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-rāz-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
- $\Theta(1)$ registers (each 1 word)
- in $\Theta(1)$ time, can
  - load word $\hat{r}_i$ into register $r_j$
  - compute ($+, -, \times, \div, \wedge, \vee$) on registers
  - store register $r_j$ into memory $\hat{r}_i$
- what's a word? $w \geq \lg(\text{mem. size})$ bits
- assume basic objects (e.g. int) fit in word
- Unit 4 deals with big numbers
- realistic & powerful $\Rightarrow$ implement abstractions

2) Pointer Machine:
- dynamically allocated objects
- object has $\Theta(1)$ 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:

1. "list" is actually an array \( \to \) RAM
   - \( L[i] = L[j] + 5 \) \( \to \) \( \Theta(1) \) time
2. object with \( O(1) \) attributes \( \to \) pointer machine
   - \( x = x\_next \) \( \to \) \( \Theta(1) \) time

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

**list**:
- \( L\_append(x) \) \( \to \) \( \Theta(1) \) time
  - obvious if you think of infinite array
  - but how would you have \( >1 \) on RAM?
- via table doubling \( \left[ \text{Lecture 9} \right] \)

- \( L = L1 + L2 \equiv L = [] \)
  - \( \Theta(1) \)
  - for \( x \) in \( L1 \):
    - \( L\_append(x) \) \( \Theta(1) \)
  - for \( x \) in \( L2 \):
    - \( L\_append(x) \) \( \Theta(1) \)

- \( L1\_extend(L2) \equiv L1 += L2 \)
  - \( \Theta(1) \)
- \( L2 = L1[i:j] \) = \( L2 = [] \) for \( k \) in range\((i,j)\): \( L2.append(L1[i:j]) \) \( \{ \Theta(j-i) + 1 \} = O(1L1) \) 

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

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

- \( L.sort() \) \( \rightarrow \Theta(1L1 \log L1) \)
- via comparison sort [Lecture 3 (§4 & 7)]

- tuple, str: similar (think of as immutable lists)

- dict: \( D[\text{key}] = \text{val.} \) \( \{ \Theta(1) \text{ time} \) with high probability
- via hashing [Unit 3 = Lectures 8-10]

- set: similar (think of as dict without vals.)

- heapq: heappush & heappop \( \rightarrow \Theta(lg n) \) time
- via heaps [Lecture 4]

- long: \( x+y \) \( \rightarrow O(|x|+|y|) \) time \( \approx 1.58 \)
- \( x*y \) \( \rightarrow O((|x|+|y|) \log^3) \) time
- via Karatsuba algorithm [Lecture 11]
Document distance problem: compute $d(D_1, D_2)$

- **Applications:** find similar documents, detect duplicates & plagiarism, web search ($D_2 =$ query)

- 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: $D[w] =$ # occurrences of word $w$
- e.g.: $D_1 =$ "the cat" $D_2 =$ "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: $d''(D_1, D_2) = \frac{D_1 \cdot D_2}{|D_1| \cdot |D_2|}$

- geometric (rescaling): $d(D_1, D_2) = \arccos d''(D_1, D_2)$ (20° = same, 90° = 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:}
     \text{if not alphanumeric:}
       \text{add previous word (if any) to list}
     \text{start new word}
     \Theta(1)

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

3: for word, count1 in doc1:
   if word, count2 in doc2:
     total += count1 * count2
   \Theta(1)
(3): start at first word of each list
   if words equal: $\leq O(\lvert \text{word1} \rvert)$
   \[ \text{total} += \text{count1} \times \text{count2} \]
   if word1 $\leq$ word2: $\leq O(\lvert \text{word1} \rvert)$
   advance list1
   else:
   advance list2
   repeat until either list done

Dictionary approach:

(2)': \[ \text{count} = \{ \} \]
   for word in doc:
   if word in count:
     count[word] += 1
   else:
     count[word] = 1

(3) as above $\rightarrow O(\lvert \text{doc}\rvert)$ w.h.p.

\[ O(\Sigma \lvert \text{word1} \rvert) = O(\lvert \text{doc} \rvert) \]

\[ \Theta(\lvert \text{word1} \rvert) + \Theta(1) \text{ w.h.p.} \]

\[ \Theta(1) \]

with high prob
Code: (lecture2_code.zip & _data.zip on website)

- 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