

## Lecture 2: Models of Computation

### Lecture Overview

- What is 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

### What is an Algorithm?

- Mathematical abstraction of computer program
- Computational procedure to solve a problem

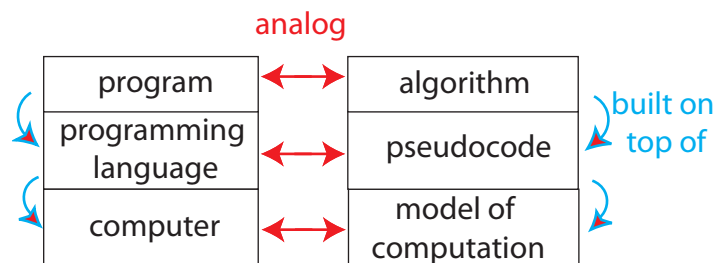
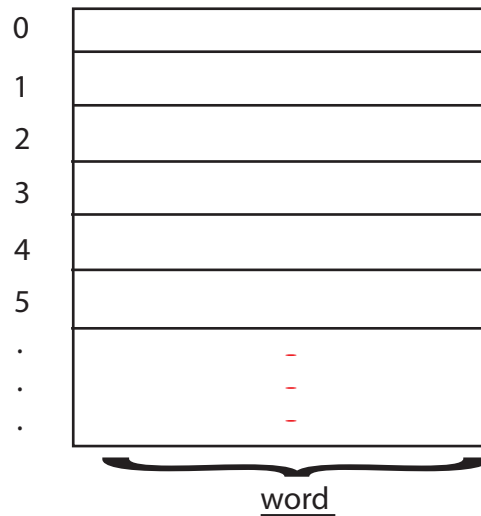


Figure 1: Algorithm

### Model of computation specifies

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

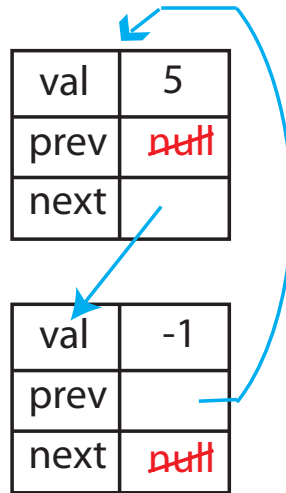
## Random Access Machine (RAM)



- 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?  $w \geq \lg(\text{memory size})$  bits
  - assume basic objects (e.g., int) fit in word
  - unit 4 in the course deals with big numbers
- realistic and powerful  $\rightarrow$  implement abstractions

## Pointer Machine

- dynamically allocated objects (namedtuple)
- 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 Model

Python lets you use either mode of thinking

1. “list” is actually an array  $\rightarrow$  RAM

$$L[i] = L[j] + 5 \rightarrow \Theta(1) \text{ time}$$

2. object with  $O(1)$  attributes (including references)  $\rightarrow$  pointer machine

$$x = x.next \rightarrow \Theta(1) \text{ time}$$

Python has many other operations. To determine their cost, imagine implementation in terms of (1) or (2):

1. list

- (a)  $L.append(x) \rightarrow \theta(1) \text{ time}$

obvious if you think of infinite array

but how would you have  $> 1$  on RAM?

via *table doubling* [Lecture 9]

- (b)  $\underbrace{L = L1 + L2}_{(\theta(1+|L1|+|L2|) \text{ time})} \equiv L = [] \rightarrow \theta(1)$

for $x$ in $L1$ :	}	$\theta( L_1 )$	}
$L.append(x) \rightarrow \theta(1)$			
for $x$ in $L2$ :	}	$\theta( L_2 )$	}
$L.append(x) \rightarrow \theta(1)$			

- (c)  $L1.extend(L2) \equiv \text{for } x \text{ in } L2:$   
 $\equiv L1 + = L2 \quad L1.append(x) \rightarrow \theta(1)$   $\left. \vphantom{\begin{array}{l} L1.extend(L2) \\ \equiv L1 + = L2 \end{array}} \right\} \theta(1 + |L_2|) \text{ time}$
- (d)  $L2 = L1[i : j] \equiv L2 = []$   
 $\text{for } k \text{ in range}(i, j):$   
 $L2.append(L1[i]) \rightarrow \theta(1)$   $\left. \vphantom{\begin{array}{l} L2 = L1[i : j] \\ \equiv L2 = [] \end{array}} \right\} \theta(j - i + 1) = O(|L|)$
- (e)  $b = x \text{ in } L \equiv \text{for } y \text{ in } L:$   
 $\& L.index(x) \quad \text{if } x == y:$   
 $\& L.find(x) \quad \quad b = True;$   
 $\quad \quad \quad \text{break}$   
 $\quad \quad \quad \text{else}$   
 $\quad \quad \quad b = False$   $\left. \vphantom{\begin{array}{l} b = x \text{ in } L \\ \& L.index(x) \\ \& L.find(x) \end{array}} \right\} \theta(1)$   $\left. \vphantom{\begin{array}{l} b = x \text{ in } L \\ \& L.index(x) \\ \& L.find(x) \end{array}} \right\} \theta(\text{index of } x) = \theta(|L|)$
- (f)  $\text{len}(L) \rightarrow \theta(1) \text{ time}$  - list stores its length in a field
- (g)  $L.sort() \rightarrow \theta(|L| \log |L|)$  - via *comparison sort* [Lecture 3, 4 & 7]
2. tuple, str: similar, (think of as immutable lists)
3. dict: via *hashing* [Unit 3 = Lectures 8-10]  
 $D[\text{key}] = \text{val}$   
 $\text{key in } D$   $\left. \vphantom{\begin{array}{l} D[\text{key}] = \text{val} \\ \text{key in } D \end{array}} \right\} \theta(1) \text{ time w.h.p.}$
4. set: similar (think of as dict without vals)
5. heapq:  $\text{heappush}$  &  $\text{heappop}$  - via *heaps* [Lecture 4]  $\rightarrow \theta(\log(n)) \text{ time}$
6. long: via *Karatsuba algorithm* [Lecture 11]  
 $x + y \rightarrow O(|x| + |y|) \text{ time}$  where  $|y|$  reflects # words  
 $x * y \rightarrow O((|x| + |y|)^{\log(3)}) \approx O((|x| + |y|)^{1.58}) \text{ time}$

## Document Distance Problem — compute $d(D_1, D_2)$

The document distance problem has applications in finding similar documents, detecting duplicates (Wikipedia mirrors and Google) and plagiarism, and also in web search ( $D_2 = \text{query}$ ).

Some Definitions:

- Word = sequence of alphanumeric characters
- Document = sequence of words (ignore space, punctuation, etc.)

The idea is to define distance in terms of shared words. Think of document  $D$  as a vector:  
 $D[w] = \# \text{ occurrences of word } W$ . For example:

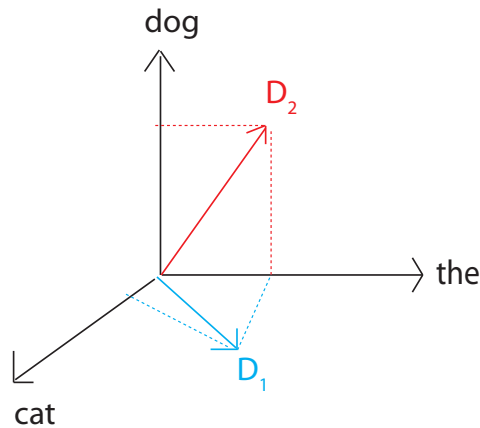


Figure 2:  $D_1$  = “the cat”,  $D_2$  = “the dog”

As a first attempt, define document distance as

$$d'(D_1, D_2) = D_1 \cdot D_2 = \sum_W D_1[W] \cdot D_2[W]$$

The problem is that this is not scale invariant. This means that long documents with 99% same words seem farther than short documents with 10% same words.

This can be fixed by normalizing by the number of words:

$$d''(D_1, D_2) = \frac{D_1 \cdot D_2}{|D_1| \cdot |D_2|}$$

where  $|D_i|$  is the number of words in document  $i$ . The geometric (rescaling) interpretation of this would be that:

$$d(D_1, D_2) = \arccos(d''(D_1, D_2))$$

or the document distance is the angle between the vectors. An angle of  $0^\circ$  means the two documents are identical whereas an angle of  $90^\circ$  means there are no common words. This approach was introduced by [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	}	$\Theta( doc )$
add previous word		
(if any) to list		
start new word		
- (2) sort word list  $\leftarrow O(k \log k \cdot |word|)$  where  $k$  is #words  
 for word in list:
 

if same as last word: $\leftarrow O( word )$	}	$\Theta(1)$	}	$O(\sum  word ) = O( doc )$
increment counter				
else:				
add last word and count to list				
reset counter to 0				
- (3) for word, count1 in doc1:  $\leftarrow \Theta(k_1)$   
 if word, 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 ≤ word2:  $\leftarrow O(|word|)$   
 advance list1  
 else:  
 advance list2  
 repeat either until list done
 

}	$O(\sum  word ) = O( doc )$
---	-----------------------------

### Dictionary Approach

- (2)' count = {}  
 for word in doc:
 

if word in count: $\leftarrow \Theta( word ) + \Theta(1)$ w.h.p	}	$\Theta(1)$	}	$O( doc )$ w.h.p.
count[word] += 1				
else				
count[word] = 1				
- (3)' as above  $\rightarrow O(|doc_1|)$  w.h.p.

**Code (lecture2\_code.zip & \_data.zip on website)**

t2.bobsey.txt 268,778 chars/49,785 words/3354 uniq

t3.lewis.txt 1,031,470 chars/182,355 words/8534 uniq

seconds on Pentium 4, 2.8 GHz, C-Python 2.62, Linux 2.6.26

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

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.006 Introduction to Algorithms  
Fall 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.