Lecture 2: Models of Computation

Lecture Overview

- What is an algorithm? What's a model of computation and why do we care?
- Random access machine
- Pointer machine
- Python model
- Document distance: problem & algorithms

What is an Algorithm?

- We know from the previous lecture: Mathematical abstraction of computer program + well specified computational procedure to solve a problem
- It's useful to visualize this abstraction as follows:

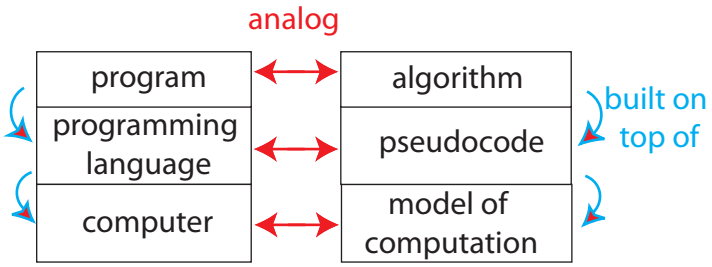

Figure 1: Algorithm

Model of computation specifies

- what operations an algorithm is allowed
- cost (time, space, ...) of each operation (how to measure 'time'? Wall-clock time? Usage statistics via profilers? More theoretical/abstract ways?)
- cost of algorithm = sum of operation costs
- We'll discuss two main styles/paradigms: random access machine, which is assembly-style, and pointer machine, which is more like an object oriented setup.

Random Access Machine (RAM)

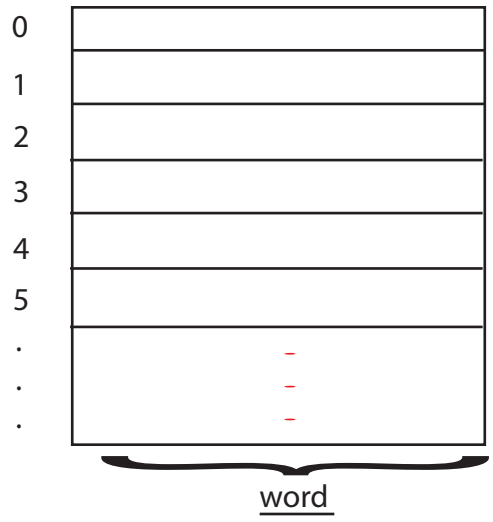

Figure 2: Random Access Machine

- Random Access Memory (also abbreviated RAM) modeled by a big array
- $\Theta(1)$ registers (each 1 word)
- In $\Theta(1)$ time, can
  - load word @ r_i in the array into register r_j (registers can store addresses/array indices)
  - compute $(+, -, *, /, \&, |, \hat{})$ on registers and compare
  - store the word in register r_j into memory @ r_i
- What's a word? Length of word $w \geq \lg(\text{memory size})$ bits to store address (address = array index)
  - assume basic objects (e.g., int) fit in word
- realistic and powerful $\rightarrow$ implement abstractions

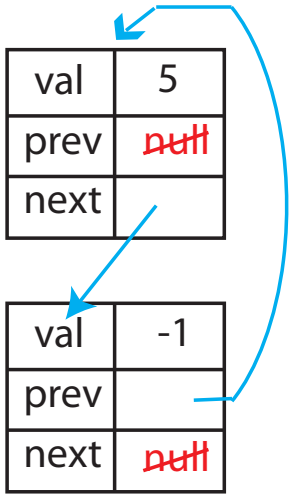

Figure 3: Pointer Machine

Pointer Machine

- dynamically allocated objects namedtuple
- object has $O(1)$ fields
- field = word (e.g., int) or pointer to object or null/none (a.k.a. reference – really, we're talking about references here, not pointers, i.e., you can't do pointer arithmetic on a pointer machine, all you can do is store and follow them)
- weaker than (can be implemented on) RAM

Python Model

Python lets you use either mode of thinking

1. "list" is actually an array → RAM

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

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

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

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

1. list

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

obvious if you think of infinite array

– but how would you do this with RAM?
via *table doubling* [Lecture 9]

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

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

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

Don't worry too much about edge cases like the "+1" – while it's important to be aware of these things, we'll make sure we emphasize this kind of thing when you need to be careful. In this case, it's especially important because the two lists could be empty. You also need extra O(1) time to check whether there are any elements in L1 and L2 (the x in L1/2 bit) even if they're of 0 length, but we're subsuming that into the 1.

(c) $L1$.extend($L2$) $\equiv$ for x in $L2$: } $\theta(1 + \lvert L_2\rvert)$ time
  $L1$.append(x) $\rightarrow \theta(1)$ }

Why the extra "1" in the "$1 + \lvert L2\rvert$" here? Once could argue it's just $\lvert L2\rvert$. However, we're putting the 1 here to include the time for checking whether there are any elements in L2 (as in the previous case).

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

(e) $b = x$ in L $\equiv$ for y in L: } $\theta(1)$ } $\theta(\text{index of } x) = \theta(\lvert L\rvert)$
& L.index(x) if $x == y$:
& L.find(x) $b = True$;
  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]
   D[key] = val
   key in D
   } $\theta(1)$ time w.h.p.
4. set: similar (think of as dict without vals)
5. heapq: heappush & heappop - via *heaps* [Lecture 4] $\rightarrow \theta(\log(n))$ 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:

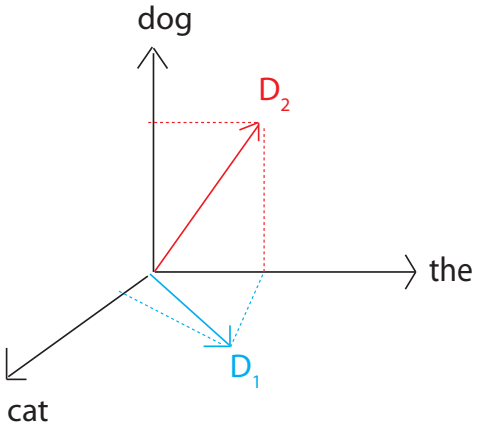

Figure 4: D_1 = "the cat", D_2 = "the dog"

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. Look at the formula above – bigger values indicate better matches. However, almost identical short documents will produce smaller values than longer documents with fewer words in common.
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)

Define:

n = number of words
$|word|$ = number of characters in $word$
$|doc|$ = number of characters in document = $\sum |word|$

Note that you'll need to do some of the stuff below twice (e.g., both documents need to be split into words) – we're just specifying them once with things like n – in real life, you'd need to do it with n_1 and n_2 for the two documents. The asymptotic values below are also only for what is specified; we'll need to add/double some of them. The ones with apostrophes represent alternative steps.

(1) Splitting:

```
→ for char in doc:                                  } Θ(|doc|)
      if not alphanumeric
         add previous word                          } Θ(1)
             (if any) to list
         start new word
```

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

```
for word in list:                                   } O(∑|word|) = O(|doc|)
    if same as last word:    ← O(|word|)
       increment counter                            } Θ(1)
    else:
       add last word and count to list
       reset counter to 0
```

(3)

```
for word, count1 in doc1:    ← Θ(n1)                } O(n1 · n2)
    if word, count2 in doc2: ← Θ(n2)
       total += count1 * count2   Θ(1)
```

(3)'

```
start at first word of each list                    } O(∑|word|) = O(|doc|)
if words equal:    ← O(|word|)
    total += count1 * count2
if word1 ≤ word2: ← O(|word|)
    advance list1
else:
    advance list2
repeat either until list done
```

Dictionary Approach

(2)' count = {}

for word in doc:
    if word in count: ← $\Theta(|word|) + \Theta(1)$
        count[word] += 1
    else
        count[word] = 1

$\Theta(1)$

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

w.h.p

(3)' as above → $O(|doc_1|)$ w.h.p.