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 Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday morning.

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 is Elephants all the way down. Try to write the minimal amount of code in each function. Some errors will not be easy to debug using the debugger or print statements.

Use **no for** loops or comprehensions (which include **for** loops) in your code. Do not call the **min** or **max** or **sorted** functions or the **sort** method on lists. If you use local variables, each can be **assigned a value only once** in a call, and it **cannot be re-assigned or mutated**; try to use no local variables (except where they are mentioned in hints). Of course, **do not mutate** any parameters.

1. (5 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** (once) recursively and use unpacking to bind the resulting **2-tuple** (and never change either binding); then looking at the first value in the argument **list**, determine what result to return, using the bindings: one index in the **2-tuple** will remain the same, the other will include the first value in the argument **list**. You can use **+** to concatenate two **lists** to create a third, but cannot mutate any **list** (e.g., no calls to **append**). Don't call **separate** twice recursively (it could run appreciably slower).

2. (5 pts) Define a **recursive** function named **is_sorted**; it is passed a **list** of values that can be compared (e.g., all **int** or all **str**) it returns a **bool** telling whether or not the values in the **list** are in non-descending order: lowest to highest, allowing repetitions. For example, **is_sorted([1,1,2,3])** returns **True**; but **is_sorted([1,2,3,1])** returns **False**. Hint: my solution had a base case including any **list** that doesn't have at least two values to compare against each other.

3. (5 pts) Define a **recursive** function named **sort**; it is passed a **list** of comparable values 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 in non-descending order. **You must code the following algorithm.**

For any non-empty argument **list**, separate it (hint: call **separate** from question 1. It is OK to bind the resulting **2-tuple**, but never change that binding) into two **list**s: those with values **<=** the first value in the **list** and those with values **>** the first value; the first value should not appear in the first **list**, unless there are multiple occurrence of it in the argument **list** (said another way, the sum of the lengths of the two **list**s should be one less than the length of the argument **list**). Note that all the values in the first **list** are strictly **<** all the values in the second **list**. Hint:slice.

Recursively call **sort** on each of these two **list**s and 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 1, 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`. You can use `+` to concatenate `lists`, but cannot mutate any `list` (e.g., no calls to `append`).

So, `sort([4,5,3,1,2,7,6])` would call `separate` first, to compute the `lists` `[3,1,2]` and `[5,7,6]` (note that `4`, the first value in the `list`, used to do the separation, is not in either `list`) which when sorted would be `[1,2,3]` and `[5,6,7]`, resulting in the returned `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; you are redefining `sort` here.

4. (5 pts) Define a **recursive** function named `merge_chars`; it is passed two `str` arguments, in which all the **characters appear in alphabetic order**; it returns a `str` that contains all the characters in its argument strings, also **appearing in alphabetic order**. For example, `merge_chars('abxy','lmz')` returns`'ablmxyz'`. Of course you **cannot** just concatenate the strings and `sort` the result (that would make the solution trivial and not recursive); you can compare strings to the empty string and compare **the first character** in one string to **the first character** in the other, and can slice strings and concatenate them.

5. (5 pts) Define a purely **recursive** function (no looping) named `nested_count`; it is passed a nested `list` of `int`s and a single `int` as arguments; it returns the number of times the single `int` argument appears in the nested `list` argument, no matter how far deeply this `int` is nested in the `list`. For example, calling

```
nested_count( [[1,2,[4,[1],8],[1,3,2]],[1,1]], 1 )
```

returns a sum of `5` because there are five `1`s appearing in the nested `list`.

Hints: Use the standard base case for an empty `list`. Any non-empty nested `list` of `int`s will have a first value that is either an `int` or a `list` of nested `int`s (it will not contain `tuple`s, `set`s, `dict`s, etc. only `int`s and nested `list`s of `int`); this first value will be followed by a (possibly empty) nested `list` of `int`s.

You might find an interesting use for calling the `int()` conversion function: note `int(True)` returns `1` and `int(False)` returns `0`.