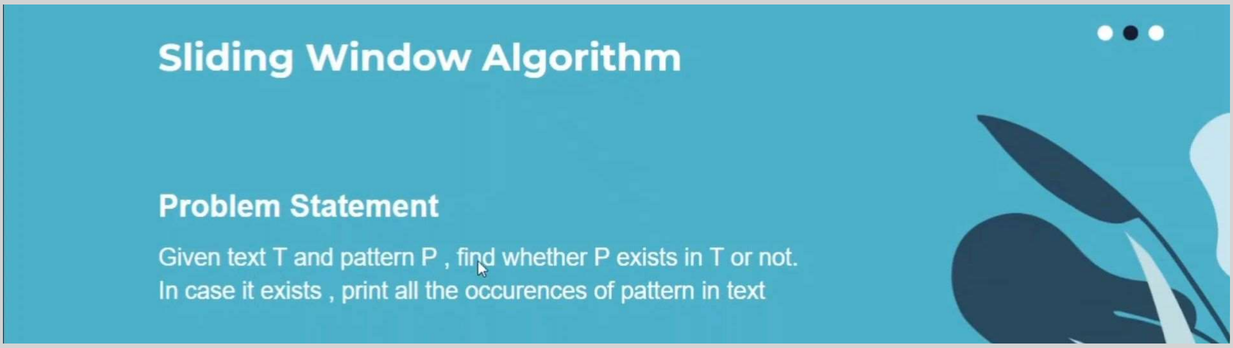
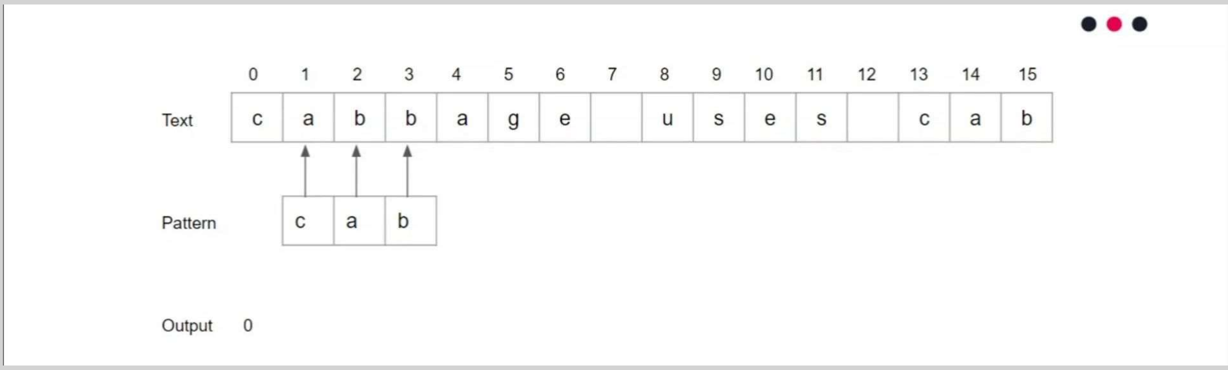
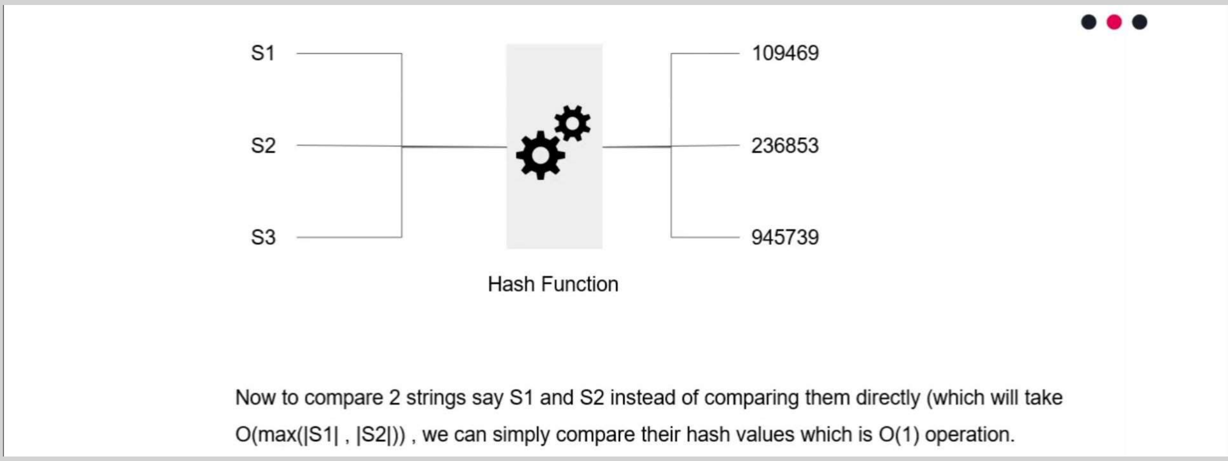
←

String Algorithms





1. String Hashing



Polynomial Rolling Hash

Polynomial Rolling Hash

$$\text{hash}(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + ... + s[n-1] \cdot p^{n-1} \mod m$$

$$\sum_{i=0}^{n-1} s[i] \cdot p^i \mod m$$

$$\text{hash}(\text{“}coding\text{”}) = c \cdot p^0 + o \cdot p^1 + d \cdot p^2 + i \cdot p^3 + n \cdot p^4 + g \cdot p^5$$

a = 1   b = 2   P >= size of character set
c = 3
d = 4   e = 6
f = 6
.   .
.
.   .
.
y = 25   z = 26

Why should we use modulo?

Answer : Integer Overflow.

Since hash function is polynomial , so hash values increase exponentially.

Inteter : 10 characters
Long Long int : 20 characters
Assuming p : 11

Why P >= | char set | ?

Answer : To reduce number of collisions

Let P = 11
hash(“L”) = 12 * 11^0 = 12
hash(“AA”) = 1 * 11^0 + 1 * 11^1 = 1 + 11 = 12

Code:

```cpp
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

snappify.com

Substring Hash in O(1):

Implementation Details

dp[i] = hash value of substring(0 , i)

for “coding” , dp[ ] array will look like this

$$dp[0] = c \cdot p^0$$

$$dp[1] = c \cdot p^0 + o \cdot p^1$$

$$dp[2] = c \cdot p^0 + o \cdot p^1 + d \cdot p^2$$

$$dp[3] = c \cdot p^0 + o \cdot p^1 + d \cdot p^2 + i \cdot p^3$$

$$dp[4] = c \cdot p^0 + o \cdot p^1 + d \cdot p^2 + i \cdot p^3 + n \cdot p^4$$

$$dp[5] = c \cdot p^0 + o \cdot p^1 + d \cdot p^2 + i \cdot p^3 + n \cdot p^4 + g \cdot p^5$$

** $H(str[\,l : r\,]) = (\,dp[r] - dp[l-1]\,) \,/\, p^l$

=> ((dp[r] - dp[l-1]) * modinv(p^l))%m

store mod_inverse previously for every p^l

mod_inv (a,m) => pow(a,m-2,m)

Code:

```cpp
/*
    const int p = 31, m = 1e9 + 9;
*/
int power(int x, int y, int p) {
    int res = 1;
    x = x % p;
    while (y > 0) {
        if (y & 1)
            res = (res * x) % p;
        y = y >> 1;
        x = (x * x) % p;
    }
    return res;
}
int modinv(int a, int m) {
    return power(a, m - 2, m);
}
struct Hash{
    vector<int> pref,powers,inv_powers;
    int p,m;
    Hash(string &s,int _p,int _m): p(_p) , m(_m)
    {
        int n=s.size();
        powers.resize(n+1,0);
        pref.resize(n+1,0);
        inv_powers.resize(n+1,0);
        powers[0]=1;
        int p_inv=modinv(p,m);
        inv_powers[0]=1;
        for(int i=1;i<=n;i++){
            pref[i]=(pref[i-1]+(s[i-1]-'a'+1)*powers[i-1])%m;
            powers[i]=(powers[i-1]*p)%m;
            inv_powers[i]=(inv_powers[i-1]*p_inv)%m;
        }
    }
    int get(int l,int r){
        return ((pref[r]-pref[l-1]+m)*inv_powers[l-1])%m;
    }
};
```

snappify.com

<u>Rabin Karp:</u>

Store p^l and then Just check

H(str[l : r]) *p^l == (dp[r] - dp[l-1])

KMP Algorithm

Knuth-Morris-Pratt Algorithm

KMP algorithm depends upon prefix function for its implementation.

Prefix function or pi function has below definition.

Pi(i) = length of longest proper prefix of substring(0 , i) which is also a suffix

Prefix Function

String	: a b c a b c d
Prefix function	: 0 0 0 1 2 3 0

String	: a a b a a a b
Prefix function	: 0 1 0 1 2 2 3

Imp. observation

The first important observation is, that the values of the prefix function can only increase by at most one.

$\pi(i+1) \leq \pi(i)+1$

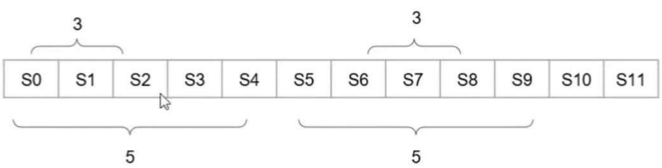
String	: a a b a a a b
Prefix function	: 0 1 0 1 2 2 3

Proof

Let

$\pi(i) = 3$

$\pi(i+1) = 3 + 2 = 5$



Improvement

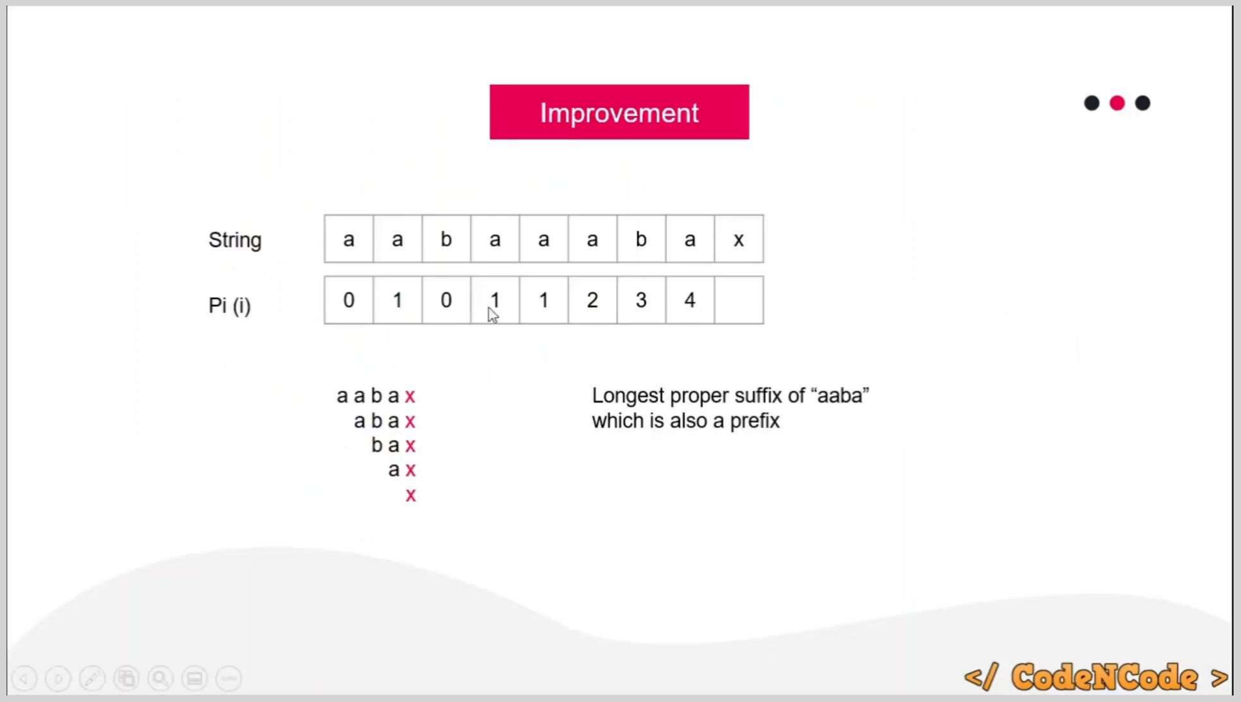
pie(i-1)

String	a	a	b	$\boxed{c}$	a	a	b	$\boxed{c}$	_
Pi (i)	0	1	0	0	1	2	3		

$if(S[\pi(i-1)] == S[i])$

$\pi(i) = \pi(i-1)+1$

just have to check the ith char because prev chars are matching.
if(s[i] != s[j])



Code:

```cpp
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
// KMP
string t = pat + '$' + s;
for (int i = sz(pat) + 1; i < n; i++) {
        if (pi[i] == sz(pat))
            occ.pb(i - 2 * sz(pat));
}
```