Assignment 3, Due 4/28

Do at least four of the following problems.

Problem 1

Write a function `remove_duplicates` that takes as input a list, and returns a new list which contains exactly one copy of each entry in the input list. Here's an example:

```
>>> remove_duplicates([3,3,3,3])
[3]
>>> remove_duplicates([3, [3]])
[3, [3]]
>>> remove_duplicates([3, [3], [3]])
[3, [3]]
```

As a side note, you may find the `in` operator useful:

```
>>> 3 in [3]
True
>>> [3] in [3]
False
>>> [3] in [[3]]
True
```

Problem 2

This one is a new spin on a problem from last week. Take your solution from `print_table` on last week's homework and create an `@interact` that has input fields for a list and a function, and prints a nice table (just as in last week's homework). If that's too easy, create something that will print the table, and then print out correct HTML code for typesetting that table (so that you could copy-paste it into a webpage). You might find sage's `html.table` command helpful for this part.

Problem 3

Create a function that takes a positive integer `n` as input, and returns a list containing all binary trees with exactly `n` leaves.

You can approach this two ways:

- you could try to "roll your own" code, which simply generates the list of all trees, or
- dig into the Sage documentation, and find out how to use Sage to do this. (Hint: you'll want to start by looking into the combinatorics module, which is located in `sage.combinat`.)

If you're going with the "roll your own" approach, it might help to realize that there's a one-to-one correspondence between binary trees with `n` leaves and ways of parenthesizing the expression $x_1 \times x_2 \times \cdots \times x_n$. This

means that you could represent each tree as a *really* nested tuple: for instance, letting $(((x_1x_2)(x_3(x_4x_5)))x_6)$ correspond to `(((1, 2), (3, (4, 5))), 6)`.

You can represent the trees in any way you'd like, as long as you document it. Make sure that your output is complete and contains no duplicates. If you're curious, the number of such trees is known as the $\texttt{n}^{\text{th}}$ Catalan number, which you can find out more about on the Wikipedia article.

Problem 4

You may have seen the amusing fact that $(1 + 2 + \ldots + n)^2 = 1^3 + 2^3 + \ldots + n^3$. One can prove this several ways, with induction being the most common. (There's also an amusing visual proof.) However, let's say you saw this for the first few values of `n`, and wanted to check out more. Write a program which takes as input a positive integer `n` and verifies this fact for every positive integer up to and including `n`. (That is, compute both sides and check that they match.) You should print some output as you go — but printing something for every single integer would probably be overkill. Try to print a manageable amount of data.

Problem 5

This is a two-part problem, to design a class and use it as a decorator.

First, we'll make a new class called `file_logger`. We want to be able to use this as a decorator, as follows:

```
>>> my_logger = file_logger("some_filename")
>>> @my_logger
... def f(n, m):
...     do_stuff()
...     return something_interesting
...
```

Now every call to `f` will act just like normal — except that it will log all inputs and outputs to the file `some_filename`.

This should be implemented as a Python class with at least two methods: `__init__` and `__call__`. The `__init__` method should take two arguments: `self` (which methods on a class almost always take), and `filename`, which is the file where it should log inputs and outputs. The `__call__` method should take a single argument, a function, and should return a new function which takes the same arguments (think `*args, **kwds`) and logs all inputs and outputs to the file `filename`.