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 staff
- Sign-up - on line (see announcements)
  - on sheet being passed around
  - web = http://courses.csail.mit.edu/6.006 (Spring 2009)
- Ask students to raise hands for:
  - credit/ether/unsure
  - fres/soph...
  - MIT/not
  - course 6/19 other
  - python
  - 6.01
  - 6.042

- Pre-regs: see TA's if haven't got them, but have C
- Recitations: on-line; get assignment tonight
- Lectures/recitation/puzzle sets/2 quizzes/final (3/11, 4/15)
- Textbook: CLRS, rec: Miller/Raeum
- relation to 6.046
Course Overview

- Efficient procedures for solving problems on large inputs
- Scalability (now can have & we'll 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: CAE: Now iteration #9!
  - This is only beta version - will have rough edges
  - We want your feedback - think of yourselves as 'co-designers'.

Sample Content:

- 7 modules, each with matching problem & problem set (except last)
  - Intro & linked data structures: Document Distance
  - Sorting
  - Search
  - Shortest Paths
  - Numerics

Diagram:

- Dynamic Programming
- String Matching
- Greedy
- Graph Theory
- Linear Algebra
- Numerics

Other topics:
- Chimp DNA
- Chimp's DNA
- Gas Simulation
Document Distance Problem (Document Similarity)

- Given two "documents" how similar are they? common problem
  - identical - em?
  - modified or related (DNA, plagiarism, authorship)
  - Did Francis Bacon write Shakespeare's plays?
  - Need to define metric
- define word = sequence of alphabemes "6,000 is fun" 4 words
- word frequencies: \( D(w) = \) times \( w \) occurs in document \( D \)
  - can think of frequency as a vector, each word is a coordinate
  - \( \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \)
  - \( w \), the, is, 6,000, easy, fun
  - for some canonical ordering of \( D \) words...

\[ D_1 \cdot D_2 = \sum_w D_1(w) \cdot D_2 (w) \] inner product

\[ |D| = N(D) = \sqrt{D \cdot D} \] length, norm

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

\(0 \leq \theta \leq \pi/2\)

\[ \begin{array}{ll}
\uparrow & \text{identical} \\
\uparrow & \text{no common words}
\end{array} \]

Problem: given \( D_1, D_2 \) compute \( \theta \) (angle between their word frequency vectors)

Open question if this distance is a "good measure or not. You might think why/why not

AE: or no \( \arccos \) giving \( 0 \leq \theta \leq 1 \)
Data sets:
- Jules Verne "2889" 25k
- Bobby Twins 268k
- Lewis & Clark 1M
- Arabian Nights 3M
- Churchill 10M
- Shakespeare 5.5M
- Bacon 320k

Procedure (details):

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

Look at Python 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 \( n^2 \)?

Discuss [Python vs C? (where at program, large) \( x \times 10 \) or so only (AE:)]

Choice & algorithm \( \Theta(n^2) \Rightarrow \Theta(n) \) \( 10^3 \) or more speedup

AE: these days complicated software/hardware issues also relevant but not focus of this class
Profiling:
- How much time is spent in each routine:
  \[ \text{time} = \text{exec_time} + \text{sub_time} \]

1. \# calls
2. tot time - exclusive of subroutine calls
3. procall = \#1 / \#0
4. cum = including subroutine calls
5. procall = \#0 / \#0

Results:
- Bobon vs Lewis
  - 194 seconds total = 3 minutes!
  - 107 in get_words from line list
  - 44 in count_frequency
  - 13 in get_word from string
  - 12 in intra-sort

\[
\text{get_words_from_line_list}(L):
\begin{align*}
\text{word_list} &= \text{[]} \\
\text{for line in } L: \\
\text{words_in_line} &= \text{get_words_from_string(line)} \\
\text{word_list} &= \text{word_list + words_in_line}
\end{align*}
\]

return word_list

!?

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

```python
word_list, extend (words_in_line) <- time proportional to len(words_in_line)

[word_list, append (word)] 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 and analyzing costs will be their homework... )
from defaultdict import *

W = get_words_from_line_list(L)
F = count_frequency(W)
L = read_file("a3bobby.txt")

if all(words distinct, don't print)
    time = O(n^d)
    j words

\text{for word in word-list}
\text{for new-word in word-list}
\text{if entry in L:}
    \text{entry}[w] = entry[w]+1
\text{else: break}

\text{else: break}

L = append(L, \text{word[",", i]})

\text{def count-frequency(word-list):
    L = []
    \text{for entry in word-list:
        \text{if entry in L:}
            \text{entry}[w] = entry[w]+1
        \text{else: break}
    \text{L = append(L, \text{word[",", i]})}
}

\text{def main():
    \text{word-list = read_file("words-in-line")
    \text{word-list = read_file("words-in-line")
    \text{use defaultdict to count-frequency}}

\text{t3: bobby.txt 243KB}
\text{t4: lewis.txt 2MB}
\text{t5: random.txt 9MB}
\text{t6: million-words.txt 1GB}
\text{t7: billion-words.txt 2GB}
\text{t8: trillion-words.txt 4GB}
\text{t9: petaword-words.txt 12GB}
\text{t10: exaword-words.txt 27GB}
Dictionaries - Hash Tables

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

\[ D = \emptyset \] empty mapping
\[ D['ab'] = 2 \]
\[ D['the'] = 3 \] fast!
\[ D \]
\[ D['ab'] \] fast!
\[ D['xyz'] \] error
\[ D . has_key('xyz') \]
\[ D . items () \]
\[ D . keys () \]

show docdists 4 count_frequency

cuts time in 1/2 (more for larger files) (84 secs \(\rightarrow\) 41 secs)

Analysis: time \(\Theta(n)\) doesn't depend on what's already in table

Remaining time goes to:

get_words_from_string (VS times with Tron lite) 13 secs
insertion_sort (VS 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