When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q5helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q5solution** module online by **Wednesday**, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on **Wednesday** right after 11:30 pm. In this way you can see my solutions before the Thursday midterm.

Ground Rules: The purpose of your solving problems on this quiz is to learn how to write directly-recursive functions. Try to synthesize your code carefully and deliberately, using the general form of all recursive functions and the 3 proof rules discussed in the notes for synthesizing recursive functions. Remember, it's elephants all the way down: don't think about what happens in the recursive calls. Try to write the minimal amount of code in each function. Code **will** not be easy to debug using the debugger or print statements. Thinking is more effective.

You may not use **for/while** loops nor comprehensions in your code. Do not call the **sorted** function or the **sort** method on lists. Avoid local variables in your code. If you use local variables, each can be **assigned a value only once** in the call, and it **cannot be mutated** (I used such local variables in only **separate** and **sort**). Of course, **do not mutate** any parameters.

1. (4 pts) Define a **recursive** function named **compare**; it is passed two **str** arguments; it returns one of three **str** values: **'<'**, **'='**, or **'>'** which indicates the relationship between the first and second parameter (how these strings would compare in Python). For example **compare('apples','oranges')** returns **'<'** because for Python strings, **'apples' < 'oranges'**. Hint: My solution had 3 base cases that compare the parameter strings to the empty string; the non-base case compares only the **first character** in one string to the **first character** in the other. Your solution must not do much else: it **cannot** use relational operators on the entire strings or slices of strings, which would make the solution trivial; it can use relational operators, **but only on empty strings and single character strings**.

2. (4 pts) Define a **recursive** function named **is_palindrome**; it is passed a **str** argument and returns a **bool** resulting telling whether or not the string reads the same both forwards and backwards, while ignoring the case of all its letters. For example, **is_palindrome('DoGeeseSeeGod')** returns **True**; **is_palindrome('NeverOutOrEven')** returns **False**. Hint: look at the first and last letters.

3. (4 pts) Define a **recursive** function named **separate**; it is passed a predicate and a **list**; it returns a **2-tuple** whose **0** index is a **list** of all the values in the argument **list** for which the predicate returns **True**, and whose **1** index is a **list** of all the values in the argument **list** for which the predicate returns **False**. Calling **separate((lambda x:x>=0),[1,-3,-2,4,0,-1,8])** returns **([1,4,0,8],[-3,-2,-1])**. Actually, the values in the returned **list**s can be in any order, but this order leads to simpler code. Hint: For the non-base case, first call **separate** recursively and bind the result to the returned **2-tuple** (and never change that binding); then look at the first value in the **list**, determine what **2-tuple** to return, using the binding. You can use **+** to concatenate **lists**, but cannot mutate any **list** (e.g., no calls to **append**).

4. (4 pts) Define a **recursive** function named **sort**; it is passed a **list** of comparable values (e.g., all **int**s or all **str**s) as an argument; it returns a new **list** (constructing a new one, not mutating the argument) with every value in the original list, but ordered non-descending. **You must code the following algorithm.**

> For any non-empty argument **list**, use the first value in the **list** to separate the rest of the **list** (call **separate** from question 3) into two **list**s: those with values **<=** the first value and those with values **>** the first value; don't include this first value in either **list**: the sum of the lengths of the two resulting **list**s should be one less than the length of the argument **list**). Note: all the values in the first **list** are strictly **<** all the values in the second **list**.

Recursively call **sort** on each of these two **list**s and use list concatenation to return a single list that contains the sorted values in the first **list** (the smaller ones), followed by the value used to separate the **list**, followed by the sorted values in the second **list** (the larger ones). Again, as in question 3, for the non-base case, first call **separate** and bind the result (and never change that binding); then determine what result to return, using the binding and recursive calls to **sort**.

So, **sort([4,5,3,1,2,7,6])** would call **separate** first to compute the **list**s **[3,1,2]** and **[5,7,6]** (note that **4**, the first value in the **list**, is used to do the separation, and is not in either resulting **list**) . When sorted recursively, these **list**s are **[1,2,3]** and **[5,6,7]**. Concatenating these **list**s, with the separator value between them, results in returning the **list [1,2,3,4,5,6,7]** (the **4** is put in the result **list**, in the correct place). Your function should run to completion with no noticeable delay. Of course you should not use any built-in Python methods for sorting.

5. (4 pts) Define a **recursive** function named **unnested**; it is passed a nested **list** of values: a **list** that can contain a **list** that can contain a **list**, etc. The **unnested** function returns a **list** with **no nesting,** containing all the values in the nested **list**, appearing in the order they appear in the nested **list**. For example

unnested([1,2,3,4,5,6,7,8,9,10]) returns **[1,2,3,4,5,6,7,8,9,10]**
unnested([[1,[2,3],4,5],[[6,7,8],9,10]]) also returns **[1,2,3,4,5,6,7,8,9,10]**

Hints: Use the 3 proof rules to guide your code: think carefully before even testing your code. Use the standard base case for an empty **list**. Any non-empty **list** will have a first value that is either an a **list** or something not a **list**; this first value will be followed by other values in the **list**. You can call **unnested** with any **list** as an argument, and it returns an unnested version of that **list**. You must write this function completely recursively, **without for loops** (even though looping would simplify your code).

6a. (2 pts) Define a **merge** function, whose first argument is a **list** (possibly empty), always containing equal-length strings; its second argument is a single string. It returns a **list**, always containing equal-length strings, that contains all the longest strings from the first and second arguments. For example

merge([],'abc') returns **['abc']**
merge(['abc', 'lmn'],'wxyz') returns **['wxyz']**
merge(['abc', 'lmn'],'xyz') returns **['abc', 'lmn', 'xyz']**
merge(['abc', 'lmn'],'xy') returns **['abc', 'lmn']**

Write the **merge** function using the functional style, with no mutation of **list**s (e.g., no **append**).

6b. (3 pts) Define a function named **all_longest**; it is passed an open file and returns an iterable of all the lines that have the longest length: maybe just one; maybe multiple lines. Ignore any lines that start with the character **#**. You must use **map**/**filter**/**reduce** to solve the problem. Hint: use **merge** (from part 6a) as one of the **lambda**s. Calling **list(all_longest(open('test.txt')))** returns **['cde','opd']** if the file **'test1.txt'** contains the lines

```
ab
cde
ef
#ghijklmn
opq
```

You can write **all_longest** as a pure composition of functions

```
return reduce(...,filter(...,map(...,iterable)
```

or like

```
m = map(...,iterable) # you can bind each name once,
f = filter(...,m)     # but not rebind or mutate it,
r = reduce(...,f)     # as in functional programming
return r
```

Note: **reduce(f,[1,2,3],0)** computes **f(f(f(0,1),2),3)**: it uses **0** (the 3rd argument to **reduce**) for the first reduction with **f**; the 2nd argument in the first reduction is the first value (**1**) from the iterable.