Admin:

http://stellar.mit.edu/S/course/6/fa13/6.006

Pset #1 out today. Due 9/17.

See TA if you need to switch recitation sections.

Reading: CLRS Chapters 1-3.

Outline:

☐ Defns & terminology

☐ Python cost model

☐ Document Distance Problem
Key Notions:

Computational problem = mapping from inputs to outputs

may be one-to-many if more than one answer is possible (e.g., peak-finding problem), but typically one-to-one.

Input space = set of possible inputs

Input instance = a particular input (aka a problem instance)

Output space = set of possible outputs

\[ f(x) = y \]

\( x \) may be a single number, or a complex data structure (e.g., a string, array, graph, ...)

\[ x \]
Algorithm = well-defined computational procedure
mapping inputs to outputs

In practice, inputs & outputs are represented
concretely as data structures of some sort,
& a algorithm is realized as a program.
We'll move fluidly back & forth between these
levels of abstraction...

Model of computation:
Specifies how data is represented, & how
computation may be performed.

Examples: Circuit model
Turing machine model
RAM (random access machine) model
Python model

Also describes cost of resources used
(e.g. time, memory, randomness, wires,...)
(aka "cost model")
In this course, we care about (asymptotic) efficiency of algorithms: how do resources used scale with size of input? 

Size of input: Needs to be defined for each problem, although there are common patterns.

Examples: length of input array, number of bits used to represent input (e.g., for large-precision integers).

Number of vertices & edges in a graph.

Note: for $n \times n$ input matrix may call input size "$n" even though there are $n^2$ entries in matrix.

Typically interested in worst-case running time (call it, say, $T(n)$ for inputs of size $n$).

Would like result of form $T(n) = \Theta(f(n))$

for some nice function $f$. $\Theta$ hides constant factors & expresses asymptotic equivalence.
Python cost model (examples)

**Lists (aka “arrays”)**

\[ L = [a_1, a_2, \ldots, a_n] \]

- \( L[i] = x \) \hspace{1cm} \text{time } \Theta(1)
- \( L.\text{append}(x) \) \hspace{1cm} \text{time } \Theta(1)

*(ref “table doubling” lecture, to come)*

- \( L_1.\text{extend}(L_2) \) \hspace{1cm} \text{time } \Theta(|L_2|)
- \( L_1 + L_2 \) \hspace{1cm} \text{time } \Theta(|L_1| + |L_2|)

**Dictionaries (aka “hash tables”)**

\[ D = \{ x_1 : y_1, x_2 : y_2, \ldots, x_n : y_n \} \]

- \( D[x_i] = y_i \) \hspace{1cm} \text{time } \Theta(1)
- \( x \text{ in } D \) \hspace{1cm} \text{time } \Theta(1)
Document Distance Problem

How similar are two documents D1 & D2?

- Information retrieval/search
- Who wrote Shakespeare's plays?
- Plagiarism detector?

Not really a hard problem, but useful to illustrate many concepts of this course: data structures, sorting, hashing, asymptotic analysis.

Definitions:

- document = sequence of characters
- word = sequence of alphanumeric characters

so document = sequence of words, separated by whitespace & punctuation

(note: we ignore case)

We'll ignore word order, & view a document as equivalent to its set of word frequency counts.
Can view document as a vector, where coordinates are words:

\[ D_1 = \text{"to be or not to be"} \]
\[ = \text{to be or not} \]
\[ = (2, 2, 1, 1) \]
\[ D_2 = \text{"it was not to be"} \]
\[ = \text{to be or not it was} \]
\[ = (1, 1, 0, 1, 1, 1) \]

Length of vector \( \|D\| = \sqrt{\langle D \cdot D \rangle} \)

Measure similarity by angle between vectors:

\[ \Theta = \arccos \left( \frac{D_1 \cdot D_2}{\|D_1\| \cdot \|D_2\|} \right) \]

\[ 0 \leq \Theta \leq \frac{\pi}{2} \]

Document distance problem is:

**Given:** \( D_1, D_2 \)

**Compute:** \( \Theta \)
## Data Sets:

<table>
<thead>
<tr>
<th>Data Set</th>
<th>(from Project Gutenberg)</th>
<th>(bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Jules Verne</td>
<td>&quot;2889&quot;</td>
<td>25 K</td>
</tr>
<tr>
<td>2 Bobse Twins</td>
<td></td>
<td>268 K</td>
</tr>
<tr>
<td>3 Lewis &amp; Clark</td>
<td></td>
<td>1 M</td>
</tr>
<tr>
<td>4 Shakespeare</td>
<td></td>
<td>5.6 M</td>
</tr>
<tr>
<td>5 Bacon</td>
<td></td>
<td>320 K</td>
</tr>
</tbody>
</table>

## Different versions of program:

<table>
<thead>
<tr>
<th>Version</th>
<th>(new laptop)</th>
<th>(old laptop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Initial</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2 With profiling</td>
<td>38</td>
<td>194 secs</td>
</tr>
<tr>
<td>3 wordlist, extend</td>
<td>19</td>
<td>84 secs</td>
</tr>
<tr>
<td>4 use dictionaries</td>
<td>5.2</td>
<td>41 secs</td>
</tr>
<tr>
<td>5 translate</td>
<td>5.1</td>
<td>13 secs</td>
</tr>
<tr>
<td>6 merge-sort</td>
<td>0.3</td>
<td>6 secs</td>
</tr>
<tr>
<td>7 no lists</td>
<td>0.11</td>
<td>—</td>
</tr>
<tr>
<td>8 treat file as one line</td>
<td>0.08</td>
<td>—</td>
</tr>
</tbody>
</table>

- Other optimizations possible (e.g. use of "pypy", which doesn't do much here, but can get ~7x speedup on some problems...)
Review change from 2/docdist2.py to 3/docdist3.py

2:

```python
def get_words_from_line_list(L):
    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list = word_list + words_in_line
    return word_list
```

3:

```python
word_list.extend(words_in_line)
```

For 2, if there are \( n \) lines & 1 word per line

\[
time \propto 1 + 2 + \ldots + n = \Theta(n^2)
\]

For 3,

\[
time \propto \Theta(n)
\]

Changing algorithm from \( \Theta(n^2) \) to \( \Theta(n \log n) \) or even \( \Theta(n) \) doubles other constant-factor improvements. (consider \( n = 10^6 \).)
Three fixes removed $O(n^2)$ time takes:

1. $\rightarrow$ 2: gathering words in document
   append rather than concatenate

2. $\rightarrow$ 3: use dictionaries rather than lists for counts

   in 3: $\left[ \left[ \text{"the"}, 2 \right], \left[ \text{"cat"}, 1 \right], \ldots \right]$ list

   in 4: $D[\text{"the"] = 2$ dict

3. $\rightarrow$ 4: use merge-sort rather than insertion sort to
   produce sorted list of words & count
   (preparing to doing inner product)

Then running time is $\Theta(n \log(n))$ [due to merge sort].

4. $\rightarrow$ 5: Doing inner product directly on dicts, rather
   than using sorted lists, removes need to sort!

   def inner_product(D1, D2):
     sum = 0
     for k in D1:
       if k in D2:
         sum = sum + D1[k] * D2[k]
     return sum

   (Python supports iteration over keys in a dict.)

Then running time is $\Theta(n)$!