Admin: → PSet 2 due tonight!
    → PSet 3 out
    → Looking at problems 3-1 to 3-4
        before the quiz is a good review
    → No Part B this time
        → Quiz 1 next week!
        See handout on Stellar
    → Review recitations on Wed
    → NO LECTURE on Thur
    → "Normal" recitations on Fri
    → Practice on past quizzes
        courses.csail.mit.edu/6.006
    → Cookies today! 😊

Today: → Table resizing
    → Amortized analysis
    → String matching & Karp-Rabin alg.
    → Rolling hash

Reading: CLRS Ch 17
Last time: Hashing with chaining:

- All possible keys
- Hash function
- Colliding items
- Expected chain length
- \( \lambda = \frac{m}{m} \)
- Load factor

\( \Theta(l + \lambda) \)

Important: This assumes that:
- \( h \) satisfies Simple Uniform Hashing Assumption (or Universal Hashing)
- \( h \) takes \( O(1) \) time to compute

"Good" hash functions:
- Division method: \( h(k) = k \mod m \)
- Multiplication method:
  \( h(k) = [(a \cdot k) \mod 2^u] >> (u-1) \)
- Universal hashing: random \( m = 2^r \)
- Can be broken by certain sets of keys

Provably good for any set of keys: \( h(k) = [(a \cdot k + b) \mod p] \mod m \)
How large should the table be?

- Want \( m = \Theta(n) \) at all times
- But: we don't know at first what \( n \) will be!
- Our guess on \( m \) too small \( \Rightarrow \) slow (lots of collisions)
- \( m \) too large \( \Rightarrow \) wasteful on space

Key idea: Start with table being small (e.g., \( m = 1 \))
- grow (\& shrink) as necessary

How to grow/shrink a hash table?

Rehashing: \( \Rightarrow \) Choose the new size \( m' \)
\& new hash function \( h' \)
\( \Rightarrow \) Build new hash table from scratch
insert each item from the old table into the new one (\& discard old one when done)

\( \Rightarrow \) Time needed:
\[ \Theta(n+m) = \Theta(n), \text{ if } m = \Theta(n) \]
(This is NOT \( O(1) \) time!)
How fast to grow the table?

Assume \( n \) reaches \( m \) (after an insert)

\[ \rightarrow m = m + 1 \?
\]

\[ \Rightarrow \text{could need to rehash every step} \]

\[ \Rightarrow m \text{ inserts cost } \Theta(1+2+3+\ldots+n) = \Theta(n^2) \gg n! \]

(Table doubling)

\[ \rightarrow m \times 2 ? \quad \text{(Note: still } m = \Theta(n) \text{)} \]

\[ \Rightarrow \text{rehash at each } 2^i \text{-th insertion} \]

\[ \Rightarrow m \text{ inserts cost } \Theta(1 + 2 + 4 + 8 + \ldots + n) = \Theta(n) \]

(Similar in spirit to build-heap [L4])

Result: A few inserts cost \( \Omega(n) \) time

but \( O(1) \) "on average"

Broader concept: Amortized analysis

\[ \rightarrow \text{Common technique in DSs. Think: like paying rent:} \]

\[ \rightarrow \text{Operation has amortized cost } T(n) \quad $1500/\text{month} \approx $50/\text{day} \]

\[ \rightarrow \forall k \geq 1, k \text{ operations cost } \leq k \cdot T(n) \]

\[ \Rightarrow "T(n) \text{ amortized } \approx "T(n) \text{ on average}" \text{, but averaged over all ops.} \]

Note: Some of these ops might still take long time (but most don't!)
Back to hashing:

- Can maintain $m = \Theta(n)$ under insertions in $\Theta(1)$ amortized time

$\Rightarrow \alpha = O(1) \Rightarrow$ search in $O(1)$ expected time (under SUHA/univ. hashing + $O(1)$ time eval. of $h$)

Delete operations?

$\Rightarrow$ Already $O(1)$ expected time as is

$\Rightarrow$ But: $m$ can get big w.r.t. $n$ - wasteful!

- e.g.: $m \times$ insert, $m \times$ delete

$\Rightarrow$ Solution: When $n$ decreases to $\left\lfloor \frac{m}{4} \right\rfloor$ shrink to half the size

$\Rightarrow$ $O(1)$ amortized cost to maintain $m = \Theta(n)$ for both inserts & deletions

But: Analysis harder $\Rightarrow$ see CLRS 17.4

Sidenote: can make everything $O(1)$ time per op.
im an non-amortized sense, but a bit tricky

Python: The same approach gives us resizable lists (arrays) in Python

$\Rightarrow$ list.append & list.pop in $O(1)$ amortized time
String matching problem: \( s \rightarrow t \)

Given two strings \( S[1...s] \) & \( T[1...t] \), does \( S \) occur as a substring of \( T \)?

In other words: Does there exist a shift \( k \) \((0 \leq k \leq t-s)\)

s.t. \( T[k+1...k+s] = S[1...s] \)? (s-character match)

Think: \( S \leftarrow \text{pattern} \quad T \leftarrow \text{text} \)

E.g. \( S = '6.006' \) & \( T = \) your entire INBOX

Naive algorithm:

Slide a length-\( s \) "window" through \( T \), checking all \((t-s)\) shifts for a potential match with \( S \)

Running time? \( O((t-s) \cdot s) = O(s \cdot t) \)
Can we do better?

Key bottleneck: Need to check for a match each time

**Karp–Rabin algorithm:**

Idea: Quickly "pre-screen" potential matches by comparing hashes, i.e.,

check if \( h(T[k+1...k+s]) = h(S[1...s]) \)

Note: If hashes are different \( \Rightarrow \) no match
but if hashes are equal \( \Rightarrow \) there still might be "false positives"

due to collisions

\( \Rightarrow \) If hashes match, still need to verify for match

\( \Rightarrow \) But: probability \( p \) of "false positive" will be small for a suitable choice of \( h \)

**Running time?**

\( O((t + (1 + p \cdot s) + s)) = O(t) \) if \( p \leq \frac{4}{5} \)

- \( t \) computing & comparing hashes
- \( 1 + p \cdot s \) dealing with false positives
- checking "final" match
Unfortunately: Above analysis cheats/abuses the model

Indication something is "fishy": If we take \( h \) to be a "trivial" identity hash function \( h(x) = x \)
- Then: \( \rightarrow \) prob. of false positive is 0
- \( \rightarrow \) K-R alg. becomes the naïve alg.
- \( \rightarrow \) BUT: \( (*) \) would still indicate \( O(\cdot) \) run time (instead of \( O(s^2) \))

Root of the problem: Can't really assume that computing \( h(T[k+1..s]) \)
- takes \( O(1) \) time

Note: Size of \( T[k+1..s] \) is \( s \) \( < \) a const.

\[ \Rightarrow \text{Computing } h(T[k+1...s]) \text{ should take } O(s) \text{ time} \]

Resulting "real" running time is
\[ O\left(+(s+p.s)+s\right) = O(ts) \]

"real" time needed to compute & compare hashes

No improvement!
What can we do now?

- Use a hash function that can leverage the fact that every two consecutive sliding windows differ only by two characters.

**Rolling hash:** Provides a quick way to update hash value when we slide the window by one position:

Given \( h(T[k+1..k+5]) \) compute \( h(T[k+2..k+5+1]) \) in \( O(1) \) time.

How to do that?

Next idea: **Algebraization**

- Encode the window as a \( s \)-digit number (base \( B \), for \( B \geq \text{alphabet size} \))

\[
x = d_1 \, d_2 \ldots d_s \rightarrow \sum_{i=1}^{s} d_i \, B^{s-i} \pmod{q}
\]

- To keep sizes of \( #s \) reasonable use \( \text{mod} \, q \)

- As long as \( q \) is some random prime \( \approx S \) (one needs to prove this) \( \Rightarrow \) prob. \( p \) of false positives \( \leq \frac{1}{S} \)
Now:

$y = h(x) = \frac{s}{i = 1} d_i B^{s-i}$

$x = d_1 \ldots d_s$

$\Rightarrow$ To remove first character

$x = 0d_2 \ldots d_s$

$y \leftarrow y - d_1 B^{s-1} \pmod{q}$

$\Rightarrow$ To shift everything left

$x = d_2 \ldots d_s 0$

$y \leftarrow y \cdot B \pmod{q}$

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

$x = d_2 \ldots d_s d_{s+1}$

$y \leftarrow y + 1 \pmod{q}$

$\Rightarrow$ Updating a hash after a shift $O(1)$ time

Resulting running time of Karp-Rabin alg.:

$O(s + (t-s)(1+p_s)) = O(t)$!