String Patterns: Rabin Karp

Rabin Karp

Like the Naive Algorithm, the Rabin-Karp algorithm also checks every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text, and if the hash values match then only it starts matching individual characters. So the Rabin Karp algorithm needs to calculate hash values for the following strings.

- Pattern itself
- All the substrings of the text of length m which is the size of the pattern.

How is Hash Value calculated in Rabin-Karp?

Hash value is used to efficiently check for potential matches between a pattern and substrings of a larger text. The hash value is calculated using a rolling hash function, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the text and calculate the hash value for each substring without recalculating the entire hash from scratch.

Here’s how the hash value is typically calculated in Rabin-Karp:

Step 1: Choose a suitable base and a modulus:

- Select a prime number ‘p‘ as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
- Choose a base ‘b‘ (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value:

- Set an initial hash value ‘hash‘ to 0.

Step 3: Calculate the initial hash value for the pattern:

- Iterate over each character in the pattern from left to right.
- For each character ‘c’ at position ‘i’, calculate its contribution to the hash value as ‘c * (bpattern_length – i – 1) % p’ and add it to ‘hash‘.
- This gives you the hash value for the entire pattern.

Step 4: Slide the pattern over the text:

- Start by calculating the hash value for the first substring of the text that is the same length as the pattern.

Step 5: Update the hash value for each subsequent substring:

- To slide the pattern one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position ‘i’ to ‘i+1’ is:

hash = (hash - (text[i - pattern_length] * (bpattern_length - 1)) % p) * b + text[i]

Step 6: Compare hash values:

- When the hash value of a substring in the text matches the hash value of the pattern, it’s a potential match.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

Find count of A inside B

Problem 1:

Q> Given a string A (of length N) & a string B (of length M) find the count of substrings of A which are equal to B

eg: A = "abcbdcacba"   ans ≡ 1
B = "cba"

A_2 = "ababa bac"   ans ≡ 2
B = "aba"

① Bruteforce:

∀ substring (N−M+1) of A of length m, check $\underset{O(m)}{\text{if}}$ it is B.
(sliding window)

$T.C \equiv O(N-M+1) * O(M)$

$\equiv O(N*M)$

② Using Hashing + s.w

a → 1
b → 2
c → 3
⋮
z → 26

① h('abc') ≡ sum of value of characters X won't work
two strings can have same val

② Convert to number with base next to 26 which is prime (P≡29)

eg: $h(aba) \equiv (1*p^2 + 2*p^1 + 1*p^0)$ $\boxed{P \equiv 29}$

eg using base ≡ 10

A = "abca babac"

B = "aba" → h(aba) → $1*10^2+2*10^1+1*10^0 = 121$

$h_1 \equiv 1*10^2+2*10^1+3*10^0 \equiv 123$ ✗

using sliding calculate h_2

$h_2 \equiv [(1*10^2+2*10^1+3*10^2) - 1*10^2] + 1*10^2$

⋮

$T.C \equiv O(\underbrace{M}_{\text{for } M} + \underbrace{N}_{\text{for } N \text{ elements using s.w}}) \equiv O(N)$

$S.C \equiv O(1)$

keep checking hash with key hash

edge case if length of string is large ⇒ h(s) ≡ overflow
≡ %M
(10^9+7)

Rolling Hash function

$$h(s) = \left(\sum_{i=0}^{N-1} (s[i] * p^{N-1-i})\right) \% M$$

But as we reduced the size by %M, there can be a chance of collision.

Probability of Collision

Strings	h(s)	probability of collision
S_1		0
S_2		$1/M$
S_3		$2/M$
|		
|		
S_N	$h(s_N)$	$N-1/M$
S_{N+1}	$h(s_{N+1})$	$N/M = 10^5/10^9 = 0.0001$ small probability

↳ (10^9+7)

③ Using Probabilistic Approach

A = "abcababac"   B = "aba"

$(1*p^2 + 2*p^1 + 3*p^0)\%M$ $(1*p^2 + 2*p + 1*p^0)\%M$

Case1 : $(h(s_1) != h(s_2)) \Rightarrow s_1 != s_2$

Case2 : $h(s_1) == h(s_2) \Rightarrow s_1 == s_2$

↳ $(s_1 == s_2)$ with high probability. but not 100%

↳ Do char by char comparision to verify if $[s_1 == s_2]$ Cost = $O(M)$

Total T.C = $O(N)$ best Case.

$O(N*M)$ worst Case.

Boring Substring

Problem

Given a string check whether it is possible to rearrange the characters s.t there is not boring substring in S.

Boring Substring → (len = 2) & consecutive chars.
eg: ab, bc, xy, yz, de, - - - -
[a, z] → 26 chars.

Eg:- "abc" false

"abcd" → cadb True
bdac

① **Bruteforce**

Check each permutations O(N!)

② "aabccdb" → "cc aadbb"

divide with logic split

aacc | bdb

Reverse and concatenate to get new string

ccaa bdb

" aabccb "

aacc | bb

false

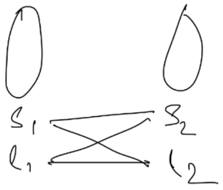
X will not work

Correct Approach

Step1 :- split the characters into two sets based on odd/even & keep track of smallest/largest.

Step2 :- Try each at (s_1, s_2), (s_1, l_2), (l_1, s_2)

if valid → True
else → false.



T.C : O(N)
S.C : O(1)