

TODAY: Models of Computation

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

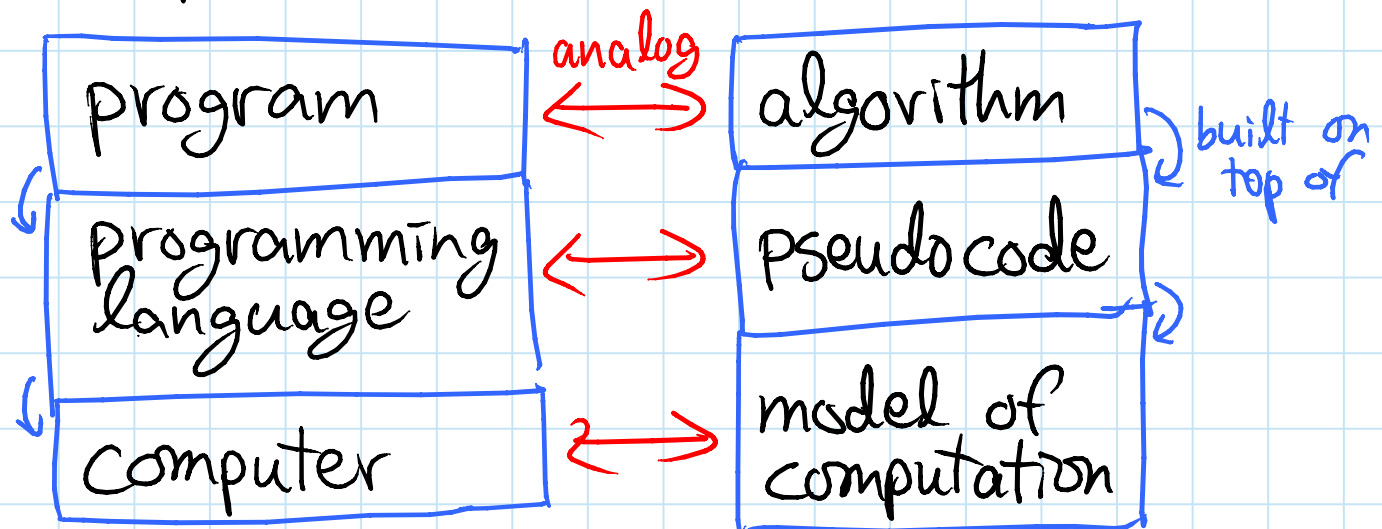
History: al-Khwārizmī "al-kha-ra-z-mi" (c. 780-850)

- "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

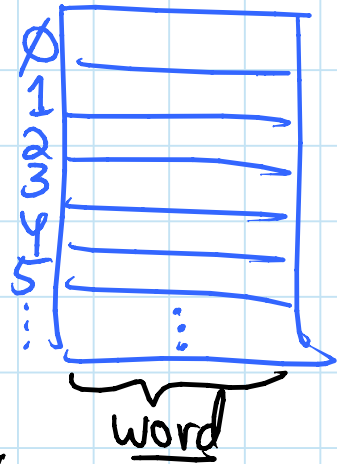


Model of computation specifies

- what operations an algorithm is allowed
- cost (time, space, ...) of each operation
- cost of algorithm = sum of op. costs

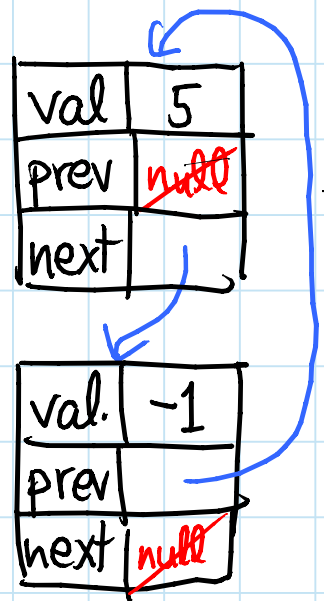
① Random Access Machine (RAM):

- Random Access Memory (RAM) modeled by a big array
- $\Theta(1)$ registers (each 1 word)
- in $\Theta(1)$ time, can
 - load word @ r_i into register r_j
 - compute $(+, -, *, /, \&, |, ^)$ on registers
 - store register r_j into memory @ 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 $\leadsto$ implement abstractions



② Pointer Machine: (named tuple)

- 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 $\rightarrow$ RAM
 - $L[i] = L[j] + 5 \rightarrow \Theta(1)$ time
- ② object with $O(1)$ attributes $\rightarrow$ pointer machine
 - $\hookrightarrow$ including references
 - $x = x.next \rightarrow \Theta(1)$ time

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

list:

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

- $L = L1 + L2 \equiv L = []$ $\{ \Theta(1) \}$
 $\downarrow$
 $\Theta(1 + |L1| + |L2|)$ time
for x in $L1$:
 $L.append(x)$ $\{ \Theta(1) \}$ $\{ \Theta(|L1|) \}$
for x in $L2$:
 $L.append(x)$ $\{ \Theta(1) \}$ $\{ \Theta(|L2|) \}$

- $L1.extend(L2) \equiv$ for x in $L2$:
 $L1.append(x)$ $\{ \Theta(1) \}$ $\{ \Theta(1 + |L2|) \}$ time
 $\equiv L1 += L2$

- $L2 = L1[i:j] \equiv L2 = []$
 for k in $\text{range}(i, j)$:
 $L2.append(L1[i])$ } $\Theta(1)$ } $\Theta(j-i+1) = O(|L|)$
- $b = x$ in $L \equiv$ for y in L :
 $\& L.index(x)$
 $\& L.find(x)$
 if $x == y$:
 $b = \text{True}$
 break } $\Theta(1)$ } $\Theta(\text{index of } x) = O(|L|)$
 else:
 $b = \text{False}$ } $= O(|L|)$ in worst case
- $\text{len}(L) \rightarrow \Theta(1)$ time
 - list stores its length in a field
- $L.sort() \rightarrow \Theta(|L| \lg |L|)$
 - via comparison sort [Lecture 3 (& 4 & 7)]

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

dict: $D[\text{key}] = \text{val.}$ } $\Theta(1)$ time
 key in D } 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 $\rightarrow \approx 1.58$
 $x*y \rightarrow O((|x|+|y|)^{\lg 3})$ time
 - via Karatsuba algorithm [Lecture 11]

Document distance problem: compute $d(D_1, D_2)$

- applications: find similar documents
detect duplicates & plagiarism
web search ($D_2 = \text{query}$)
Wikipedia mirrors & Google

- word = sequence of alphanumeric chars.

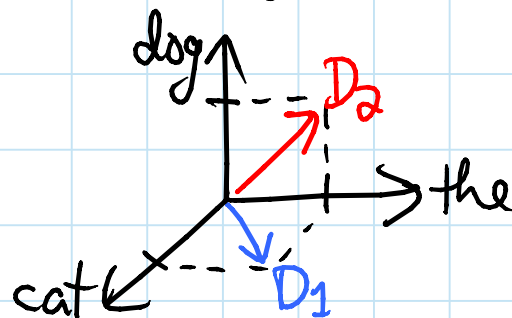
- document = sequence of words
(ignore space, punctuation, etc.)

- idea: define distance in terms of shared words

- think of document D as vector:

$D[w] = \# \text{ occurrences of word } w$

- e.g.: $D_1 = \text{"the cat"}$
 $D_2 = \text{"the dog"}$



- 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: $d''(D_1, D_2) = \frac{D_1 \cdot D_2}{\# \text{ words} \rightarrow |D_1| \cdot |D_2|}$

- geometric: $d(D_1, D_2) = \arccos d''(D_1, D_2)$ $\begin{cases} 0 = \text{same} \\ 90^\circ = \text{not} \end{cases}$
(rescaling) = angle between vectors

[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)` → what cost?
~ in general, `re` can be exponential time!

→ for char in doc:

if not alphanumeric;
add previous word
(if any) to list
start new word

$\Theta(1)$ } $\Theta(|doc|)$

↑ to compare

②: sort word list $\leftarrow O(k \lg k \cdot |word|)$
for word in list: $\leftarrow \# \text{ words}$

if same as last word: $\leftarrow O(|word|)$
increment counter
else:

add last word & count to list
reset counter to 0

$\Theta(1)$ } $O(\sum |word|)$
 $= O(|doc|)$

③: for word₁, count₁ in doc₁: $\leftarrow \Theta(k_1)$
if word₁, count₂ in doc₂: $\leftarrow \Theta(k_2)$
total += count₁ * count₂ } $\Theta(1)$ } $O(k_1 \cdot k_2)$

③': start at first word of each list
 if words equal: $\leftarrow O(|word|)$
 total += count1 * count2
 if word1 $\leq$ word2: $\leftarrow O(|word|)$
 advance list1
 else:
 advance list2
 repeat until either list done

$O(\sum |word|)$
 $= O(|doc|)$

Dictionary approach:

②': count = {}
 for word in doc:
 if word in count: $\leftarrow O(|word|)$
 count[word] += 1 $+ O(1)$ w.h.p.
 else:
 count[word] = 1 $\leftarrow O(1)$

③ as above $\rightarrow O(|doc_1|)$ w.h.p.

$O(|doc|)$
 with high prob.

Code: (lecture2_code.zip & -data.zip on website)
t2.bobsey.txt 268,778 chars / 49,785 words / 3,354 unq
t3.lewis.txt 1,031,470 / 182,355 / 8,530 unq
seconds on Pentium 4, 2.8GHz, 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