(turn cellphone off!)

Outline:
- Administrivia
- Course overview
- "Document distance" problem

Handouts:
1. Course info
2. Sign-up sheet (passed around)
3. docdist1.py

Administrivia:
- Welcome to 6.006!
- Introduce stuff
- Sign-up - on line
  and - on sheet being passed around
- web = http://courses.csail.mit.edu/6.006
- Ask students to raise hand(s) for:
  - credit/listen/unsure
  - fresh/soph/...
  - MIT/not
  - course 6 / 18 other
  - python
  - 6.01
  - 6.042
- Pre-reqs: see TA's if haven't got them, but have ☐
- Recitations: on-line; get assignment tonight - bring laptops!
  - lectures/recitations/problem sets / 2 quizzes / final
  - textbook: CLRS, rec: Miller/Raum
- relation to 6.046
Course Overview

- Efficient procedures for solving problems on large inputs
- Scalability (now can have & will use complete works of Shakespeare, human DNA, or U.S. highway map on your laptop)
- Classic data structures & elementary algorithms (CLRS)
- Real implementations (Python)
- Having some fun (problem sets!)
- Developing this course:
  - This is only 1st version - will have rough edges
  - We want your feedback - think of yourselves as "co-designers"

Contents:

- 7 modules, each with motivating problem & problem set (except list)
  - Intro & linked data structures: Document Distance Set Ops
  - Hashing
  - Dynamic Programming
    - Insert Replacing
  - Sorting
    - Heapsort
    - Gas Simulation
  - Search
    - BFS/DFS
    - 2x2 Rubik
    - Cal Tech → MIT
  - Shortest Paths
  - Numerics
    - Least Squares
**Document Distance Problem** (Document Similarity)

- Given two "documents" how similar are they? common problem
  - identical - easy?
  - modified or related (DNA, plagiarism, authorship?)
  - Did Francis Bacon write Shakespeare's plays?
  - Need to define metric
- define word = sequence of alphabets/numerics "6,000 is fun" 4 words
- word frequencies: $D(w) = \frac{\text{times } w \text{ occurs in document } D}{\text{possible}}$ 
  can think of frequency as a vector, each word is a coordinate
  count $[1, 0, 1, 1, 0, 1]$ 
  $w$: 6,000, easy, fun
  for some canonical ordering of all words...

\[
\vec{D}_1 \cdot \vec{D}_2 = \sum_w D_1(w) \cdot D_2(w) \quad \text{inner product}
\]

\[
\|D\| = N(D) = \sqrt{\vec{D} \cdot \vec{D}} \quad \text{Hypo norm}
\]

\[
\Theta = \arccos \left( \frac{\vec{D}_1 \cdot \vec{D}_2}{\|D_1\| \cdot \|D_2\|} \right) = \Theta(D_1, D_2)
\]

$0 \leq \Theta \leq \pi/2$

$\uparrow$

identical

$\uparrow$

no common words

Problem: given $\vec{D}_1, \vec{D}_2$ compute $\Theta$ (angle between their word frequency vectors)
### Data Sets:

<table>
<thead>
<tr>
<th>Author/Title</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jules Verne &quot;2889&quot;</td>
<td>25K</td>
</tr>
<tr>
<td>Bobry Twins</td>
<td>268K</td>
</tr>
<tr>
<td>Lewis &amp; Clark</td>
<td>1M</td>
</tr>
<tr>
<td>Arabian Nights</td>
<td>3M</td>
</tr>
<tr>
<td>Churchill</td>
<td>10M</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>5.5M</td>
</tr>
<tr>
<td>Bacon</td>
<td>320K</td>
</tr>
</tbody>
</table>

### Procedure (document):

1. Read file
2. Make word list
   - ["the", "year", "2889", "by", "Jules", ... ]
3. Count frequencies
   - [["the", 4012], ["year", 55], ... ]
4. Sort into order
   - [["a", 3120], ["after", 17], ... ]
5. Compute angle = \( \arccos \left( \frac{D_1 \cdot D_2}{\|D_1\| \cdot \|D_2\|} \right) \)

### Code:

- Students should be able to read this code (ask TA if not, in recit.)
- Go through routines (admittedly, not most efficient, but correct.)

### Experiment:

Bobsey vs Lewis \( \theta = 0.574 \) (3 minutes)

Doesn't seem to scale well, just "dies" on bigger files... (too long)

What is \( \Theta(n^2) \)?

### Python vs C?

- (More at program size) \( \Theta(n^2) \Rightarrow \Theta(n) \)
- \( \times 10 \) or so only
- \( 10^3 \) or more speed up
Profiling:
- How much time is spent in each routine:
  \[ \frac{\text{total execution time}}{\text{import profile profile, run ("main()")}} \]

1. \# calls
2. \text{tottime} - exclusive of subroutine calls
3. \text{percall} = \frac{2}{0}
4. \text{cum} - including subroutine calls
5. \text{percall} = \frac{4}{0}

Results: Bob vs. Lewis
- 194 seconds total - 3 minutes!
- 107 in \text{get words from line list}
- 44 in \text{count_frequency}
- 13 in \text{get word from string}
- 12 in \text{invariant sort}

\[[\text{get words from line list}(L)]:\]
\[
\text{word_list = [ ]}
\]
\[
\text{for line in L:}
\]
\[
\text{words_in_line = get words from string (line)}
\]
\[
\text{word_list = word_list + words_in_line}
\]
\[
\text{return word_list}
\]

\[\uparrow\]

\[\text{has to be this! (there isn't anything else here!)}\]
List Concatenation:

\[ L = L_1 + L_2 \]

takes time proportional to \(|L_1| + |L_2|\).

if we had \(n\) lines, each with one word

time proportional to \(1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)\)

Solution:

\[
\text{word_list}.\text{extend}([\text{words_in_line}]) \quad \text{time proportional to } \text{len(Word_in_line)}
\]

\[
\left[ \text{word_list}.\text{append}(\text{word}) \right] \quad \text{for each word in words_in_line}
\]

Python has powerful primitives built-in.

To write efficient algorithms, we need to understand their costs.

(Figuring out cost at set ops will be their homework...)
Document Distance ta.bobsey.txt 263 KB
e3. lewis.txt 1 MB

Version 1: initial
2: add profiling
3: wordlist.extend(words_in_line)
4: use dictionaries in count_frequency
5: translate
6: merge_sort

with doclist3:

def count_frequency(word_list):
    L = []
    for new_word in word_list:
        for entry in L:
            if entry[0] == new_word:
                entry[1] = entry[1] + 1
                break
        else:
            L.append([new_word, 1])
    return L

try: from doclist3 import *
    L = read_file("ta.bobsey.txt")
    W = get_words_from_line_list(L)
    F = count_frequency(W)

Analysis: time = \( O(n \cdot d) \)
if all words distinct, \( d = n \)
\( \) time = \( \Theta(n^2) \)

\[ \]

\[ \]

\[ \]

\[ \]
Dictionaries - Hash Tables

mapping from domain (finite collection of immutable things) to range (anything)

\[ D = \emptyset \]  
\[ D[\textquotesingle ab\textquotesingle] = 2 \]  
\[ D[\textquotesingle the\textquotesingle] = 3 \] fast!

\[ D[\textquotesingle ab\textquotesingle] \] fast!
\[ D[\textquotesingle xyz\textquotesingle] \] error
\[ D.\text{has}\_\text{key}(\textquotesingle xyz\textquotesingle) \]
\[ D.\text{items}() \]
\[ D.\text{keys}() \]

show doceilsh counter-frequency
cuts time in 1/2 (more for large files) \((8s \Rightarrow 4s)\)
Analysis: time \(\Theta(n)\) doesn't depend on what's already in table

Remaining time goes to:

get_words_from_string \((25\times\text{times with translate})\) 13 secs
insertion_sort \((26\times\text{times with merge-sort})\) 11 secs

Important to understand costs (running times) of Python primitives!
See Python Cost Model (web site) for some experimentation...
(You'll mimic this for Python set operations...) HW #1