Advanced Computer Contest Preparation

Lecture 31

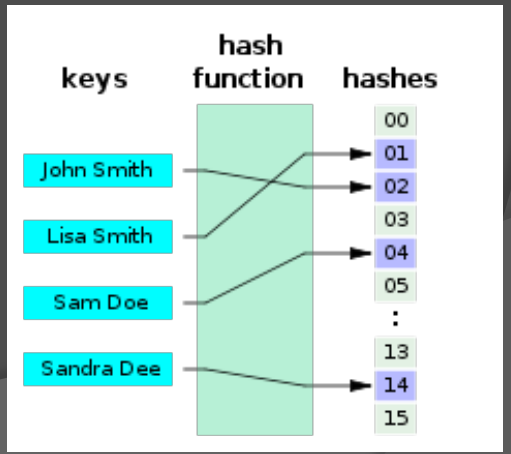
STRING ALGORITHMS (I)

Contents

- Hashing
- Knuth-Morris-Pratt (KMP) Algorithm

Hashing

- A technique to compress data into integers
- Applies mainly to strings and more complex data structures
- Similar functionality to state compression for DP



String Hashing

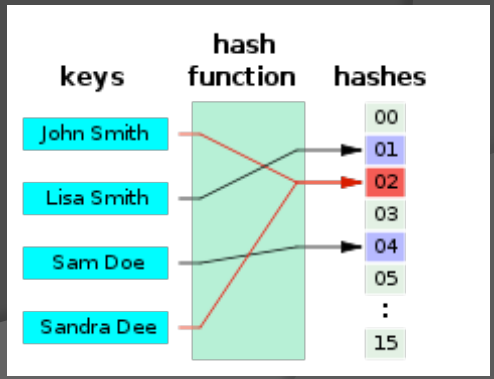
- Define S to be a string $S_0S_1...S_{N-1}$
- Define $hsh(i)$ to be a function that returns an integer given the string $S_0S_1...S_i$
- Let $hsh(0) = int(S_0)$
- For all $i > 0$, $hsh(i) = (hsh(i-1)*SEED + int(S_i)) \% MOD$
- This is the same as converting a string into a base 10 number from a base $SEED$ number, modulo MOD

Substrings

- Suppose we want the hash value of a substring $S_xS_{x+1}...S_{y-1}S_y$
- For convenience, all of the following derivations are under modulo MOD
- $hsh(x) = hsh(x-1)*SEED + int(S_x)$
- $hsh(x+1) = (hsh(x-1)*SEED + int(S_x))*SEED + int(S_{x+1})$
- $hsh(x+1) = hsh(x-1)*SEED^2 + int(S_x)*SEED + int(S_{x+1})$
- ...
- $hsh(y) = hsh(x-1)*SEED^{y-x} + k$
- k is the value of the hash of $S_xS_{x+1}...S_{y-1}S_y$

Collisions

- Occurs when two different strings have the same hash value
- By choosing good values for $SEED$ and MOD, we reduce the probability that collisions occur
  - Typically, choose a moderately large prime for $SEED$ (2-4 digits) and a very large prime for MOD (close to data type limit)
  - Choosing not to mod and allowing overflow may be effective



Collision Tips

- Handle collisions correctly
  - Will not provide details
- Hope that collisions will not occur, or will not affect your answer
  - On contests with hacking and submission penalty: avoid hashing
  - On contests with systesting: avoid hashing or submit many times with different *SEED* and *MOD* values
  - Otherwise: Submit as few times as possible, if you get **WA**, try changing *SEED* and *MOD*

Usage

- Integer comparison is very fast compared to string comparison
- Integers can easily be indexed
  - `unordered` data structures automatically hash, but need custom hash functions for complex data structures

Sample Problems

- Given a string S, what is the number of palindromic substrings of length M?
  - A palindrome is a string that is equal to its reverse (e.g. *"racecar"*)
- Given a string S, what is the longest substring that appears twice in the string?
  - Hint: use binary search and hashing

String Finding

- Given strings S and T, find the first occurrence of T in S
- Naive algorithm: For each character in S, see if the next $|T|$ characters are the same as T
  - Worst case runtime: $O(|S||T|)$
- Hashing: Hash T and all substrings of S of length $|T|$, find the first one that matches
  - Worst case runtime: $O(|S| + |T|)$, but risk of collisions

Knuth-Morris-Pratt Algorithm

- The KMP algorithm solves the string finding algorithm in $O(|S| + |T|)$ time
- Correctness is guaranteed, unlike hashing

KMP

- Idea is to use a next array/function based on T
- For all $i > 0$, $nxt(i)$ is the largest x such that $T_0T_1...T_{x-1} = T_{i-x}...T_{i-1}$
  - In the prefix of T of length i, P, x is the length of the longest prefix of P that is equal to the suffix of P of that length
- $nxt(0) = -1$

KMP Next Array Example

- Suppose $T =$ *"abcdabcab"*
- Next array is:

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
a	*b*	*c*	*d*	*a*	*b*	*c*	*a*	*b*
-1	*0*	*0*	*0*	*0*	*1*	*2*	*3*	*1*

Using the Next Array

- Issue with naive algorithm: when character matching fails, we go back in both S and T
- Using the next array, we only go back in T
- Suppose that we are at S_i and T_j, $j \geq 0$ and these characters do not match
  - We know that S_{i-1} and T_{j-1} match
    - Assume S_{-1} and T_{-1} are empty characters
  - Using the next array, we know that $T_0 T_1 \ldots T_{nxt(j)-1} = T_{j-nxt(j)} \ldots T_{j-1}$
  - Therefore, S_{i-1} and $T_{nxt(j)-1}$ match
  - Check S_i and $T_{nxt(j)}$
- If the characters match or $j = -1$, increment both i and j

KMP Pseudocode

```
S,T
nxt[]
//assume nxt is correct
i ← 0, j ← 0
while i < S.length() do
      while j != -1 and S[i] != T[j] do
            j ← nxt[j]
      i ← i+1, j ← j+1
      if j = T.length
            //found T in S
```

Generating the Next Array

- The next array can be generated in $O(|T|)$ time
- Use a similar idea for using the next array to generate the next array
- Let i be the right index of the suffix, j be the right index of the prefix
- If $T_i != T_j$, set $j = nxt(j)$
  - Recall $T_0T_1...T_{nxt(j)-1} = T_{j-nxt(j)}...T_{j-1}$
- Once $T_i = T_j$ or $j = -1$, set $nxt(i+1) = j+1$

Next Array Pseudocode

```
T
nxt[]
nxt[0] ← -1
i ← 0, j ← -1
while i < T.length() do
     while j != -1 and T[i] != T[j] do
          j ← nxt[j]
     i ← i+1, j ← j+1
     nxt[i] ← j
```

THANK YOU!