#4: Compare the times it takes to sort a random array vs a linked list with a bubble sort.

For this program to work, we need certain header files for generating random variables, which we will use to fill in the elements and nodes of our array and linked list, and for measuring time as the program runs, which will be for measuring how long each sort takes to execute (stdio.h, time.h, etc). We also need to make this program capable of creating nodes for linked lists. To do that, we code in a structure that allows for nodes to be declared, with an empty pointer ‘next’, which is used to refer to the next node in the list, and an empty ‘data’, which is the data that is meant to be passed into the node. Next, we establish functions for inserting nodes into a linked list, bubble sorting an array, and bubble sorting a linked list.

In the main function, we call in a random seed with srand(time(nullptr)). Without this, we would end up with the same random numbers every time rand() is called. Next, a constant int is used for the size of the array and linked list. The array and linked list are created, and a random number is passed into the first element and node. If we didn’t directly pass in a value into the first node of the list with head->data = array[0], we would end up with 0 for the first node’s data. The array and list is filled with the same random numbers for simplicity’s sake. The node insertion function works by first creating a new node to be inserted, then creates a temporary node that shares a pointer with “head”, then it passes "NULL" into inserted_Node->next, because there will be no node after this one. Then the data is passed into the new node, and then a while loop is used to traverse to the end of the list with the temporary node. Finally, it makes the last node in the list point to the new inserted node, making it the new last node.

Next, the functions for timing the two bubble sorts are called, and the time taken is passed into the integers “array_Time” and “lList_Time”. The start time is declared at the beginning of each function. For the array bubble sort, two for loops are used, then an if statement to check if the previous element in an array is larger than the next one. If so, it uses a third temporary variable to help swap the two elements around. The for loops are set up in such a way so that the loops go through every element in the list enough times so that the entire array and list is bubble sorted. Then the end time is declared, and the execution time is calculated and returned at the end. For the linked list sort, the same thing essentially happens with a few changes. “new_Head = &(*new_Head)->next;” changes “new_Head” to point to the next node so the loop can continue, otherwise the entire list won’t be sorted, only the first 2 elements.

The two measured times are then printed to console and then compared in an if loop, and whichever time is lower will be stated with cout. As it turns out, running the program with varying high sizes shows that a bubble sort for a linked list will always take longer than a bubble sort for an array. In fact, after running the program multiple times (in Visual Studio), it seems that the array bubble sort is roughly twice as fast as the linked list sort, with the array sort averaging around 2700 - 2900 milliseconds and the linked list bubble sort averaging around 5400 - 5800 milliseconds.