less efficient space wise than the naive solution. However, the efficient solution performs in O(n) time while the naive solution performs in quadratic time complexity, which is significantly faster. Therefore, it would be wiser to choose the solution with lower time complexity, especially with a question that can have large input list sizes. Using a quadratic time complexity function in that case would be the larger future detriment.

3b. It would instead have to find the minimum head and node values from a list containing k ListNodes, meaning it would have to traverse and find the first minimum value to place in the output list from k sorted linked lists instead of just 2. Either this, or use a data structure that holds some form of numerical sorting so all the terms can be merely added and then outputted in a sorted manner.

3c. Elements would be touched a lot, given that comparisons of numerical size have to be made between every element of every ListNode to find the minimum of every “column”. For example, in a list of [9,10], [3,4,5,5], [5,6,7], [2,8], [1,3], the first terms alone would have to be touched many times when comparing if 9 is less than 3, 3 is less than 2 and 1, and etc.

3d. A priority queue/heapq would be helpful in this case. Although it does not fully sort each element numerically, it always puts the minimum value first, which would make it easy to iterate through the dataset and pop terms in numerical order. This would also help with the element

touching problem, in that we would not have to continuously loop through the entire set of values to find the minimum; we can use the properties of how a heapq sorts to return a sorted LinkedList

3e. My plan uses a heapq/priority queue to collect the first elements from each of the ListNodes, and then cycles through the rest of the elements of each Listnode to assemble a singular sorted ListNode. The caveat is that the number of elements present in the heapq at a given time will always be <= the size of the list of ListNodes to preserve time and space complexity to O(n*log(k)) and O(k). It would uses the indices of the values within the ListNodes to properly match the values when they enter the hashmap, hence “enumerate” will be valuable.

3g. The time complexity of my solution is O(n * log(k)) because it runs through each node while still only allowing the heapq to hold as many items as there are in the list. Solutions are only touched once. The space complexity is O(k) considering that the number of spaces in memory are being held by the ListNode values within the heapq.

4a. The most ideal solution would utilize a dictionary that maps numbers to their frequencies via a while loop that runs through the inputted ListNode. Whilst this is happening, the max frequency could also be calculated, which would later help with finding which singular or multiple values serve as the mode. Once this is mapped, another while loop could be used to then add all elements of the dictionary with the maximum frequency to a set() representing mode, which is returned.

4c. The structural property is that the farthest left term is the least and farthest right term is the greatest, so there's some form of numerical ordering. Left and right trees could become their own sorted LinkedLists/ListNodes, which then combined with the ListNode of the root could become an ordered single sorted ListNode, and then the mode could be found. Furthermore, the BST, when converted into a LinkedList, orders the modes into consecutive streaks within the returned linkedlist, which makes it far easier to loop through that linked list and return the modes.

4f. An algorithm that only works on the BST version would have an advantage over a more generic algorithm under most circumstances. By nature of eliminating unneeded searches, the BST algorithm would normally be much more efficient. However, if the data takes an unbalanced form, resembling more of a Linked List, the BST algorithm might not be as efficient as the dictionary algorithm. In certain cases, the BST algorithm might also have memory storage issues that prevent it from being better than a dictionary based algorithm.