Joshua Wang CS313 HW1 Q4,5,6,10 White Paper

Question 4: This question asks us to compare the time sorting an array vs linked lists with bubble sort with insertion sort. The first step to this is being able to properly time these functions with a timer class, which tracks the current time in its creation and current time in its deletion, and through the difference of the start and the end gives us the runtime of a function. Since this bubble sort requires a nested loop to carry a largest number to the back every iteration, the array requires a swap to happen when a previous array is greater than the next array. The linked list holds the same concept, although instead of comparing indexes through +/-1 index, it can be done by comparing the info of a current node to its linked node. Insertion sort is a sort with similar complexity, utilizing a while loop within a for loop to compare a single element to its predecessors. For arrays, it indexes upwards in this while loop to make space for the inserted array, while for linked lists does insertion through link manipulation. Overall, array sorting was faster than linked sorting, with insertion sorting having the edge in both. This is likely because the while loop in insertion acts more efficiently in determining what data needs to be sorted, while the nested loops in bubble sorting usually result in the worst case all the time.

Question 5: This question requires a multi level sort that changes at above 10 elements, and then a timer comparison test with 2 library sorts. The sorts I chose to implement were bubble sort and quicksort. I chose bubble sort because it is easy to implement and its inefficiencies are nearly indistinguishable with a small number of elements. On the other hand, I chose quick sort for >10 because I speculated that library sorts would be more efficient than most sorts, especially when working with larger elements, and so a powerful sort would be fitting for when elements are greater than 10. The reason why quicksort is fitting for large amounts of data is that it relies on partitioning data, divvying it up into a way that reminds me of binary search. I compared my multisort to the library's sort and stable sort, and it surprised me when it came out to be quicker than both, even with a low and high number of elements. Perhaps those sorts were less efficient than I thought.

Question 6: This question asked to find a way to compare solutions to crack a vigenere cipher, then mutate via genetic algorithm to get closer to the real solution. This one was hard to understand, and so I ended up trying to do what was asked to the best of my ability although without doing exactly what was asked. At first I figured that maybe I could try to find the key through frequency analysis, attempting to find the factors that would determine the size of the keyword. Although I was able to somewhat get factors by getting the distance between repeating characters, the best factor to be chosen is hard to quantify through code as there are usually multiple factors to choose from. I ended up making my code take an encrypted string and "mutate" it one char at a time from an alphabet selection, moving to the next char if it matched the plaintext. Although I struggled with figuring out a way to crack the cipher, I suppose it is a testament to how secure it really is.

Question 10: This question asked to create a function that sorts linked lists so that odd numbers are at the beginning and even numbers are at the end. I ended up making something similar to a swap in which two temporary linked lists are created, and the original linked list is filtered so that odd data goes in the beginning of one list and even data goes in the beginning of the other. This way they can be read linearly back into the original list, and then deleted to free up space. The benefit of sorting in this method is that it also maintains the original order of the list, and does not require any links to be altered.