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

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? $w \geq \lg$ (memory size) bits
  - assume basic objects (e.g., int) fit in word
  - unit 4 in the course deals with big numbers
- realistic and powerful → 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 $\to$ RAM

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

2. object with $O(1)$ attributes (including references) $\to$ 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) $\rightarrow \theta(1)$ time

   obvious if you think of infinite array

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

   via table doubling [Lecture 9]

\[
L = \underbrace{L_1 + L_2}_{(\theta(1 + |L_1| + |L_2|) \text{ time})} \equiv L = [] \rightarrow \theta(1)
\]

\[
\begin{cases}
\text{for } x \text{ in } L_1: \\
\quad \text{L.append(x)} \rightarrow \theta(1)
\end{cases}
\]

\[
\begin{cases}
\text{for } x \text{ in } L_2: \\
\quad \text{L.append(x)} \rightarrow \theta(1)
\end{cases}
\]

\[
\begin{cases}
\theta(|L_1|) \\
\theta(|L_2|)
\end{cases}
\]
Lecture 2

6.006 Fall 2011

(c) \(L1.extend(L2) \equiv \text{for } x \text{ in } L2:\)
\[\equiv L1+ = L2 \quad L1.append(x) \to \theta(1)\]
\[\theta(1 + |L2|) \text{ time}\]

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

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

(f) \(\text{len}(L) \to \theta(1) \text{ time} - \text{list stores its length in a field}\)

(g) \(\text{L.sort()} \to \theta(|L| \log |L|) - \text{via comparison sort [Lecture 3, 4 & 7]}\)

2. \text{tuple, str: similar, (think of as immutable lists)}

3. \text{dict: via hashing [Unit 3 = Lectures 8-10]}
\[D[\text{key}] = \text{val} \quad \text{key in } D\]
\[\theta(1) \text{ time w.h.p.}\]

4. \text{set: similar (think of as dict without vals)}

5. \text{heapq: heappush & heappop - via heaps [Lecture 4] \to \theta(\log(n)) \text{ time}}

6. \text{long: via Karatsuba algorithm [Lecture 11]}
\[x + y \to O(|x| + |y|) \text{ time} \quad \text{where } |y| \text{ reflects # words}\]
\[x * y \to O((|x| + |y|)^{\log(3)}) \approx O((|x| + |y|)^{1.58}) \text{ time}\]

\textbf{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:

- \textbf{Word} = sequence of alphanumeric characters
- \textbf{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 \textbf{vector}:
\[D[w] = \# \text{ occurrences of word } W. \text{ For example:}\]
As a first attempt, define document distance as

\[ d'(D_1, D_2) = D_1 \cdot D_2 = \sum W D_1[W] : D_2[W] \]

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:

\[ d''(D_1, D_2) = \frac{D_1 \cdot D_2}{|D_1| : |D_2|} \]

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) \( \text{re.findall (r” w+”, doc)} \rightarrow \text{what cost?} \)
in general re can be exponential time
\( \rightarrow \text{for char in doc:} \)
\( \text{if not alphanumeric} \)
\( \quad \text{add previous word} \quad \Theta(1) \)
\( \text{(if any) to list} \)
\( \text{start new word} \quad \Theta(|doc|) \)

(2) sort word list \( \leftarrow O(k \log k \cdot |\text{word}|) \) where \( k \) is #words
\( \text{for word in list:} \)
\( \text{if same as last word:} \quad \leftarrow O(|\text{word}|) \)
\( \quad \text{increment counter} \quad \Theta(1) \)
\( \text{else:} \)
\( \quad \text{add last word and count to list} \)
\( \quad \text{reset counter to 0} \quad \Theta(1) \)

(3) for word, count1 in doc1: \( \leftarrow \Theta(k_1) \)
\( \text{if word, count2 in doc2:} \leftarrow \Theta(k_2) \)
\( \text{total } += \text{ count1} * \text{ count2} \quad \Theta(1) \)

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

**Dictionary Approach**

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

(3)' as above \( \rightarrow O(|\text{doc}|) \text{ w.h.p.} \)
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
6.006 Introduction to Algorithms
Fall 2011

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