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; in such cases, use the 3 proof rules.

Use **no `for`** loops or comprehensions (which include `for` loops) in your code. I did not use local variables in any of my functions; 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. Of course, **do not mutate** any parameters. Do not use `try`/`except` statements (avoid them by using `if` statements to check base cases explicitly). Do not call the `sorted` function or the `sort` method on lists. **See Problem 5 for slightly different rules**.

1. (5 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 in Python, `'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, which would make the solution trivial; it can use relational operators, **but only on empty strings and single character strings**. If you can compare arbitrary strings, you could write this function trivially without recursion.

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 `merge`; it is passed two `list` arguments (each is guaranteed to contain the same type of values and each is **sorted** in non-descending order; **don't** bother to check these conditions); it returns a new `list` containing all the values from both argument `list`s, in non-descending order. The call `merge([1,3,5,8,12],[2,3,6,7,10,15])` returns `[1,2,3,3,5,6,7,8,10,12,15]`.

Hints: If either or both `list`s are empty, the problem can be solved without recursion; if both `list`s are non-empty, one of the arguments to the recursive call will be a `list` slice that shorter by one value; the other `list` remains the same length. Of course, you **cannot** just concatenate the `list`s and then call the `sort` method or `sorted` function: the structure of the `merge` function itself must produce the correct ordering. Remember that each argument `list` is already sorted.

4. (5 pts) Define a **recursive** function named `sort`; it is passed any unordered `list` (in which all the values can be compared, e.g., all `int` or all `str`) and it returns a new `list` (not mutating the argument) that contains every value from its argument `list`, but in sorted/non-descending order. You **cannot** call any of Python's functions/methods that perform sorting: the `sort` function itself must produce the correct ordering.

Hints: For any `list` that has at least 2 values, break the `list` in half (using slices, compute the first and second halves of the `list`; each will be smaller than the original `list`); recursively call **`sort`** (the function you are writing here) to sort each smaller `list` (don't worry about how this is done: it's elephants all the way down); then use the merge function, written above, to merge these two sorted **`lists`** returned from these recursive calls; finally, return the merged `list`, which contains all the sorted values in both smaller `list`s. Note that **`merge`** requires its arguments be sorted, which they will be if they are computed by calling **`sort`** recursively.

For example, calling **`sort([4,5,3,1,6,7,2])`** would call **`sort`** recursively on the `list`s `[4,5,3]` and `[1,6,7,2])`, returning the `list`s `[3,4,5]` and `[1,2,6,7]` respectively, which when merged would return the `list [1,2,3,4,5,6,7]`. Note when the length of the `list` is even, both sublists will have the same length; when the length of the `list` is odd, it is natural to divide it into two lists, the first will have one one fewer value than the second.

5. (5 pts) General problem statement: Suppose that you are on a trip and buy a bunch of gifts to bring home to your family. When you arrive at the airport for your trip home, you discover that you may carry only a limited amount of weight on the plane. Determine the maximum value of gifts you can bring home while staying within the weight limit.

Python problem statement: Define a **recursive** function named **`max_value`**; it is passed two arguments: (1) a **`tuple`** of **`2-tuples`**: each **`2-tuple`** contains an **`int`** (representing the **weight** of a **gift**) followed by an **`int`** (representing the **value** of that same **gift**); (2) the maximum amount of weight allowed. The **`max_value`** function computes the maximum value of gifts you can bring home while not exceeding the weight limit. It doesn't compute **which gifts** (a similar but harder problem), but instead computes just the **value of the gifts**.

You might think you can solve this problem iteratively, by sorting the **`2-tuples`** either by decreasing **value** or increasing **weight** of the **gifts**, and then iterating through the list choosing which ones to bring. But that won't work. For example, calling **`max_value(((10,70),(15,80),(20,140),(20,150),(30,200)), 50)`** would should return the value **`360`** (1st, 3rd, and 4th gift): we cannot choose gifts from the left so long as their weight is not too big, nor choose gifts from the right so long as their weight is not too big, to get the optimal value. But there is a "simple" recursive solution to this problem.

Hint: Use the 3 proof rules discussed in the notes. Better to stare at your code while thinking hard about these rules than to try to debug the recursive code: it's Elephants all the way down. Study the reasoning used in the notes to solve the **minimum number of stamps** (**`mns`**) problem, which is similar to this one. Try to replicate that reasoning for this problem. Here, the basic recursive structure decides whether or not to include a gift (computing **`max_value`** for both cases); of course, weight limits may eliminate the possibility of including some gifts. Note that the **`tuple`** will **get smaller** for each recursive call (one fewer gift to consider whether to include it or not); the **`int`** (weight) can also **get smaller**, in those cases where the gift will be included in the calculation; compute your base case appropriately.

Finally, although the ground rules stated at the top of this quiz allow for **no local variables**, in this function you may write two: **`w,v = gifts[0]`** to simplify your code: storing the **weight** and **value** of the first **gift** in the **`tuple`**; but you **may not rebind `w`** or **`v`**. If course you could always not create these bindings and just write **`gifts[0][0]`** and **`gifts[0][1]`** to refer to these two values, but the resulting code would be harder to read/understand/write.