

TODAY: Models of Computation

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

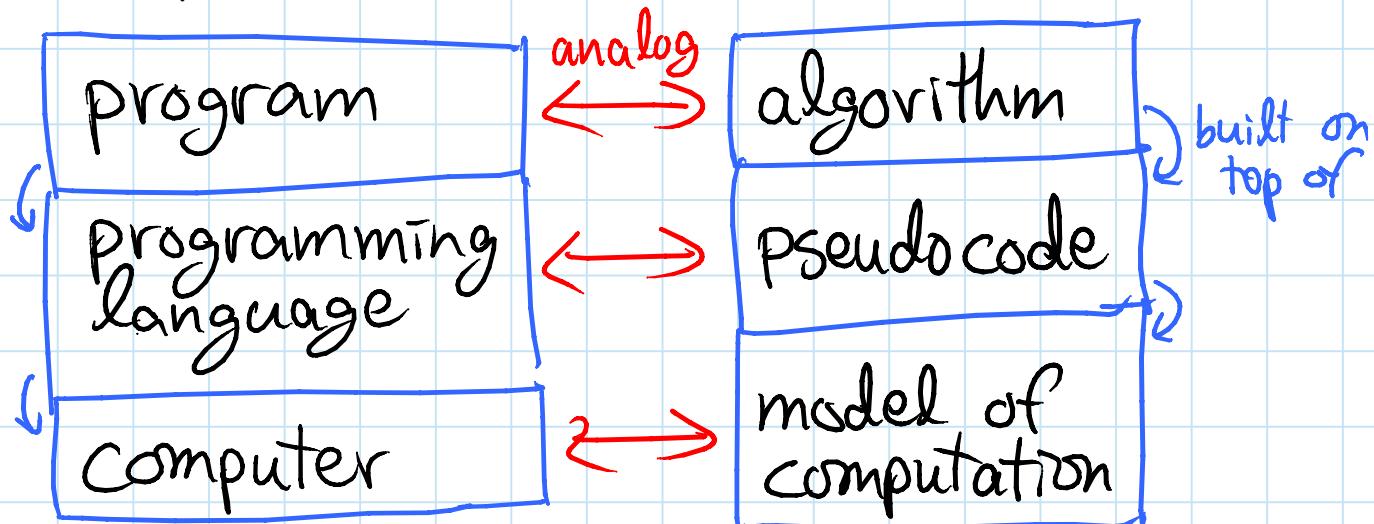
History: al-Khwārizmī "al-kha-raz-mi" (c. 780-850)

- "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

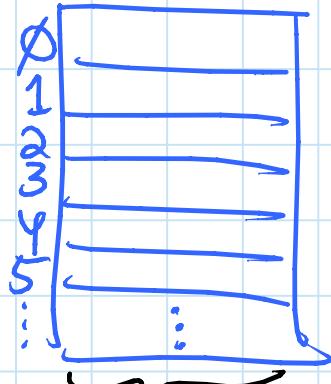


Model of computation specifies

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

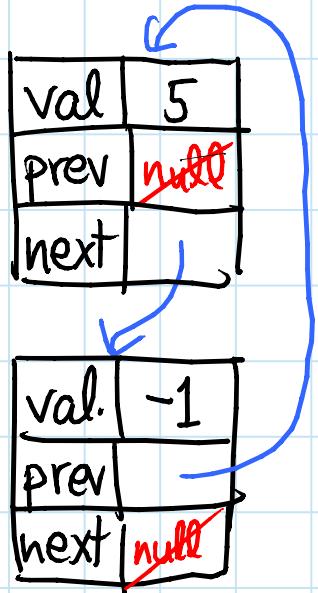
① Random Access Machine (RAM):

- 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 ($+, -, *, /, \&, \mid, ^T$) 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: (namedtuple)

- 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 $\Theta(1)$ attributes \rightarrow pointer machine
↳ including references
- $x = x.\text{next}$ $\rightarrow \Theta(1)$ 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]

- $L = L_1 + L_2 \equiv L = [] \quad \left\{ \Theta(1) \right.$
 \downarrow
 $\Theta(1 + |L_1| + |L_2|)$ time
for x in L_1 :
 $L.append(x) \quad \left\{ \Theta(1) \right\} \quad \left\{ \Theta(|L_1|) \right\}$
for x in L_2 :
 $L.append(x) \quad \left\{ \Theta(1) \right\} \quad \left\{ \Theta(|L_2|) \right\}$

- $L_1.extend(L_2) \equiv$ for x in L_2 :
 $\equiv L_1 += L_2$ $L_1.append(x) \quad \left\{ \Theta(1) \right\} \quad \left\{ \Theta(1 + |L_2|) \right\}$ time

- $L2 = L1[i:j] \equiv L2 = []$
 $\quad \text{for } k \text{ in range}(i, j):$
 $\quad \quad L2.append(L1[k])$ $\left\{ \begin{array}{l} O(j-i) \\ +1 \end{array} \right\} = O(|L|)$
- $b = x \text{ in } L \equiv \text{for } y \text{ in } L:$
 $\quad \& L.index(x)$ $\quad \quad \text{if } x == y: \left\{ \begin{array}{l} O(1) \\ b = True \end{array} \right\} = O(|L|)$
 $\quad \& L.find(x)$ $\quad \quad \text{else: break} \quad \left\{ \begin{array}{l} O(\text{index of } x) \\ = O(|L|) \end{array} \right\} = O(|L|) \text{ in worst case}$
 $\quad \quad b = False$
- $\text{len}(L) \rightarrow O(1) \text{ time}$
 - List stores its length in a field
- $L.sort() \rightarrow O(|L| \lg |L|)$
 - via comparison sort [Lecture 3 (8 & 4 & 7)]

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

dict: $D[key] = \text{val.}$ $\left\{ \begin{array}{l} O(1) \text{ time} \\ \text{key in } D \end{array} \right\} \text{with high probability}$

- via hashing [Unit 3 = Lectures 8-10]

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

heapq: heappush & heappop $\rightarrow O(\lg n)$ time

- via heaps [Lecture 4]

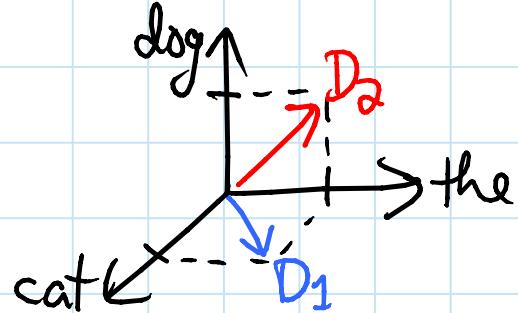
long: $x+y \rightarrow O(|x|+|y|)$ time $\rightarrow \approx 1.58$
 $x*y \rightarrow O((|x|+|y|)^{\lg 3})$ time

- via Karatsuba algorithm [Lecture 11]

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

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

- 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:
 $D[w] = \# \text{ occurrences of word } w$
- 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% same words
- fix: normalize by # words: $d''(D_1, D_2) = \frac{D_1 \cdot D_2}{\# \text{ words} \rightarrow |D_1| \cdot |D_2|}$
- geometric: $d(D_1, D_2) = \arccos d''(D_1, D_2)$
(rescaling)
 $= \text{angle between vectors}$

[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

} $\Theta(1)$

↑ to compare

②: sort word list $\leftarrow \Theta(k \lg k \cdot |\text{wordl}|)$

for word in list:

if same as last word: $\leftarrow \Theta(|\text{wordl}|)$
increment counter

else:

add last word & count to list
reset counter to 0

} $\Theta(1)$

$\Theta(\sum |\text{wordl}|)$
 $= \Theta(|\text{doc}|)$

③: for word, count1 in doc1: $\leftarrow \Theta(k_1)$

if word, count2 in doc2: $\leftarrow \Theta(k_2)$

total += count1 * count2 } $\Theta(1)$

} $\Theta(k_1 \cdot k_2)$

$\textcircled{3}'$: start at first word of each list }
 if words equal: $\leftarrow O(1|\text{word}|)$
 total += count₁ * count₂
 if word₁ \leq word₂: $\leftarrow O(1|\text{word}|)$
 advance list₁
 else:
 advance list₂
 repeat until either list done

$O(\sum |\text{word}|)$
 $= O(|\text{doc}|)$

Dictionary approach:

$\textcircled{2}'$: count = {} }
 for word in doc:
 if word in count: $\leftarrow O(1|\text{word}|)$ + $O(1)$ w.h.p.
 count[word] += 1 }
 else:
 count[word] = 1 } $O(1)$
 $\textcircled{3}$ as above $\rightarrow O(|\text{doc}_1|)$ w.h.p.

$O(|\text{doc}|)$
 with
 high
 prob.

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 chars / 182,355 words / 8,530 uniq
seconds on Pentium 4, 2.8GHz, C-Python 2.6.2,
Linux 2.6.26

- docdist1: 228.1
 - ①, ②, ③ (with extra sorting)
 - words = words + words.on_line
 - words += words.on_line
- docdist2: 164.7
 - ③' ... with insertion sort
- docdist3: 123.1
 - ②' but still sort to use ③'
- docdist4: 71.7
 - split words via string.translate
- docdist5: 18.3
 - merge sort (vs. insertion)
- docdist6: 11.5
 - ③ (full dictionary)
- docdist7: 1.8
 - whole doc., not line by line
- docdist8: 0.2