Name _____

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.

---

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**. The call **separate(predicate.is_positive,[1,-3,-2,4,0,-1,8])** returns **([1,4,8], [-3,-2,0,-1])**. Note **0** is not positive. Hint: like the fast version of the **power** function in the notes, you can define and bind (but not rebind) a local name or can write a nested function (like **square** in **power**) to help with the computation. You must call **separate** recursively only once (but you can save the values it returns).

2. (5 pts) Define a **recursive** function named **is_sorted**; it is passed a **list**; 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**.

3. (5 pts) Define a **recursive** function named **sort**; it is passed a **list**; it returns a new **list** (not mutating the passed one) with every value in the original list, but ordered in non-descending order. Hint: code the following algorithm.

For any non-empty argument list, separate it (use **separate** from question 1) into two lists: 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 lists should be one less than the length of the argument list). Note that all the values in the first list are **<** all the values in the second list. Recursively call **sort** on each of these two lists 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. Again, as in question 1, you can define and bind (but not rebind) local names or can write a nested function (like **square** in **power**) to help with the computation.

So, **sort([4,5,3,1,2,7,6])** would call **separate** to compute the lists **[3,1,2]** and **[5,7,6]** (note no **4**) 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).

4. (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 they would compare). Hint: my solution compared strings to the empty string and compared **one character** in one string to **one character** in the other. Your solution must not do much else : it **cannot** use relational operators on the entire strings, which would make the problem trivial; it can use relational operators only on characters.

5. (5 pts) Write a function named **peaks**, which is passed an iterable (that produces numeric values). This function returns a list of **int** for those values in the iterable that are bigger than the value preceding and following them: **peaks([0,1,-1,3,8,4,3,5,4,3,8])** returns **[1,8,5]**. This result means the values **1**, **8**, and **5** are **strictly bigger** than the value preceding and following them.

For this solution, you **must** solve it using the **functional programming** style, using the functionals **map**, **filter**, and **reduce** (and one "large" function and two small **lambdas** that you write to pass to these functionals). The form of your function will be **map** on a **filter** on a **reduce** of the iterable. Write your function in the following form:

```
def peaks(iterable):
    reduced   = reduce(triple, iterable, [[]])#reduce produces a list of triples: note []
    filtered  = filter(lambda ..., reduced)   #filter out triples not representing peaks
    mapped    = map(lambda ..., filtered)     #map triple to its middle value
    return list(mapped)                       #map is a generator; store its values in list
```

Define **triple** as a function (not a **lambda**). Calling **reduce** with **triple** converts all the iterable values into overlapping triples. Suppose the iterable produces the value **a**, **b**, **c**, **d**, and **e**. By calling **reduce** with **triple** and the unit **[[]]**, it means that **triple** will first be called with the arguments **[[]]** and **a**; then **triple** is called with its result, **[[a]]** and **b**; then **triple** is called with its result **[[a,b]]** and **c**, producing **[[a,b,c]]**.

triple([[]],a) returns **[[a]]**
triple([[a]],b) returns **[[a,b]]**
triple([[a,b]],c) returns **[[a,b,c]]**

Above, when **triple**'s first argument is a **list** with only one **list** in it (and that inner **list** contains 0, 1 or 2 values), **triple** just puts the second argument at the end of the **list** in the argument **list**. In this way, it is building the first **3-list** with the first three values in iterable.

For all subsequent arguments, **triple** returns a **list** of all the current **3-lists** (its left argument) with a new **3-list** concatenated at its end: the new **3-list** starts with the last two values of the **3-list** previously at the end, and its third value is the value supplied as the second argument. Here are three more examples

triple([[a,b,c]],d) returns **[[a,b,c],[b,c,d]]**

triple[[a,b,c],[b,c,d]],e) returns **[[a,b,c],[b,c,d],[c,d,e]]**

triple[[a,b,c],[b,c,d],[c,d,e]],f) returns **[[a,b,c],[b,c,d],[c,d,e],[d,e,f]]**

Test your **triple** function first, to verify it returns correct lists. Calling **reduce** actually returns a bigger list.

Finally, to complete the **peaks** function, **filter**'s **lambda** filters out triples whose middle value is not a peak; **map**'s **lambda** selects the middle value from each triple, resulting in a **list** of those middle values greater than their surrounding values.