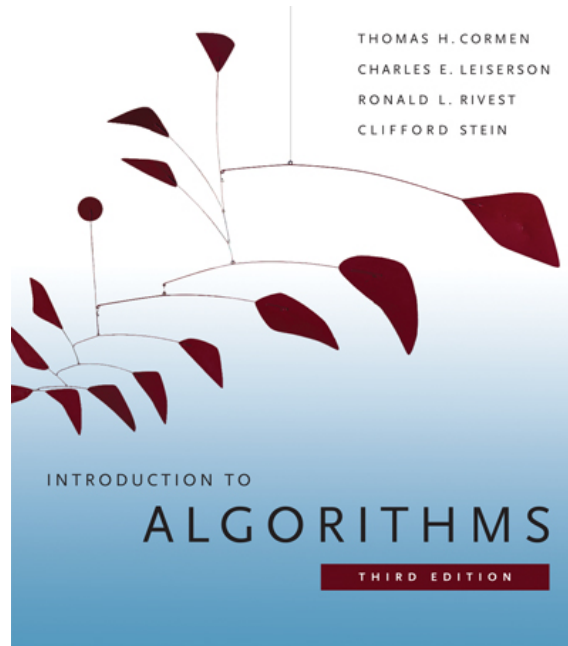


# 6.006- *Introduction to Algorithms*



## *Lecture 6*

**Alan Deckelbaum**

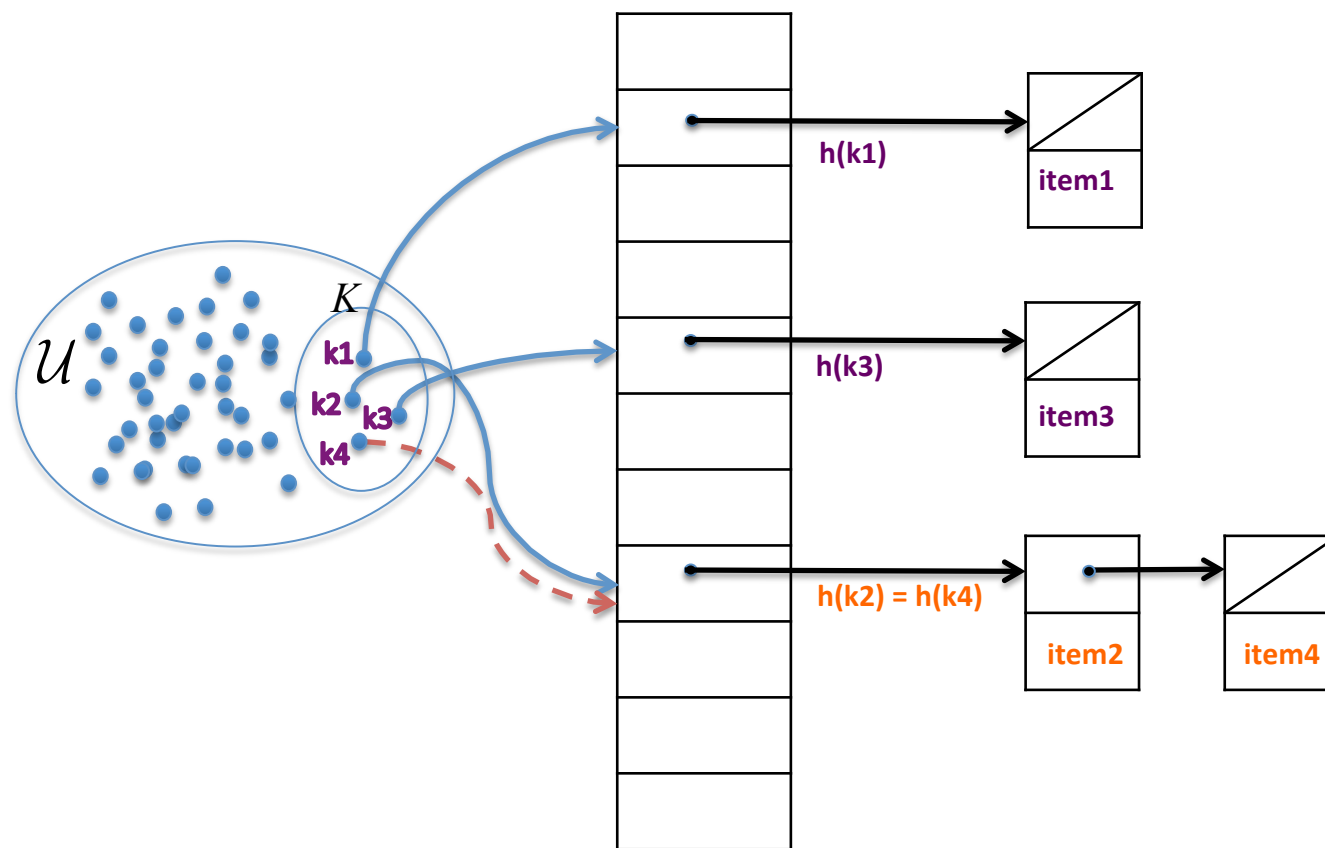
**CLRS: Chapter 17 and 32.2.**

**LAST TIME...**

# Dictionaries, Hash Tables

- **Dictionary:** Insert, Delete, Find a key
  - can associate a whole item with each key
- **Hash table**
  - implements a dictionary, by spreading items over an array
  - uses *hash function*
    - h: Universe of keys (**huge**) → Buckets (**small**)
  - *Collisions*: Multiple items may fall in same bucket
  - *Chaining Solution*: Place colliding items in linked list, then scan to search
- **Simple Uniform Hashing Assumption (SUHA):**
  - h is “random”, uniform on buckets
  - Hashing  $n$  items into  $m$  buckets → expected “load” per bucket:  $n/m$
  - If chaining used, expected search time  $O(1 + n/m)$

# Hash Table with Chaining



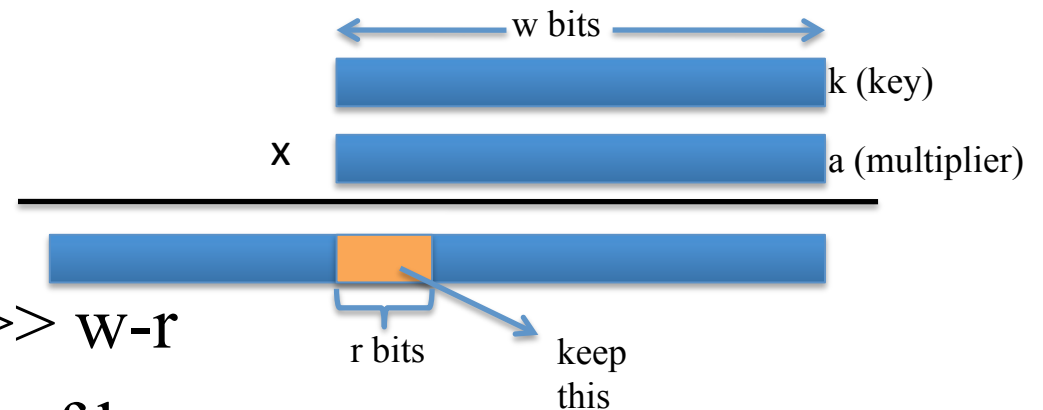
$\mathcal{U}$  : universe of all possible keys-huge set

$K$  : actual keys-small set, but not known when  
designing data structure

# Hash Functions?

- Division hash
  - $h(k) = k \bmod m$
  - Fast if  $m$  is a power of 2, slow otherwise
  - Bad if e.g. keys are regular

- Multiplication hash
  - $a$  an odd integer
  - $h(k) = (a \cdot k \bmod 2^w) \gg w-r$
  - Better on regular sets of keys



# Non-numbers?

- What if we want to hash e.g. strings?
- Any data is bits, and bits are a number
- E.g., strings:
  - Letters a..z can be “digits” base 26.
  - “the” =  $t \cdot (26)^2 + h \cdot (26) + e$   
 $= 19 \cdot (676) + 8 \cdot (26) + 5$   
 $= 334157$
- Note: hash time is length of string, not  $O(1)$   
(wait a few slides)

# Longest Common Substring

- Strings S,T of length n, want to find longest common substring
- Algorithms from last time:  
 $O(n^4) \rightarrow O(n^3 \log n) \rightarrow O(n^2 \log n)$
- Winner algorithm used a hash table of size n:

Binary search on maximum match length L; to check if a length works:

- Insert all length-L substrings of S in hash table
- For each length-L substring x of T
  - Look in bucket  $h(x)$  to see if x is in S

# Runtime Analysis

- Binary search cost:  $O(\log n)$  length values  $L$  tested
- For each length value  $L$ , here are the costly operations:
  - Inserting all  $L$ -length substrings of  $S$ :  $n-L$  hashes
    - Each hash takes  $L$  time, so total work  $\Theta((n-L)L)=O(n^2)$
  - Hashing all  $L$ -length substrings of  $T$ :  $n-L$  hashes
    - another  $O(n^2)$
  - Time for comparing substrings of  $T$  to substrings of  $S$ :
    - How many comparisons?
    - Under SUHA, each substring of  $T$  is compared to an expected  $O(1)$  of substrings of  $S$  found in its bucket
    - Each comparison takes  $O(L)$
    - Hence, time for all comparisons:  $\Theta(nL)=O(n^2)$
- So  $O(n^2)$  work for each length
- Hence  $O(n^2 \log n)$  including binary search



# Faster?

- Amdahl's law: if one part of the code takes 20% of the time, then no matter how much you improve it, you only get 20% speedup
- Corollary: must improve **all** asymptotically worst parts to change asymptotic runtime
- In our case
  - Must compute sequence of  $n$  hashes faster
  - Must reduce cost of comparing in bucket

**FASTER COMPARISON**

# Faster Comparison

- **First Idea:** when we find a match for some length, we can stop and go to the next value of length in our binary search.
- **But,** the real problem is “false positives”
  - Strings in same bucket that don’t match, but we waste time on
- **Analysis:**
  - $n$  substrings to size- $n$  table: average load **1**
  - SUHA: for every substring  $x$  of  $T$ , there is **1** other string in  $x$ ’s bucket (in expectation)
  - Comparison work:  **$L$**  per string (in expectation)
  - So total work for all strings of  $T$ :  **$nL = O(n^2)$**

# Solution: Bigger table!

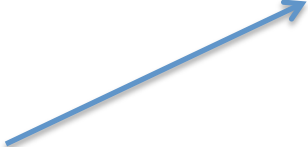
- What size?
- Table size  $m = n^2$ 
  - $n$  substrings to size- $m$  table: average load  $1/n$
  - SUHA: for every substring  $x$  of  $T$ , there is  $1/n$  other strings in  $x$ 's bucket (in expectation)
  - Comparison work:  $L/n$  per string (in expectation)
  - So total work for all strings of  $T$ :  $n(L/n) = L = O(n)$
- Downside?
  - Bigger table
  - ( $n^2$  isn't realistic for large  $n$ )

# Signatures

- Note  $n^2$  table isn't needed for fast lookup
  - Size  $n$  enough for that
  - $n^2$  is to reduce cost of false positive compares
- So don't bother making the  $n^2$  table
  - Just compute for each string another hash **value** in the larger range  $1..n^2$
  - Called a **signature**
  - If two signatures differ, strings differ
  - $\text{Pr}[\text{same sig for two different strings}] = 1/n^2$ 
    - (simple uniform hashing)

# Application

- Hash substrings to size  $n$  table
- But store a signature with each substring
  - Using a second hash function to  $[1..n^2]$
- Check each T-string against its bucket
  - First check signature, if match then compare strings
  - Signature is a small number, so comparing them is  $O(1)$



strictly speaking  $O(\log n)$ ; but if  $n^2 < 2^{32}$  the signature fits inside a word of the computer; in this case, the comparison takes  $O(1)$

# Application

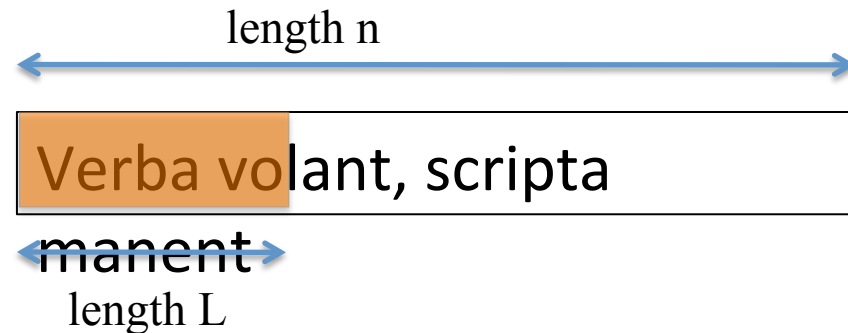
- Runtime Analysis:
    - for each T-string:
      - $O(\text{bucket size}) = O(1)$  work to compare **signatures**;
    - so overall  $O(n)$  time in signature comparisons
    - Time spent in string comparisons?
      - $L \times (\text{Expected Total Number of False-Signature Collisions})$ 
        - $n$  out of the  $n^2$  values in  $[1..n^2]$  are used by S-strings
        - so probability of a T-string signature-colliding with some S-string:  $n/n^2$
        - hence total expected number of collisions 1
  - so total time spent in String Comparisons is  $L$
- fine print:** we didn't take into account the time needed to compute signatures; we can compute all signatures in  $O(n)$  time using trick described next...

**FASTER HASHING**



# Rolling Hash

- We make a sequence of  $n$  substring hashes
  - Substring lengths  $L$
  - Total time  $O(nL) = O(n^2)$
- Can we do better?
  - For our particular application, yes!



# Rolling Hash Idea

- e.g. hash all 3-substrings of “there”
- Recall division hash:  $x \bmod m$
- Recall string to number:
  - First substring “the” =  $t \cdot (26)^2 + h \cdot (26) + e$
- If we have “the”, can we compute “her”?

$$\begin{aligned} \text{“her”} &= h \cdot (26)^2 + e \cdot (26) + r \\ &= 26 \cdot (h \cdot (26) + e) + r \\ &= 26 \cdot (t \cdot (26)^2 + h \cdot (26) + e - t \cdot (26)^2) + r \\ &= 26 \cdot (\text{“the”} - t \cdot (26)^2) + r \end{aligned}$$

- i.e. subtract first letter’s contribution to number, shift, and add last letter

# General rule

- Strings = base-b numbers

- Current substring  $S[i \dots i+L-1]$

$$\begin{array}{r}
 S[i] \cdot b^{L-1} + S[i+1] \cdot b^{L-2} + S[i+2] \cdot b^{L-3} \dots + S[i+L-1] \\
 - S[i] \cdot b^{L-1} \\
 \hline
 S[i+1] \cdot b^{L-2} + S[i+2] \cdot b^{L-3} \dots + S[i+L-1] \\
 \times b \\
 \hline
 S[i+1] \cdot b^{L-1} + S[i+2] \cdot b^{L-2} \dots + S[i+L-1] \cdot b \\
 + \qquad \qquad \qquad S[i+L] \\
 \hline
 S[i+1] \cdot b^{L-1} + S[i+2] \cdot b^{L-2} \dots + S[i+L-1] \cdot b + S[i+L] \\
 = S[i+1 \dots i+L]
 \end{array}$$

# Mod Magic 1

- So:  $S[i+1 \dots i+L] = b S[i \dots i+L-1] - b^L S[i] + S[i+L]$
- where
$$S[i \dots i+L-1] = S[i] \cdot b^{L-1} + S[i+1] \cdot b^{L-2} + \dots + S[i+L-1] (*)$$
- **But**  $S[i \dots i+L-1]$  may be a huge number (so huge that we may not even be able to store in the computer, e.g.  $L=50$ ,  $b=26$ )
- **Solution** only keep its *division hash*:  $S[\dots] \bmod m$
- This can be computed without computing  $S[\dots]$ , using **mod magic!**
- Recall:  $(ab) \bmod m = (a \bmod m) (b \bmod m) (\bmod m)$   
 $(a+b) \bmod m = (a \bmod m) + (b \bmod m) (\bmod m)$
- With a clever parenthesization of (\*):  $O(L)$  to hash string!

# Mod Magic 2

- Recall:  $S[i+1 \dots i+L] = b S[i \dots i+L-1] - b^L S[i] + S[i+L]$
- Say we have hash of  $S[i \dots i+L-1]$ , can we still compute hash of  $S[i+1 \dots i+L]$  ?
- Still **mod magic** to the rescue!
- Job done in  $O(1)$  operations, if we know  $b^L \bmod m$



Computing  $n-L$  hashes costs  $O(n)$

$O(L)$  time for the first hash

+  $O(L)$  to compute  $b^L \bmod m$

+  $O(1)$  for each additional hash

# Summary

- Reduced compare cost to  $O(n)/\text{length}$ 
  - By using a big hash table
  - Or signatures in a small table
- Reduced hash computation to  $O(n)/\text{length}$ 
  - Rolling hash function
- Total cost of phases:  $O(n \log n)$
- Not the end: suffix tree achieves  $O(n)$