Lab 2 Report

The whole purpose of this lab was to get a better understanding on how linked lists and recursion works. First, we were instructed to make a linked list with a length of a given number and fill it with random numbers. Since each slot in the linked list has a random number in it, the entire list is pretty much out of order. Our next task was to make three methods, bubble sort, merge sort and quick sort, to sort the list full of random numbers. All of these methods do the same thing, sort the list, but some of the methods are way more efficient than others. Bubble-sort has a O(n^2) runtime, merge-sort and quick-sort are supposed to have O(nlogn) runtimes. This lab also required us to find the median of the list we filled. The median is the middle number, the number that is right between the smallest and largest numbers on the list.

**Proposed Solutions**

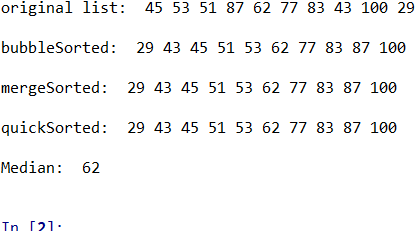
My initial attempt to make the bubble-sort method was to use two for-loops using the lengths of the lists, but I quickly realized that that approach was doomed. So, instead of using a pair of nested for-loops, I used while-loops. Step one was to first check to see if the list even had anything in it. I used an if-statement that checks to see if the head is none. Step two, I made a Boolean variable to basically represent whether a change was made or not, I used this for the first while-loop made. I change the Boolean variable right before making the second while-loop, so it can eventually exit the first loop after completing the second while-loop. The second while-loop goes through the entire list and checks each node to the node next to it, if they’re in the wrong I order, they get switched, and this goes on until the list is completely sorted.

The first check that had to be made in merge-sort was to make sure that the list provided had a longer length than one, because you can’t sort a list that only has one element, so I just returned it. So, the first if-statement checks the length of list. The merge-sort method also needed a helper method, the merge function, which compares the elements in two lists and appends the smaller element first to the new list being created. The merge-sort method is supposed to sort the given list by splitting it in half, which I do by appending the elements from the original list given to two empty lists, and then comparing the elements with each other. After splitting the list, I make two recursive calls, using the lists that were just created, to further split the list up. The very last step is to call the merge function to compare and combine the two lists into a new sorted list, then return the new list.

Quick-sort, like merge-sort, also required the list to be split up, but the list had to be split up around a pivot, which in my case ended up being the head of the list given. I had to compare the rest of the items in the list to the pivot item. If the items were less than the pivot item, they’d go in one list, if the items were greater than the pivot they’d go in another list. And like the merge-sort method, I made two recursive calls using the two new lists that resulted from splitting the list up. The last step is where I call a helper method which simply combines the two lists, and once the two lists are combined it results in the sorted version of the original list.

**Experimental Results**

|  |  |  |  |
| --- | --- | --- | --- |
| List Lengths | Bubble-Sort  #of calls made | Merge-Sort  #of calls made | Quick-Sort  #of calls made |
| 5 | 12 | 2 | 2 |
| 10 | 90 | 5 | 2 |
| 17 | 224 | 8 | 2 |



The table above is supposed to represent the number of recursive calls that had to be made for each of the required sorting methods bubble, merge and quick. Bubble-sort is clearly at a major disadvantage. When the list sizes grow bubble-sorts’ recursive calls sky rocket, because it has to go through the whole list to make sure each element is in its proper spot on the list. Merge-sort also grows with the number of inputs, but no where near the rate at which bubble-sort grows, and I also found that quick-sort remained constant for the most part, occasionally it’d change from 2 to 4, depending on the order of the initial list provided.

To make sure that my methods actually worked, I changed the list length many times. I checked even number lengths, I checked odd number lengths, and I even checked what would happen if the list only contained one element. I even got rid of the random numbers and manually put numbers in the list just to experiment and see if anything would change or break. Above is a screen-shot of a result from the lab I worked on.

**Conclusion**

Sorting may seem like an easy task, but after some-what completing this lab I have learned that sorting is no easy task, especially with linked-lists. Probably the biggest take-away I got from this lab is how certain methods are so much more efficient than others. Even if the methods do the exact same thing, which is the case for this lab (turning an un-sorted list into a sorted list), some methods are just way more efficient than others. This lab also showed me how recursive calls work like a function, like for example all of my methods return a list, so when I made my recursive calls, I assigned them to a list.

“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

Joey Roe