**Sorting project one**

Jacob Dougherty

Serigne Diouf

Nick Bierman

For the first group, we used a radix sort.  This sort is a recursive function that sorts the list by digit.  The list starts with the one’s digit, and then moves to the ten’s digit, and so on and so forth until the list is sorted.  This sort was relatively easy, once we were able to get a hang of how it worked and how the recursion would work.  We were able to adapt the code for the linked list and make it run on the array and then the dynamic array.  But there were a couple major challenges with coding the radix sort.  The first was figuring out the base case and how to get the sort to stop being recursive.  We were able to stop this by checking if the list is sorted each time the recursive method is called. The second issue was whether the function was able to be recursive at all, because it had to be able to know what digit to check.  We were able to fix this by calculating a variable called “digitCheck”. This variable took an inputted number that was increased by one each time the function ran.  This was then used to find the digit we were checking in the variable “spotToAdd”.  The time complexity of a radix sort is n+k, where n is the number of elements in the list and k is the largest amount of a single digit across the numbers in the list.

The second sorting method that we used was Merge sort. This algorithm by far gave us the most trouble, taking us a lot more time for the project. Merge sort very basically is splitting the list, breaking it down into pieces we can compare. Continually putting it back together in order until you have just one list instead of two. We completed this for an array, a dynamic array, and a linked list. The array and the dynamic array were fairly straight forward after we were able to claw our way through comparison statements and loops. The linked list was the hardest of the three sorts, having gotten several bugs and infinite loops that seemed to have stopped our progress in its tracks. We were only able to solve the problem after hours of trying to figure it out. Our problem ended up being that one of our linked list methods, more specifically our .remove()method, ended up not having a base condition in it, that if the tail points at itself, set the tail to null.  When we solved this issue, our merge sort and our quick sort began to run as expected.

For the third group, we used a quick sort. This sort picks an element as a pivot and partitions the array surrounding the pivot. Basically, what we did was we changed the way that the pivot value was chosen. So, we took average of every element and then found the closest element in value to the average. This found element was set to the pivot element. This takes care of flaw of the quicksort that when an extreme is chosen, no elements are swapped, thus wasting a pass through the list. But when the pivot element is toward the middle of the list, we are making the most of each pass and swapping the maximum number of elements, thus making this array more efficient. We did this with the array, dynamic array, circular linked list. We had trouble figuring out a lot of bugs with the Circular Linked list and the Array due to infinite recursion.

Circular Linked Lists are a very useful data structure when looking at it in terms of it not having to be stored in a non-sequential fashion.  The big theta values on many of the circular linked list functions show that this data structure is relatively slow because there is no direct access.  So, access on a quick list is Θ(n).  An add, as well as a remove, depends on the location in the list, due to lack of direct access.  One must iterate over every value in order to find the spot to remove.  Therefore at the front of the list, an add and remove share a Θ(1), while at the end of the list the add and remove time complexity is Θ(n).  A find is a Θ(n) for the same reason as the entire linked list is relatively slow.  One has to iterate over every single value in order to find the one it is looking for.