DG WEEK 7

STRING HASHING

WHY HASH STRINGS?

- Faster string comparison
  - Comparing two strings **A** and **B** naively takes $O(\min(|A|, |B|))$ time. *Why?*
  - Using a hashing function h compare the hashes $h(A)$ and $h(B)$ in $O(1)$ time!

A POOR HASH FUNCTION

- For a string $S = \{s_i\}_{i=0}^{n-1}$ the hash function h is defined as

$$h(S) = \sum_{i=0}^{n-1} s_i$$

- Problem?

A POOR HASH FUNCTION

- For a string $S = \{s_i\}_{i=0}^{n-1}$ the hash function h is defined as

$$h(S) = \sum_{i=0}^{n-1} s_i$$

- Problem?
- Strings with same characters get hashed to the same value irrespective of order.
  - abc = 1+2+3 = 6 = 2+3+1 = bca

EXAMPLE (GOOD) HASH FUNCTION

- For a string $S = \{s_i\}_{i=0}^{n-1}$ the hash function h is defined as

$$h(S) = \sum_{i=0}^{n-1} s_i \cdot p^i \mod m$$

- p is generally chosen to be a prime number greater than the alphabet size while m is chosen to be large prime number.
- Example: $p = 31$ and $m = 10^9 + 9$ when hashing lowercase English strings (alphabet size = 26).

EXAMPLE HASHES

- data
  - $4 \times 31^0 + 1 \times 31^1 + 20 \times 31^2 + 1 \times 31^3 = 49046$
- algo
  - 453965
- Java uses this hash function for strings with $p = 31$. (Nice discussion on why 31 is used)

IT'S A ROLLING HASH

- Makes it easy to compute hashes of substrings.
- abcde

Can we use this info to compute the hash of substrings? (*say* bc)

a	ab	abc	abcd	abcde
1	63	2946	122110	4739715

IT'S A ROLLING HASH

- Makes it easy to compute hashes of substrings.
- abcde

Can we use this info to compute the hash of substrings? (*say* bc)

a	ab	abc	abcd	abcde
1	63	2946	122110	4739715

- $h(a) = 1 \times 31^0$

  $h(ab) = 1 \times 31^0 + 2 \times 31^1$

  $h(abc) = 1 \times 31^0 + 2 \times 31^1 + 3 \times 31^2$

  $h(bc) = 2 \times 31^0 + 3 \times 31^1$

$$h(bc) = (h(abc) - h(a)) \times 31^{-1}$$

APPLICATION

- ***Problem***: Given a string **S** and a pattern **p**, find all occurrences of the pattern (if it exists) in **S**. Let $|S| = n$ and $|p| = m$.

APPLICATION

- ***Problem***: Given a string **S** and a pattern **p**, find all occurrences of the pattern (if it exists) in **S**. Let $|S| = n$ and $|p| = m$.
- *Naive approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
  - How many size m substrings?
  - How much time does each naive comparison take?
  - Total time?

APPLICATION

- ***Problem***: Given a string **S** and a pattern **p**, find all occurrences of the pattern (if it exists) in **S**. Let $|S| = n$ and $|p| = m$.
- *Naive approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
  - How many size m substrings? $n - m + 1$
  - How much time does each naive comparison take? $O(m)$
  - Total time? $O(mn)$

APPLICATION

- ***Problem***: Given a string **S** and a pattern **p**, find all occurrences of the pattern (if it exists) in **S**. Let $|S| = n$ and $|p| = m$.
- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
  - How many size *m* substrings?
  - Time taken to compute hash of **p**?
  - How much time does each hash-based comparison take?
  - Total time?

APPLICATION

- ***Problem***: Given a string **S** and a pattern **p**, find all occurrences of the pattern (if it exists) in **S**. Let $|S| = n$ and $|p| = m$.
- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
  - How many size *m* substrings? $n - m + 1$
  - Time taken to compute hash of **p**? $O(m)$
  - How much time does each hash-based comparison take? $O(1)$
  - Total time? $O(m + n)$

APPLICATION

- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
- Are we missing something? (What about the time taken to compute hashes of substrings?)

APPLICATION

- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
- Are we missing something? (What about the time taken to compute hashes of substrings?) Can be done in constant time with a rolling hash.

APPLICATION

- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
- Are we missing something? (What about the time taken to compute hashes of substrings?) Can be done in constant time with a rolling hash.
- Still missing something? (What if there are collisions?)

APPLICATION

- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
- Are we missing something? (What about the time taken to compute hashes of substrings?) Can be done in constant time with a rolling hash.
- Still missing something? (What if there are collisions?) We do an exact comparison whenever the hashes match.

APPLICATION

- *Hashing Approach*: Loop through all size *m* substrings of **S** and match each string with **p**.
- Are we missing something? (What about the time taken to compute hashes of substrings?) Can be done in constant time with a rolling hash.
- Still missing something? (What if there are collisions?) We do an exact comparison whenever the hashes match.
- This algorithm for string search is known as Rabin-Karp algorithm.