# 강의 2: 계산 모델

## 강의 개요

- 알고리즘이란? 시간이란?
- 임의 접근 머신 RAM
- 포인터 머신
- 파이썬 모델
- 문서 거리: 문제 & 알고리즘

## 역사

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

- "대수학의 아버지"로, "완성과 균형 맞춤을 통한 계산에 대해 모든 것을 담은 책(The Compendious Book on Calculation by Completion & Balancing)" 저술
- 1차 & 2차 방정식 풀기: 최초의 알고리즘 중 일부

## 알고리즘이란?

- 컴퓨터 프로그램의 수학적 추상화
- 문제 해결을 위한 계산 과정

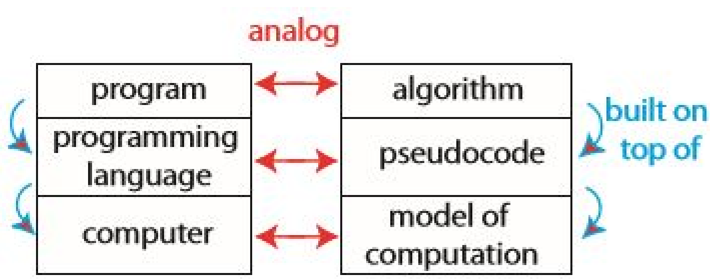
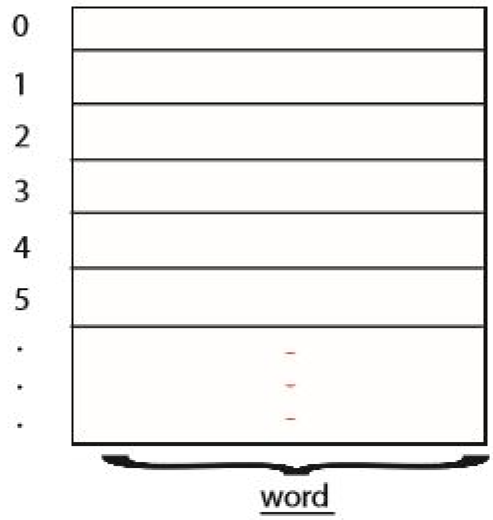



그림 1: 알고리즘

<u>계산 모델</u>이 규정하는 것

- 알고리즘이 할 수 있는 연산
- 각 연산의 비용 (시간, 공간, ...)
- 알고리즘의 비용 = 연산 비용의 합

## 임의 접근 머신 (RAM)




- 거대한 배열로 만들어진 임의 접근 머신 (RAM)
- Θ(1) 레지스터 (각 1개의 워드)
- Θ(1) 시간 안에 할 수 있는 일

– 레지스터 $r_i$ 에 있는 워드를 레지스터 $r_j$ 으로 불러오기

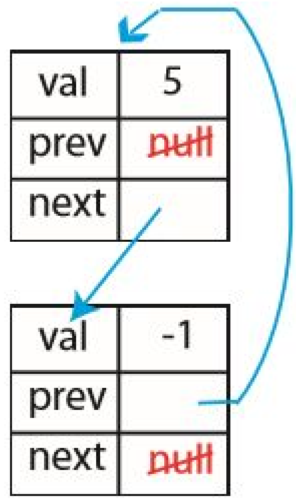
– 레지스터에서 (+, −, *, /, &, |, ^) 계산

– 레지스터 $r_j$ 를 $r_i$ 에 있는 메모리에 저장

- 워드란? $w \geq \log_2$ (memory size) bit
  - – 기본적인 객체(e.g., 정수)가 워드에 들어맞는다고 가정한다
  - – 4단원에서 큰 수를 다룬다
- 현실적이고 강력함 → 추상적 개념 구현

## 포인터 머신

- 동적 할당된 객체 (네임드 튜플)
- 객체는 O(1)개의 필드를 갖는다.
- 필드 = 워드 (e.g., 정수) 또는 객체/널을 가리키는 포인터 (a.k.a. 참조)
- RAM보다 약하다 (RAM으로 구현 가능)




## 파이썬 모델

파이썬은 두 가지 사고를 모두 적용할 수 있다.

1. “리스트”는 사실 배열이다. → RAM

   L[i] = L[j]+5 → Θ(1) 시간

2. O(1)개의 속성(참조 포함)을 가진 객체 → 포인터 머신

   x = x.next → Θ(1) 시간 → table doubling ?

파이썬에는 많은 연산이 있다. 각 연산의 비용을 알아보려면 (1)이나 (2)로 구현해보면 된다. :

1. <u>리스트</u>

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

   obvious if you think of infinite array

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

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

   $$\left.\begin{array}{l} \left.\begin{array}{l} \text{for } x \text{ in } L1: \\ \quad \text{L.append(x)} \to \theta(1) \end{array}\right\} \theta(|L_1|) \\ \left.\begin{array}{l} \text{for } x \text{ in } L2: \\ \quad \text{L.append(x)} \to \theta(1) \end{array}\right\} \theta(|L_2|) \end{array}\right\}$$

(c) $L1.\text{extend}(L2) \equiv \text{for } x \text{ in } L2:$ $\Big\}$ $\theta(1+|L_2|)$ time

$\equiv L1+=L2 \quad L1.\text{append(x)} \rightarrow \theta(1)$

(d) $L2 = L1[i:j] \equiv L2 = []$ $\Big\}$ $\theta(j-i+1) = O(|L|)$

for $k$ in range$(i, j)$:

$L2.\text{append}(L1[i]) \rightarrow \theta(1)$

(e) $b = x$ in $L$ $\equiv$ for $y$ in $L$: $\Bigg\}$ $\theta(\text{index of } x) = \theta(|L|)$

& L.index(x) if $x == y$: $\Big\}$ $\theta(1)$

& L.find(x) $b = True$;

break

else

$b = False$

(f) len(L) → θ(1) 시간 - 리스트는 자신의 길이를 필드에 저장한다

(g) L.sort() → θ(|L| log |L|) - 비교 정렬을 사용 [강의 3, 4 & 7] Compare Sort

2. <u>tuple, str</u>: similar, (think of as immutable lists)

3. <u>dict</u>: via *hashing* [Unit 3 = Lectures 8-10]

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

key in D

4. <u>set</u>: similar (think of as dict without vals)

5. <u>heapq</u>: heappush & heappop - via *heaps* [Lecture 4] → $\theta(\log(n))$ time

6. <u>long</u>: 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

## 문서 거리 문제 — d( $D_1$ $D_2$ ) 계산

문서 거리 문제는 유사한 문서 탐색이나 중복(위키피디아 미러 사이트와 구글)과 표절 발견, 그리고 웹 검색에 적용된다. ( $D_2$ = 질의어).

정의:

- <u>단어</u> = 영어와 숫자로 이루어진 문자열
- <u>문서</u> = 단어의 나열 (공백, 문장 부호 등 무시)

공통으로 갖는 단어를 통해 거리를 정의하고자 한다. 문서 D를 <u>벡터</u>로 생각한다. : D[w] = 단어 $W$의 등장 횟수. 예를 들면:

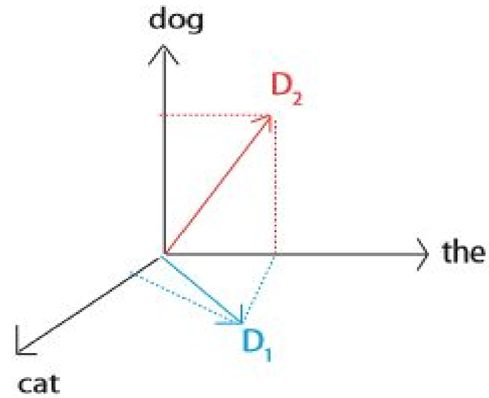



그림 2: $D_1$ = “the cat”, $D_2$ = “the dog”

첫 번째 시도로, 문서 거리를 다음과 같이 정의한다:

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

문제는 규모에 따라 변한다는 것이다. 즉, 99%의 단어가 일치하는 긴 문서들이 10%만 일치하는 짧은 문서들보다 거리가 멀게 된다.

이 문제는 단어의 개수로 정규화함으로써 해결할 수 있다:

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

여기서 |Di|는 문서 i에 들어있는 단어의 개수이다. 이것을 기하학적으로 해석(재구성)하면 다음과 같다. :

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

즉, 문서 거리는 두 벡터 사이의 각도이다. 0°는 두 문서가 같다는 것을 의미하고, 90°는 공통 단어가 없다는 것을 의미한다. 이 방법은 1975년에 Salton, Wong, Yang에 의해 소개되었다.

## 문서 거리 알고리즘

1. 각 문서를 단어들로 쪼갠다.
2. 단어 빈도를 센다. (문서 벡터)
3. 내적을 계산한다. (& 나눈다.)

(1) re.findall (r" w+", doc) $\rightarrow$ what cost?

in general re can be exponential time

$\rightarrow$ for char in doc: $\quad \Theta(|doc|)$
 if not alphanumeric
  add previous word $\quad \Theta(1)$
   (if any) to list
  start new word

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

for word in list: $\quad O(\sum |word|) = O(|doc|)$
 if same as last word: $\quad \leftarrow O(|word|)$
  increment counter $\quad \Theta(1)$
 else:
  add last word and count to list
  reset counter to 0

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

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

**Dictionary Approach**

(2)' count = {}

for word in doc: $\quad O(|doc|)$ w.h.p.
 if word in count: $\quad \leftarrow \Theta(|word|) + \Theta(1)$ w.h.p
  count[word] += 1 $\quad \Theta(1)$
 else
  count[word] = 1

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

코드 (웹 사이트의 lecture_code.zip & _data.zip)

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) (추가적인 정렬)
  words = words + words_on_line
- docdist2: 164.7 — words += words_on_line
- docdist3: 123.1 — (3)' . . . 삽입 정렬 이용
- docdist4: 71.7 — (2)' 이지만 (3)'의 정렬 이용
- docdist5: 18.3 — string.translate로 단어 분할
- docdist6: 11.5 — 병합 정렬 (vs. 삽입 정렬)
- docdist7: 1.8 — (3) (완전한 딕셔너리)
- docdist8: 0.2 — 한 줄 한 줄이 아닌 문서 전체