| Admin: |
| Quiz 1 | Wed 10/16 at 7:30 pm |
| in 32-124, 32-144, 34-101 (see website) |
| May bring one 8½ x 11 “cheat sheet” (else closed book) |
| Coverage through L10 (Hashing III) |

| Outline: |
| ☐ Universal hashing | 11.3.3 |
| ☐ Table Doubling | 17.4 |
| ☐ Rolling Hash (for string matching) | 32.2 |

Readings: Chap 10: 11.1-11.3, 17.4, 32.2
Universal Hashing

Recall that a family $\mathcal{H} = \{h\}_{i=1}^{m}$ of hash functions mapping $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$ is universal if for each pair $k, l$ of distinct keys in $\mathcal{U}$ the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ [i.e. causing $k$ & $l$ to collide] is at most $|\mathcal{H}|/m$.

That is $\Pr[h(k)=h(l)] \leq 1/m$.

We can pick $h$ randomly from $\mathcal{H}$ at start of execution, & get good average-case behavior for all input sets!

Thm: Universal families of hash functions exist.

(e.g. $h_{ab}(k) = (ak+b) \mod p \mod m$; Thm 11.5)
Expected time to do search with universal hash fn:

= expected time to look up a given key k
= Θ(1 + expected # keys in k's list)
= Θ(1 + expected # keys that collide with k)
= Θ(1 + (n-1)/m)  

n = # keys in table

Since each key collides with prob. 1/m

& because of linearity of expectation

= Θ(1 + α)

α = n/m

= Θ(1) if m = Θ(n)

Constant (expected) time !

⇒ How to ensure m = Θ(n)?
**Table Doublings**

Suppose we start with empty table, & keep inserting elements (and doing searches, but no deletes).
Load factor keeps increasing 😞

**Idea:** When \( \alpha \) reaches 1:
- allocate new table of \( 2 \times \) size
- rehash all elements into new table
- discard old table

(now \( \alpha = \frac{1}{2} \))

Equivalently:
- double table size when \( m = 2, 4, 8, 16, \ldots \)

What about all this re-hashing work?
- can be a lot (\( \Theta(n) \)) when it happens
- but is small when averaged over all operations, since you need to do \( \Theta(n) \) insertions before you get to point where \( \Theta(n) \) rehashing work is needed.
Chapter 17 is all about "amortized analysis" - the cost of an expensive operation can be "amortized" over other operations within a sequence of operations.

\[
\text{Cost of doing } n \text{ inserts (total)} = \frac{\text{cost of inserts only}}{n} + \frac{\text{cost of table doublings}}{n}
\]

Cost of table doublings

\[
\leq 2 + 4 + 8 + \ldots + n + 2n \\
\text{(size of final table is } \leq 2n) \\
\leq 4n \\
\text{(assume re-hashing into table of size } n \text{ costs } n)
\]

So total cost (including rehashing) of doing \( n \) inserts

\[
\leq 5n
\]

So "average" cost per insert is still \( \Theta(1) \), even when we include cost of table doublings.

[Deletes can also be handled - (see recitation).]
"Rolling Hash"  

Cut application of hash fns for pattern matching  

(Uses hash fns but not hash tables)  

Problem: Find occurrence of "algorithm" in (long text...)  

Text = $T[1..n] = \text{"Before there were computers,..."}$  

[CLRS, Preface]  

Pattern = $P[1..m] = \text{"algorithm"}$  

Return first $s$ (shift) s.t.  

$$T[s+1..s+m] = P[1..m]$$  

$(0 \leq s \leq n-m)$  

(m-character match)  

Naive algorithm takes time $O((n-m)m) = O(nm)$  

Can we do better? Yes [ref Chapter 32]  

(Many ways to do so; here we look at "Robin-Karp" algorithm,  

based on hash fns...)  

Idea: Have a "sliding window" of size m characters  

So $T[s+1..s+m]$ is "visible"  

Do a "quick check" to see if this is a "possible  

matching by comparing  

$$h(T[s+1..s+m]) \text{ to } h(P[1..m])$$
Need quick way to update hash value when we slide window one position to right:

\[ T[s+1\ldots s+m] = \ldots [0\ C\ A] \]

move to \( T[s+2\ldots s+m+1] = \ldots [C\ A\ B] \)

leaves window

joins window

Idea: consider window contents as \( m \)-digit number (base \( B \), for suitable \( B \))

\[ x = d_1 \ldots d_m \implies \sum_{i=1}^{m} d_i \cdot B^{m-i} = x \]

To remove high-order digit

\[ x \leftarrow x - d_1 \cdot B^{m-1} \pmod{q} \]

To "shift left"

\[ x \leftarrow x \cdot B \pmod{q} \]

To specify new low-order digit \( l \) (after shift)

\[ x \leftarrow x + l \pmod{q} \]

To keep sizes of \( #'s \) reasonable, do all this modulo \( q \) for some large prime \( q \).
If hash at window matches hash at pattern, then must verify match (time $m$), because there could have been collision (with hash $(p)$). We expect about $n/g$ such "spurious hits".

\[
\text{Time} = m \quad \text{initial hash}
\]
\[
+ (n-m) \quad \text{sliding window}
\]
\[
+ m \cdot (n/g) \quad \text{handle spurious hits}
\]
\[
+ m \cdot v \quad \text{for } v \text{ valid hits}
\]
\[
= O(n + m \cdot v)
\]
\[
= O(n) \quad \text{if } q_f \geq m
\]
\[
= O(n) \quad \text{if } v = O(1), \text{ or just want first hit.}
\]

Much better than $O(n \cdot m)$ naive alg.