Outline: Hashing I
- Dictionaries & Python
- Motivation
- Hash functions
- Chaining
- Simple uniform hashing
- “Good” hash functions

Reading: CLRS 11.1, 11.2, 11.3

⇒ PS1 due tonight @ 11:59
⇒ PS2 out
**Dictionary problem:** Abstract Data Type (ADT) maintain set of items, each with a key, subject to
- `insert(item):` add item to set
- `delete(item):` remove item from set
- `search(key):` return item with key if it exists

- assume items have distinct keys (or that inserting new one clobbers old)

- balanced BSTs solve in $O(\log n)$ time per op. (in addition to inexact searches like nextlargest)

- **goal:** $O(1)$ time per operation

**Python dictionaries:** items are (key, value) pairs

e.g. \[ d = \{ 'algorithms': 5, 'cool': 42 \} \]

\[
\begin{align*}
d\text{.items()} & \rightarrow [(\text{‘algorithms’, 5}), (\text{‘cool’, 5})] \\
d[\text{‘cool’}] & \rightarrow 42 \\
d[42] & \rightarrow \text{KeyError} \\
\text{‘cool’ in d} & \rightarrow \text{True} \\
42 \text{ in d} & \rightarrow \text{False}
\end{align*}
\]

- **Python set** is really dict where items are keys
Motivation: Document Distance
- already used in
  def count_frequency(word-list):
    D = {}
    for word in word-list:
      if word in D:
        D[word] += 1
      else:
        D[word] = 1
- new docdist? uses dictionaries instead of sorting:
  def inner_product(D1, D2):
    sum = 0
    for key in D1:
      if key in D2:
        sum += D1[key] * D2[key]
  ⇒ optimal $\Theta(n)$ document distance
  assuming dictionary ops. take $O(1)$ time

Motivation: PS2
  How close is chimp DNA to human DNA?
  = Longest common substring of two strings
    e.g. ALGORITHM vs. ARITHMETIC
  - dictionaries help speed algorithms
    e.g. put all substrings into set, looking for duplicates
    $\Rightarrow \Theta(n^2)$
How do we solve the dictionary problem?

**Simple approach**: Direct-access table
- Store items in an array, indexed by key
- Problems:
  1. Keys must be nonnegative integers (or, using two arrays, integers)
  2. Large key range \( \Rightarrow \) large space
     e.g. one key of \( 2^{256} \) is bad news

**Solution**: Map key space to integers
- In Python: `hash(object)` where object is a number, string, tuple, etc., or object implementing `__hash__`
- Misnomer: should be called "prehash"
- Ideally, \( x = y \iff \text{hash}(x) = \text{hash}(y) \)
- Python applies some heuristics
  e.g. `hash('\0B') = 64 = hash('\0\0C')`
- Object’s key should not change while in table
  (else can’t find it anymore)
- No mutable objects like lists
Solution 2: hashing (verb from ‘hache’=hatchet, Germanic)
- reduce universe $U$ of all keys (say, integers) down to reasonable size $m$ for table
- idea: $m \approx n$, $n = |K|$, $K$=keys in dictionary
- hash function $h: U \rightarrow \{0, 1, ..., m-1\}$

- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions? We'll see two ways
- chaining: TODAY
- open addressing: NEXT WEEK

Chaining: linked list of colliding elements in each slot of table

- search must go through whole list $T[h(\text{key})]$,
- worst case: all keys in $K$ hash to same slot
  $\Rightarrow \Theta(n)$ per operation
Simple uniform hashing: an assumption:
each key is equally likely to be hashed
to any slot of table, independent of
where other keys are hashed
- let n = #keys stored in table
  m = #slots in table
- load factor $\alpha = \frac{n}{m}$
  = average # keys per slot

Expected performance of chaining: assuming $\mathcal{O}(1 + \alpha)$
- apply hash function & access slot
- search the list
[actually $\Theta(1 + \alpha)$, even for successful search, see CLRS]

$= \mathcal{O}(1)$ if $\alpha = \mathcal{O}(1)$ i.e. $m = \Omega(n)$
Hash functions:

**Division method:** \( h(k) = k \mod m \)
- \( k_1 \) & \( k_2 \) collide when \( k_1 \equiv k_2 \pmod{m} \)
  i.e. when \( m \) divides \( |k_1 - k_2| \)
- fine if keys you store are uniform random
- but if keys are \( x, 2x, 3x, \ldots \) (regularity)
  and \( x \) & \( m \) have common divisor \( d \)
  then use only \( 1/d \) of table
- likely if \( m \) has a small divisor e.g. \( 2 \)
- if \( m = 2^r \) then only look at \( r \) bits of key!

\[ \text{- Good practice: } m \text{ is a prime} \]
& not close to power of 2 or 10
(to avoid common regularities in keys)
\[ \text{- Inconvenient to find prime: division slow} \]

**Multiplication method:** \( h(k) = [(a \cdot k) \mod 2^w] \gg (w-r) \)
- where \( m = 2^r \) & \( w \)-bit machine words
  & \( a = \) odd integer between \( 2^{w-1} \) & \( 2^w \)
- \text{Good practice: } a \text{ not too close to } 2^{w-1} \text{ or } 2^w
- faster: multiplication & bit extraction