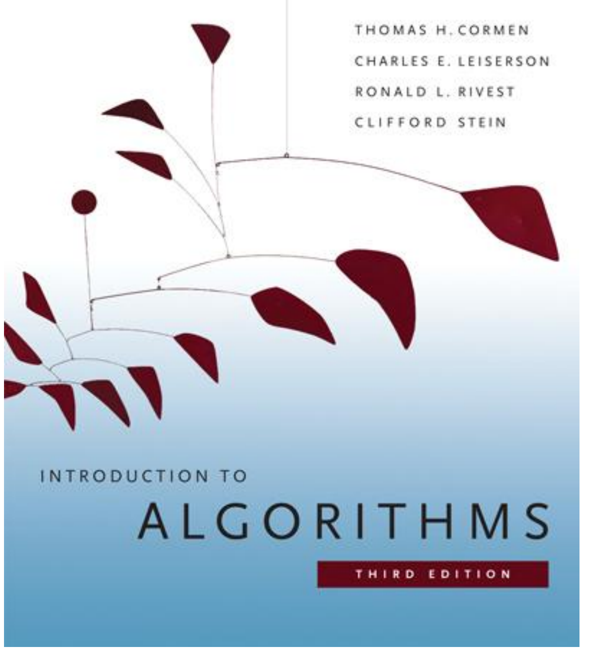
6.006- *Introduction to Algorithms*



Lecture 1

Prof. Constantinos Daskalakis

Today’s Menu

- Motivation
- Administrivia
- Course Overview
- Linked Lists and Document Distance
- Intro to “Peak Finding”

“Al-go-rithms”…wha?

- Remember Logarithms?
  - they have nothing to do with Algorithms
- Well specified method for solving a problem u sing a finite sequence of instructions
- Description might be English, Pseudocode, or real code
- Key: no ambiguity

Al-Khwārizmī (780-850)

Efficient Algorithms: Why?

- Solving problems consumes resources that are often limited/valuable:
  - Time: Plan a flight path
  - Space: Query a database
  - Energy: Save money
- Bigger problems consume more resources
- Need algorithms that “scale” to large inputs, e.g. searching the web…

Efficient Algorithms: How?

- Define problem:
  - Unambiguous description of desired result
- Abstract irrelevant detail
  - “Assume the cow is a sphere”
- Pull techniques from the “algorithmic toolbox”
  - [CLRS] class textbook
- Implement and evaluate performance
  - Revise problem/abstraction
- Generalize
  - Algorithm to apply to broad class of problems

Administrivia

- Handout: course info
- Profs: Daskalakis, Jaillet
- TAs: Goldstein, Griner, Bhattacharya, Madry
- Sign up for class at https://sec.csail.mit.edu/ to get a recitation assignment
- Prereqs: 6.01, 6.042
- Python
- Grades: Problem sets (30%)
  Quiz1 (Oct 13: 7.30-9.30pm; 20%)
  Quiz2(Nov 17: 7.30-9.30pm; 20%)
  Exam (30%)
- Read collaboration policy!

Content

- 8 modules with motivating problem/pset
- Linked Data Structures: Document Distance
- Divide&Conquer: Peak Finding
- Hashing: Efficient File Update/Synchronization
- Sorting
- Graph Search: Rubik's Cube
- Shortest Paths: Google Maps
- Dynamic Programming: print justification
- Numerical Algorithms: linear systems

Document Distance

- Given 2 documents, how similar are they?
  - if one “document ” is a query, this is web search
  - find “similar documents” to a given one
  - detect plagiarism
- Goal: algorithm to compute similarity

Problem Definition

- Need unambiguous definition of similarity

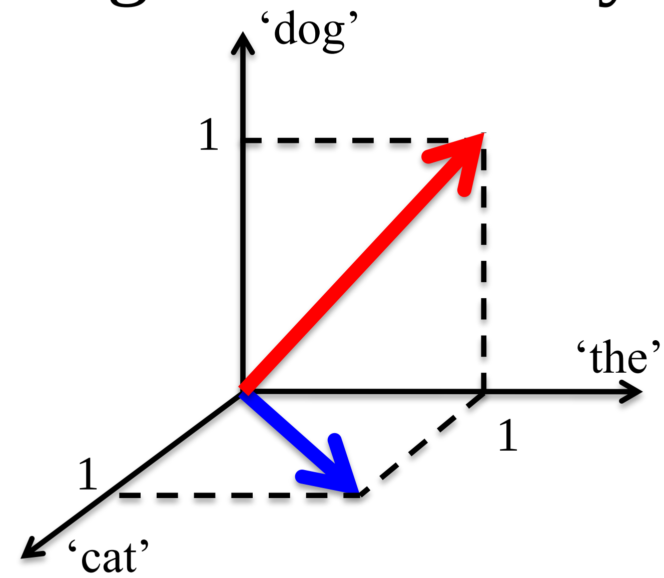
- Word: sequence of alpha characters
  - Ignore punctuation, formatting
- Document: sequence of words
- Word frequencies:

  $D(w)$ is number of occurences of w in D
- Similarity based on amount of word overlap

Vector Space Model

- [Salton, Wang Yang 1975]
- Treat each doc as a vector of its words
  - one coordinate per word of the English dictionary

    e.g. doc1 = "the cat"
    doc2 = "the dog"



  - similarity by dot-product

$$D_1 \circ D_2 \equiv \sum_w D_1(w) \cdot D_2(w)$$

$d1 \circ d2 = 1$

  - trouble: not scale invariant

    documents "the the cat cat" and "the the dog dog" will appear closer than doc1 and doc2

Vector Space Model

- Solution: Normalization
  - divide by the length of the vectors

$$\frac{D_1 \circ D_2}{||D_1|| \cdot ||D_2||}$$

  - measure distance by angle:

$$\theta(D_1, D_2) = \text{acos}\left(\frac{D_1 \circ D_2}{||D_1|| \cdot ||D_2||}\right)$$

e.g. $\theta=0$ documents “identical”
(if of the same size, permutations of each other)

$\theta=\pi/2$ not even share a word

Algorithm

- Read file
- Make word list (divide file into words)
- Count frequencies of words
- Compute dot product
  - for every word in the first document, check if it appears in the other document; if yes, multiply their frequencies and add to the dot product
    - worst case time: order of #words(D_1) x #words(D_2)
  - micro-optimization:
    - sort documents into word order (alphabetically)
    - compute inner product in time #words(D_1) + #words(D_2)

Python Implementation

- Docdist1.py (see handout)
- Read file: read_file(filename)
  - Output: list of lines (strings)
- Make word list: get_words_from_line_list(L)
  - Output: list of words (array)
- Count frequencies: count_frequency(word list)
  - Output: list of word-frequency pairs
- Sort into word order: insertion_sort()
  - Output: sorted list of pairs
- Dot product: inner_product(D1, D2)
  - Output: number

Inputs:

- Jules Verne: 25K
- Bobsey Twins: 268K
- Francis Bacon: 324K
- Lewis and Clark: 1M
- Shakespears: 5.5M
- Churchill: 10M

Profiling

- Tells how much time spent in each routine
  - import profile
  - profile.run(“main()”)
- One line per routine reports
  1. #calls
  2. #total time excluding subroutine calls
  3. Time per call (#2/#1)
  4. Cumulative time, including subroutines
  5. Cumulative per call (#4/#1)

auk:~/Class/6006/lectures/l01/

File Edit View Options Transfer Script Tools Help

auk:~/Class/6006/lectures/l01/

```
t1.verne.txt          t4.arabian.txt         t7.tenmillion.txt
t2.bobsey.txt         t5.churchill.txt       t8.shakespeare.txt
t3.lewis.txt          t6.onemillion.txt      t9.bacon.txt
auk:l01> python source/docdist2.py data/t2.bobsey.txt data/t3.lewis.txt
File data/t2.bobsey.txt : 6667 lines, 49785 words, 3354 distinct words
File data/t3.lewis.txt : 15996 lines, 182355 words, 8530 distinct words
The distance between the documents is: 0.574160 (radians)
         3861660 function calls in 94.738 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(acos)
  1241849    4.320    0.000    4.320    0.000 :0(append)
  1300248    4.432    0.000    4.432    0.000 :0(isalnum)
   232140    0.772    0.000    0.772    0.000 :0(join)
   368314    1.300    0.000    1.300    0.000 :0(len)
   232140    0.760    0.000    0.760    0.000 :0(lower)
        2    0.000    0.000    0.000    0.000 :0(open)
        2    0.000    0.000    0.000    0.000 :0(range)
        2    0.008    0.004    0.008    0.004 :0(readlines)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(sqrt)
        1    0.004    0.004   94.738   94.738 <string>:1(<module>)
        2   34.366   17.183   34.394   17.197 docdist2.py:105(count_frequency)
        2    9.781    4.890    9.781    4.890 docdist2.py:122(insertion_sort)
        2    0.000    0.000   94.438   47.219 docdist2.py:144(word_frequencies_for_file
)
        3    0.156    0.052    0.292    0.097 docdist2.py:162(inner_product)
        1    0.000    0.000    0.292    0.292 docdist2.py:188(vector_angle)
        1    0.004    0.004   94.734   94.734 docdist2.py:198(main)
        2    0.000    0.000    0.008    0.004 docdist2.py:49(read_file)
[illegible]   23.605   11.803   50.255   25.128 docdist2.py:65(get_words_from_line_list)
 226[illegible]   12.409    0.001   26.650    0.001 docdist2.py:77(get_words_from_string)
        1    0.000    0.000   94.738   94.738 profile:0(main())
        0    0.000             0.000          profile:0(profiler)
   232140    1.424    0.000    2.184    0.000 string.py:218(lower)
   232140    1.396    0.000    2.168    0.000 string.py:306(join)


auk:l01> █
```

Ready   ssh2: AES-256   41, 10   41 Rows, 87 Cols   VT100   CAP   NUM docdist1.py - C:\Documents and Settings\David\My Doc...

File Edit Format Run Options Windows Help

```python
###################################################
# Operation 2: split the text lines into words ##
###################################################
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """

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

Ln: 130 Col: 18

What's with +?

- L=L1+L2 is concatenation of arrays
- Take L1 and L2
- Copy to a bigger array
- Time proportional to sum of lengths
- Suppose n one-word lines
- Time $1+2+\ldots+n = n(n+1)/2 = \cup(n^2)$

Solution

- word_list.extend(words_in_line) : appends list named “words_in_line” to list named “word_list”
- Takes time proportional to length of list “words_in_line”
- Total time in example of n one-word lines: ∪(n)
- resulting improvement:
  - get_words_from_line_list 23s→0.12s

Other Improvements

- Docdist4.py:
  - Instead of inserting words in list, insert in dictionary: total to 42s
- 5.py:
  - Process words instead of chars: to 17s
- 6.py: merge sort instead of insertion: 6s
- 7.py: dictionary (again) instead of sort: 0.5s

Next time: Peak Finding

- Array of numbers
- Find one that is bigger than its neighbors
- A local minimum