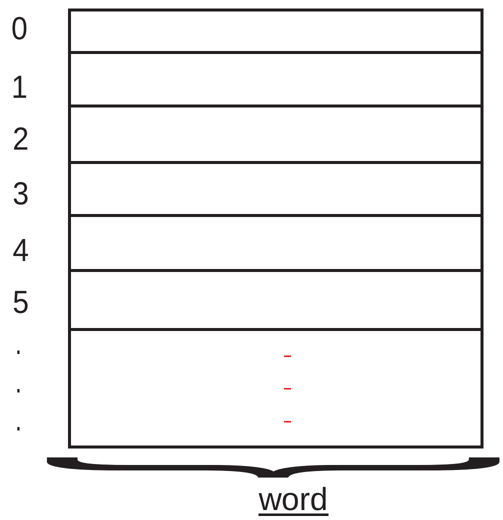
Random Access Machine (RAM)



- 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 $(+, -, *, /, \&, |, \hat{})$ on registers
  - store register r_j into memory @ r_i
- What's a word? $w \geq \lg$ (memory size) bits
  - assume basic objects (e.g., int) fit in word
  - unit 4 in the course deals with big numbers
- realistic and powerful $\longrightarrow$ implement abstractions

Python Model

Python lets you use either mode of thinking

1. "list" is actually an array $\longrightarrow$ RAM

   $L[i] = L[j] + 5 \longrightarrow \Theta(1)$ time

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

   $x = x.next \longrightarrow \Theta(1)$ time

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

1. list

(a) L.append(x) $\longrightarrow \theta(1)$ time

obvious if you think of infinite array

but how would you have > 1 on RAM?
via *table doubling* [Lecture 9]

(b) $\underbrace{L = L1 + L2}$ $\equiv L = [\,] \longrightarrow \theta(1)$
$(\theta(1+|L1|+|L2|)$ time)

for x in $L1$:
L.append(x) $\longrightarrow \theta(1)$ $\quad \theta(|L_1|)$

for x in $L2$:
L.append(x) $\longrightarrow \theta(1)$ $\quad \theta(|L_2|)$

(c) $L1.\text{extend}(L2) \equiv$ for x in $L2$: Σ $\theta(1 + |L_2|)$ time
$\equiv L1 += L2$ $\quad L1.\text{append}(x) \rightarrow \theta(1)$

(d) $L2 = L1[i:j] \equiv L2 = []$ Σ $\theta(j - i + 1) = O(|L|)$
for k in range(i, j):
$L2.\text{append}(L1[i]) \rightarrow \theta(1)$

(e) $b = x$ in L $\equiv$ for y in L: $\quad$ $\theta(\text{index of } x) = \theta(|L|)$
& L.index(x) $\quad$ if $x == y$: $\quad$ $\theta(1)$
& L.find(x) $\quad$ $b = True$;
break
else
$b = False$

(f) len(L) $\rightarrow \theta(1)$ 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] Σ

   D[key] = val $\quad$ $\theta(1)$ time w.h.p.
   key in D

4. set: similar (think of as dict without vals)
5. heapq: heappush & heappop - via *heaps* [Lecture 4] $\rightarrow \theta(\log(n))$ time
6. long: via *Karatsuba algorithm* [Lecture 11]

   $x + y \rightarrow O(|x| + |y|)$ time where $|y|$ reflects # words
   $x * y \rightarrow O((|x| + |y|)^{\log(3)}) \approx O((|x| + |y|)^{1.58})$ 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 = 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: $D[w]$ = # occurrences of word W. For example:

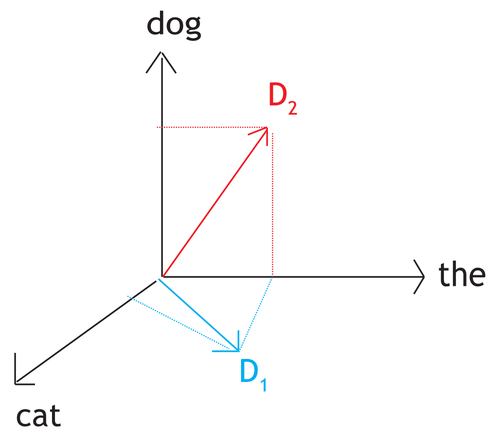


Figure 2: D_1 = "the cat", D_2 = "the dog"

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° means the two documents are identical whereas an angle of 90° 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) $\longrightarrow$ what cost? in general re can be exponential time

$\longrightarrow$ for char in doc:

$\Theta(|doc|)$

if not alphanumeric

add previous word (if any) to list

start new word

$\Theta(1)$

(2) sort word list $\leftarrow O(k \log k \cdot |word|)$ where k is #words

for word in list:

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

increment counter

else:

add last word and count to list

reset counter to 0

$\Theta(1)$

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

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

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

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

$O(k_1 \cdot k_2)$

(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

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

Dictionary Approach

(2)' count = {}

for word in doc:

if word in count: $\leftarrow \Theta(|word|) + \Theta(1)$ w.h.p

count[word] += 1

else

count[word] = 1

$\Theta(1)$

$O(|doc|)$ w.h.p.

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

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