I. Administrivia

- Pset 4 due 5pm today
- Fill out form if you have a conflict with the final exam
- Final **Wednesday 10/16** in class, review session **Wednesday 10/9** in class

II. Recursion

- A recursive function is a function that calls itself until a “base condition” is true
- Example: Recursive list-flattening
  - You are given a list containing elements which are either (1) ints or (2) lists meeting the same description (i.e. lists containing elements which are either ints or lists...point out the recursion)
  - You want to remove all internal lists and instead have all of their elements contained in the base-level list.
  - Example: [1, [2, 3], [4, [5, 6]], 7] should give us [1, 2, 3, 4, 5, 6, 7]

```python
def flatten(L):
    flat_L = []
    for e in L:
        if type(e) == list:
            flat_L.extend(flatten(e))
        else:
            flat_L.append(e)
    return flat_L
```

III. Classes & Inheritance

- Parent and Child classes / Superclass and Subclasses: Usually subclass is a special case or particular type of the superclass

Example:

- Parent class, Pet, that other animals inherit from. Contains attributes and methods that all pets share
- Child class, Cat
  - Inherits __init__, get_name, get_species, __str__ from Pet
  - Overrides make_sound
  - The child class automatically inherits **ALL** methods and attributes from the parent, unless redefined in the child class, in which case they “override” the parent class definitions
  - Adds new Cat-specific method, purr
- Child class, Dog
  - Calls parent __init__ method and then adds additional attribute
● Overrides make_sound
● Adds new Dog-specific method, chase

- Child class, CatDog
  - Inherits from both Cat and Dog, cat first. Which sound will it make?

- Child class, DogCat
  - Inherits from both Dog and Cat, dog first. Which sound will it make?

IV. Complexity/Order of Growth

- “Big-O” notation
- Gives us an idea of how long an algorithm will take to run with respect to the size of its inputs (arguments), regardless of what machine it’s running on.
- Gives the worst case scenario
- We don’t care about lower-order terms or constants. We are interested in trend as input grows very large, so highest order terms dominate.
  - Adding:
    - $O(n^2) + O(n) + O(1) \rightarrow O(n^2)$
  - Multiplicative or additive constants don’t matter
    - $O(10*n) \rightarrow O(n)$
    - $O(\log_2(n)) \rightarrow O(\log(n)/\log(2)) \rightarrow O(\log(n))$
    - $O(n + 1) \rightarrow O(n)$
  - We want the tightest bound possible: technically, an algorithm that is $O(n)$ is also $O(n^2)$, $O(2^n)$, etc, since $O$ is just $\leq$ but we want closest upper bound possible.

Common orders of growth
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Time</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>constant</td>
<td>Adding two numbers together, appending to a list - Independent of input size!</td>
</tr>
<tr>
<td>O(log (n))</td>
<td>logarithmic</td>
<td>Binary search</td>
</tr>
<tr>
<td>O(n)</td>
<td>linear</td>
<td>list.copy(), or scanning through an entire list to look for a value</td>
</tr>
<tr>
<td>O(n log(n))</td>
<td>log-linear</td>
<td>merge-sort</td>
</tr>
<tr>
<td>O(n^2), O(n^3), etc.</td>
<td>polynomial</td>
<td>nested for loops</td>
</tr>
<tr>
<td>O(2^n), O(3^n), etc.</td>
<td>exponential</td>
<td>Trying to guess an n-character password - Bad!</td>
</tr>
</tbody>
</table>

Complexity of Python methods/objects [https://wiki.python.org/moin/TimeComplexity](https://wiki.python.org/moin/TimeComplexity)

- **Constant-time operations:**
  - Assigning a variable (x = 1)
  - Performing basic operations (+, -, /, **, <, >, ==, etc.)
  - Some built-in methods for data structures in Python are also constant-time, but many are not. Although we don’t look at the underlying machinery of many built-in methods, the complexity of their implementations affects the complexity analysis of our own methods.

- **Dictionaries (will depend on hash function)**
  - O(1): lookup, checking if key in dictionary, length, insert, delete
  - O(n): d.keys() or d.values() - list of length n must be generated

- **Lists**
  - O(1): append, length
  - O(n): insert, delete (move around elements), copy, check if item in (unsorted) list
  - O(n log n): sort

**Strategies for Order-of-Growth Analysis**

- **Loops:** # of iterations times cost of each iteration
- **Recursive functions**
  - How many recursive calls are being made?
  - How much work does each recursive call take?
  - Draw a tree connecting subproblems
V. Search and Logarithmic Complexity

- Naive searching strategy: linear scan of the list.
  - Brute force, takes $O(n)$
  - in the worst case, we have to look at every value.
- Faster: **binary search**, very similar to bisection search on PS1
  - Pick an index $i$ that is at the middle of the input list, $L$
  - If $L[i]$ is $<$ the number we’re looking for, we want right half of list
  - If $L[i]$ is $=$ number we’re looking for, we’re done
  - If $L[i]$ is $>$ the number we’re looking for, we want left half of list
- The list must be sorted, so that we can decide which half to choose
- Implemented iteratively in PS1, but can also do recursively
- How long does this take?
  - Number of iterations: How many times can we divide a list/range of numbers of length $n$ in half? $\rightarrow \log$ base 2 of $n$, which simplifies to $\log(n)$
  - Each step: constant time
  - So overall $O(\log n)$