Gagnaskipan - Data Structures - Final exam spring 2021
In all the assignments you can look at the **"__main__" == __name__** part of the given code as well as the text files ***expected_out.txt*** for clarification of how the program should behave.

<u>This is a home exam, so all data can be used but it is forbidden to use help from any other people, fellow students or not, or to ask specific questions on forums during the exam.</u>

Good luck!

(20%) Multiple choice
These questions are in a separate quiz assignment on Canvas.

(20%) Problem 1: Array operations
You are given the class ArrayList. Implement the operations **is_sorted_asc** and **in_range**. Limitations from previous array assignments apply. In short the only operation you can use directly on the array is the brackets (***[integer_value]***) with a single integer value. Keep in mind you are not allowed to move data to another data structure and back to the list in this problem.

(10%) is_sorted_asc()
- Returns True if all the elements in the arraylist are in ascending order
    - *1,2,3 is ascending, while 3,2,1 is not*
- Otherwise returns False
- An empty arraylist is sorted

(10%) in_range(index_low, index_high, val_low, val_high)
- Checks elements between the indexes index_low and index_high
    - *If index_low: 0 and index_high: 2 check the values at index 0,1 and 2*
    - *This will not be tested for invalid index values*
- Checks if those elements are between the values val_low and val_high
    - *If val_low is 3 and val_high 5 then the values 3,4 and 5 count as in range*
    - *val_low will always be less than or equal to val_high*
- Returns True if all elements in the index range between index_low and index_high are within the value range between val_low and val_high
- Otherwise returns False
- Examples
    - [|0,1,2,3|4,5,6,7,8]
        - Index_low: 0 index_high: 3 val_low: 2 val_high: 4 => False
    - [0,1,2,3|4,5|6,7,8]
        - Index_low: 4 index_high: 5 val_low: 1 val_high: 5 => True

(20%) Problem 2: Recursion in SLL

Implement the following operations using singly linked lists. You must use recursion for **full marks**. **Half marks** for non recursive solutions.

(10%) sum_of_even_or_even_index(head)

- Takes in the head of a sll as a parameter
    - The list only contains numbers
- Returns the sum of all **even value**s and all values with an **even index**
    - *The head of the list has index 0*
        - *0 is even*

(10%) count_smaller_than_prev(head)

- Takes in the head of a sll as a parameter
- Returns the count of values in the list that are smaller than their previous element(before them in the list)

(20%) Problem 3: Trees

You are given an implementation of a general tree in the class ***NumericTree***. Each node has a *numerical value* and can have *any number of children*.

The following operations should be implemented *in the class* ***NumericTree***.

(15%) sum_of_all values()

- Returns the sum of the values of all nodes in the entire tree

(5%) series_exists(number_list)

- Takes in a list of numeric values, ***number_list***, as a parameter
- Returns **True** if there is a way to traverse the tree from the root directly to a leaf so that the values of the nodes on the path are the same values and in the same order as ***number_list***.
    - Otherwise returns **False**

(20%) Problem 4: General programming

Implement the class ***GroupedMembers*** which encapsulates a collection of members. It stores the names of members as well as a group each member belongs to. You can assume member names to be unique but many members can belong to the same group. Each member only belongs to one group.

The following operations should be implemented *in the class* ***GroupedMembers***.

(5%) add_member(group, name)

- Takes the name of a group and the name of a member as parameters
- Adds this member to the collection
    - If the member name already exists, do nothing

(5%) group_list(group)

- Takes the name of a group as a parameter
- Returns a list of the names of all members in a specific group
    - If the group doesn't exist return an empty list

(5%) member_group(name)

- Takes the name of a member as a parameter
- Returns the name of the group that this member belongs to
    - If the member name doesn't exist return ***None***

(5%) other_members_in_group(name)

- Takes the name of a member as a parameter
- Returns a list containing the names of all *other* members in the group that *this member* belongs to
    - If the member name doesn't exist return an empty list

You don't need to implement perfect time complexity for this but points will be deducted from solutions that have exceptionally bad time complexity or don't utilize the available data structures in any way to consider time complexity.