CS 314 Final Review — Linked List Sums

Linked Lists

Implement an instance method for the singly-linked `IntLinkedList` class which changes a nodes value to be the sum of its own value plus the value of all subsequent nodes in the list.

Complete the following method.

```
// Changes each node's int value to be the sum of itself plus the sum of
// all the values which come after it in the list.
// pre: size > 0
// post: size is unchanged
public void addValuesToRight() {
```

Here are some example calls to `addValuesToRight()`:

```
[1, 1, 1, 1, 1].addValuesToRight() => [5, 4, 3, 2, 1]
[1, 2, 3, 4, 5].addValuesToRight() => [15, 14, 12, 9, 5]
[0, 0, 0, 0, 2].addValuesToRight() => [2, 2, 2, 2, 2]
[5, 1, 2, 0].addValuesToRight() => [8, 3, 2, 0]
[3, 1, 4].addValuesToRight() => [8, 5, 4]
[7].addValuesToRight() => [7]
```

You may use the following `IntLinkedList` implementation:

```
public class IntLinkedList {
  IntNode first;
  int size;

  //Nested node class
  private static class IntNode{
    IntNode next;
    int value;
  }
}
```

Your solution should be as efficient as possible in terms of time.
You may create helper methods.
Do not create any new data structures or use any other Java classes or methods.

```
// Changes each node's int value to be the sum of itself plus the sum of
// all the values which come after it in the list.
// pre: size > 0
// post: size is unchanged
public void addValuesToRight() {
  sumOfValuesToRight(first);
}

private int sumOfValuesToRight(IntNode n){
  if(n == null)
    return 0;
  n.value += sumOfValuesToRight(n.next);
  return n.value;
}
```

Without creating any additional data structures, the most efficient way of solving this problem is to use recursion. One could naively use two nested loops although that approach would be $O(N^2)$. By contrast, this recursive approach has $O(N)$ time complexity. If this problem had involved a doubly-linked list, this problem would have been much easier because we could have traversed the list backwards: from right to left. Using recursion allows us to traverse this singly-linked list backwards by using the call stack. Notice that, unless we've hit the base case, the first thing our recurisve helper method does is make another recursive call using the next node in the list. Thus, the first thing that will happen is we will create N stack frames, one for each node in the list. Then, as we pop off the stack frames from the call stack, we are essentially traversing the list backwards (because stacks are LIFO). If you want to think of this in terms of explicit stacks, this is the equivalent of adding references to all of the `IntNode`s into a stack and then popping off from that stack.

Using recursion for this problem allows for very compact code which is more time efficient than if we had used nested loops. However, it accomplishes this by using $O(N)$ space, a classic time-space tradeoff.