Junyu Teoh
CSCD300
HW3 Progress Report

1) **private MyLinkedList reverse(ListNode Node)**
   Step 1) We first check if Node is empty (If it is, it means there is only one node, the head). If it is null, it is sorted, and we return a new MyLinkedList that is empty. This is also the base case.
   Step 2) If it is not, we recursively call the method reverse with parameter node.next. This essentially recursively calls the method with a smaller sublist starting at head.next.
   Step 3) Once we have reached the end of the sublists, we create a sublist and start adding from the back. Since we are currently at the end of the sublist, we will start adding from the last to the first. This is essentially the reverse process taking place.
   Step 4) Once all nodes have been added to the new MLinkedList, we return the list.

2) **private ListNode reverse (ListNode prev, ListNode subhead)**
   Step1) We create a null ListNode h to contain a head, in order to return later.
   Step 2) Check if subhead is null. If it is, we are sorted. This is base case.
   Step 3) If it is not, we assign h as the result of recursively calling reverse with subhead and subhead.next. This calls this function with smaller and smaller sublists.
   Step 4) Once we reach the end of the sublists (1 node), we assign prev to subhead.next, and prev.next to null. This swaps the two nodes, and makes sure the sublist ends with null.
   Step 5) return the h. Note that this does not create a new MyLinkedList, but it does return another node that has a sublist that has been reversed.

3) **public int countSpace (String str)**
   Step1) create a int variable to store the total count.
   Step 2) Check if str.length is 1. If it is, check if whitespace. This is base case
   Step 3) else, recursively call countSpace with one less char from str.
   Step 4) If whitespace, increment the total count variable.

4) **public Boolean myContains(string s1, string s2)**
   Step1) Check if s1.length is 0. If it is, return true.
   Step 2) Check if s1.length > s2.length. If it is, return false.
   Step 3) else, recursively call myContains with s1 and s2 as parameters, but s2 with 1 less character each time.
   Step 4) Once we find a s2 substring that starts with the same char as s1, we check if the following chars in s2 is a match with s1. We create a for loop that starts at 0, ends at s1.length-1. We will check s1.charAt[i] and s2.charAt[i]. If we reach the end of the loop and they are all equal, return true. Else false.

5) **public int div(int m, int n) throws IllegalArguementException**

Step 1) Check if n is 0. If it is, throw illegalargumentexception.
Step 2) Check if n > m. If it is, return 0.
Step 3) Check if n <= m. If it is, recursively call div with m and n+n. The logic behind this is that with each iteration, we are checking how many n's we can fit inside m. For example, 10/5 results in 2 5s in 10. However, if 10/11 were to be called, 0 11s can fit inside 10.
Step 4) Increment counter by 1 everytime that we call div and it does not return 0. Return counter.

6) **private Boolean isSum24(int arr[], int targetSum)**

Step 1) Check if array.length is 0. If it is, return false.
Step 2) Recursively call method isSum24 with the subarray starting at index 1, and the same targetSum.
Step 3) Create a int sum variable, to keep track of the sum.
Step 4) Once we reach the end of the array, we keep on adding the variable to sum.
Step 5) After all the recursive calls, we check if the sum variable is equals to targetSum. If it is, return true. Otherwise, return false.

7) **private void reverseArray(int a[], int low, int high)**

Step 1) Check if low==high. If it is, return. This is base case.
Step 2) recursively call method with low++ and high--. This essentially "moves in" the "pointers" of the array.
Step 3) Create temp int to hold value of high. Swap high with value of low, and low with value of temp value variable.

8)**private void recursiveSelectionSort(int a[], int low, int high)**

Step 1) Check if low==high. If it is, return. This is base case.
Step 2) recursively call method with low++ and high. This essentially means our "pointers" of the array gets smaller. This differs from the previous function as the previous function "shrinks" on both side, but this only shrinks from the low side.
Step 3) Find the lowest value of the sublist, and place that at the beginning of the sublist. This requires a temp variable to store one of the values, similar to previous function.