Today: Models of Computation
- what's an algorithm? what is time?
- random access machine
- pointer machine
- Python model
- document distance: problem & algorithms

History: al-Khwārizmī “al-kha-raz-mī” (c. 780-850)
- “father of algebra” with his book “The Compendious Book on Calculation by Completion & Balancing”
- linear & quadratic equation solving: some of the first algorithms

http://en.wikipedia.org/wiki/Al-Khwarizmi

What's an algorithm?
- mathematical abstraction of computer program
- computational procedure to solve a problem
Model of computation specifies
- what operations an algorithm is allowed
- cost (time, space, ...) of each operation
→ cost of algorithm = sum of op. costs

1. Random Access Machine (RAM):
   - Random Access Memory (RAM)
     modeled by a big array
   - $O(1)$ registers (each 1 word)
   - in $O(1)$ time, can
     - load word $@r_i$ into register $r_j$
     - compute ($+, -, \times, \div, \&$, $\|$, $\wedge$) on registers
     - store register $r_j$ into memory $@r_i$
   - what's a word? $w \geq \log(\text{mem. size})$ bits
   - assume basic objects (e.g. int) fit in word
   - Unit 4 deals with big numbers
   - realistic & powerful → implement abstractions

2. Pointer Machine:
   - dynamically allocated objects
   - object has $O(1)$ fields
   - field = word (e.g. int)
     or pointer to object/null
     (a.k.a. reference)
   - weaker than (can be implemented on) RAM
Python lets you use either mode of thinking:

- "list" is actually an array → RAM
  \[ L[i] = L[j] + 5 \] → \( \Theta(1) \) time

- object with \( O(1) \) attributes → pointer machine
  \( \rightarrow \) including references → \( \Theta(1) \) time

Other operations: Python has many
- to determine their cost, imagine implementation in terms of ① or ②

**list:**
- \( L.append(x) \) → \( \Theta(1) \) time
  - obvious if you think of infinite array
  - but how would you have \( >1 \) on RAM?
  - via table doubling [Lecture 9]

- \( \ell = L1 + L2 \)  
  \( \Downarrow \)
  \( \Theta(1 + |L1| + |L2|) \) time

- \( L1.extend(L2) \equiv L1 += L2 \)
  \( \ell \equiv L1 +++ L2 \)
  \( \ell \equiv L1.append(x) \) \( \Theta(1) \) time
- \( L_2 = L_1[i:j] = L_2 = [] \)
  for \( k \) in range\((i,j)\):
  \( L_2.append(L_1[i:j]) \)
  \( \Theta(j-i+1) = O(1) \)

- \( b = x \in L \)
  & L.index(x) & L.find(x)
  for \( y \) in \( L \):
  if \( x == y \):
    \( b = True \)
    break
  else:
    \( b = False \)

- \( \text{len}(L) \to \Theta(1) \) time
- List stores its length in a field

- \( L.sort() \to \Theta(|L_1| \log |L_1|) \)
  - via comparison sort \[[\text{Lecture} 3 (\& 4 \& 7)]\]

\underline{tuple, str: similar (think of as immutable lists)}

\underline{dict:} \( D[\text{key}] = \text{val.} \)
  key in \( D \)
  \( \Theta(1) \) time with high probability

\underline{set: similar (think of as dict without values.)}

\underline{heapq: heappush & heappop \to \Theta(\log n) \) time}

\underline{long:}
- \( x+y \to O(|x|+|y|) \) time \( \approx 1.58 \)
- \( x*y \to O((|x|+|y|) \log^3) \) time

\underline{via Karatsuba algorithm} \[[\text{Lecture} 11] \]
Document distance problem: compute \( d(D_1, D_2) \)

- **Applications:** find similar documents
  - Wikipedia mirrors & Google
  - web search (\( D_2 = \) query)
  - detect duplicates & plagiarism

- **Word:** sequence of alphanumeric chars.
- **Document:** sequence of words
  - ignore space, punctuation, etc.

- **Idea:** define distance in terms of shared words

- Think of document \( D \) as vector:
  \[ D[w] = \# \text{ occurrences of word } w \]

- E.g.:
  \[ D_1 = \text{“the cat”}, \quad D_2 = \text{“the dog”} \]

- **Attempt 1:** \( d'(D_1, D_2) = D_1 \cdot D_2 = \sum_w D_1[w] \cdot D_2[w] \)

- **Problem:** not scale invariant
  - \( \Rightarrow \) long docs. with 99% same words
     seem farther than short docs. with 10%

- **Fix:** normalize by \# words:
  \[ d''(D_1, D_2) = \frac{D_1 \cdot D_2}{\sqrt{|D_1| \cdot |D_2|}} \]

- **Geometric:** \( d(D_1, D_2) = \arccos d''(D_1, D_2) \)
  - angle between vectors
  - \( 90^\circ = \) not

[Salton, Wong, Yang 1975]
Document distance algorithm:
1. split each document into words
2. count word frequencies (document vectors)
3. compute dot product (& divide)

1: re.findall(r"\w+", doc) \rightarrow \text{what cost?} \\
\sim \text{in general, re can be exponential time!} \\
\rightarrow \text{for char in doc:} \\
\quad \text{if not alphanumeric:} \\
\quad \quad \text{add previous word (if any) to list} \\
\quad \quad \text{start new word} \\
\quad \Theta(1) \\
\quad \Theta(1) \rightarrow \text{to compare}

2: sort word list \leftarrow O(k \lg k \cdot |\text{word list}|) \\
\quad \text{for word in list:} \\
\quad \quad \text{if same as last word:} \leftarrow O(|\text{word list}|) \\
\quad \quad \quad \text{increment counter} \\
\quad \quad \text{else:} \\
\quad \quad \quad \text{add last word & count to list} \\
\quad \quad \quad \text{reset counter to 0} \\
\quad \Theta(1) \rightarrow \Theta(1) = O(|\text{doc} 1|)

3: for word, count1 in doc1: \leftarrow \Theta(k_1) \}
\quad \text{if word, count2 in doc2:} \leftarrow O(k_2) \\
\quad \text{total += count1 \times count2} \rightarrow O(1)
③: start at first word of each list
  if words equal: \( \leq O(|\text{word}|) \)
  total += count1 * count2
  if word1 ≤ word2: \( \leq O(|\text{word}|) \)
    advance list1
  else:
    advance list2
  repeat until either list done

Dictionary approach:

①′: count = {}
  for word in doc:
    if word in count:
      count[word] += 1
    else:
      count[word] = 1
\(\Theta(1)\) w.h.p.

③ as above \(\rightarrow O(|\text{doc}_1|)\) w.h.p.
Code: (lecture2_code.zip & _data.zip on website)
t2.bobsey.txt 268,778 chars/49,785 words/3,354 uniq
  t3.lewis.txt 1,031,470 words/182,355 words/8,530 uniq
  seconds on Pentium 4, 2.86GHz, C-Python 2.6.2, Linux 2.6.26

- docdist 1: 228.1 - ①, ②, ③ (with extra sorting)
  - words = words + words.on.line
- docdist 2: 164.7 - words += words.on.line
- docdist 3: 123.1 - ③ '... with insertion sort'
- docdist 4: 71.7 - ② 'but still sort to use ③'
- docdist 5: 18.3 - split words via string, translate
- docdist 6: 11.5 - merge sort (vs. insertion)
- docdist 7: 1.8 - ③ (full dictionary)
- docdist 8: 0.2 - whole doc., not line by line