Haszowanie Napisów:

Stworzymy taką funkcję $h: \mathcal{U} \to \mathbb{Z}_m$, że jeśli dwa słowa A i B są równe, to $h(A) = h(B)$.

Tzn:

$$A = B \Rightarrow h(A) = h(B)$$

Uwaga:

Nie musi zachodzić implikacja w drugą stronę:

$$h(A) = h(B) \not\Rightarrow A = B$$ ← kolizje hashy

Wynika to z tego, że wszystkich możliwych słów jest więcej niż jesteśmy w stanie pamieścić w komputerze. Nasza funkcja h każdemu słowu przypisuje liczbę z ograniczonego zbioru. Chcemy wybrać takie ograniczenie zbioru i sposób przypisywanie liczb, tak aby zminimalizować ryzyko kolizji.

Zatem należy pamiętać, że używanie hashy nie jest rozwiązaniem które w 100% daje poprawną odpowiedź. Zadnek ich prostote i bardzo szerokie zastosowanie sprawiają, że są popularną techniką.

Hash Wielomianowy (Rolling Hash)

Niech s – napis, $|s| = n$

Wtedy:

$$h(s) = s[0] + s[1]p + s[2]p^2 + \ldots + s[n-1]p^{n-1} \pmod m$$

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

, gdzie $p, m > 0$

Zmienne p i m dobieramy tak, aby zminimalizować ryzyko kolizji.

Jeśli s składa się z liter 'a', 'b', ..., 'z' to dobrym wyborem jest $p = 31$.

Chcemy aby m było jak największe, bo prawdopodobieństwo kolizji to $\approx \frac{1}{m}$. Dobrym pomysłem na m jest duża liczba pierwsza, którą będziemy mogli łatwo mnożyć (bez używania typów większych od long long'a).

Przykładowo $m = 10^9 + 7$ jest OK.

Uwaga:

Na konkursach takich jak OI układane są testy mające wykryć hashe z popularnymi (p, m). Dlatego polecam poszukać swoich ulubionych liczb na (p, m), które nie będą "oklepane".

Kod:

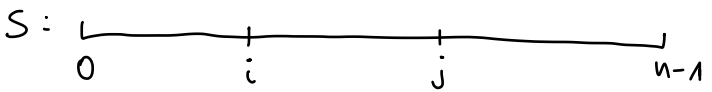
```cpp
int ppow[LIMIT];

void precompute_powers(int n) {
    ppow[0] = 1;
    for (int i=1; i<=n; i++)
        ppow[i] = (1LL * ppow[i-1] * p) % m;
}

int h(const string &s) {
    int res = 0;
    for (int i=0; i<s.size(); i++)
        res = (res + 1LL * (s[i]-'a'+1) * ppow[i]) % m;
    return res;
}
```

Szybkie wyznaczenie hashy dla spójnego podstowa w tekście.

Niech s – tekst, a $s[i \dots j]$ – podstowo w tekście od i do j włącznie.



Podobnie jak w sumach prefiksowych:

$$h(s[0 \dots j]) - h(s[0 \dots i-1]) =$$

$$= s[0]p^0 + s[1]p^1 + \dots + s[i-1]p^{i-1} + s[i]p^i + \dots + s[j]p^j$$

$$- (s[0]p^0 + s[1]p^1 + \dots + s[i-1]p^{i-1}) =$$

$$= s[i]p^i + s[i+1]p^{i+1} + \dots + s[j]p^j \pmod m$$

Wtedy po podzieleniu przez p^i:

$$s[i]p^0 + s[i+1]p^1 + \dots + s[j]p^{j-i} \pmod m$$

$$= \sum_{k=i}^{j} s[k]p^{k-i} = h(s[i \dots j])$$

Problem z dzieleniem modulo m:

Chcemy w szybki sposób znaleźć dla dowolnej liczby g jej odwrotność modulo m:

Odwrotnością liczby g to taka liczba x, że zachodzi: $gx \equiv xg \equiv 1 \pmod m$

Wtedy x oznaczamy jako g^{-1}.

Przykład:

Niech $\pmod 7$:

- $5^{-1} \equiv 3$, ponieważ $5 \cdot 3 = 15 \equiv 1 \pmod 7$
  - ponieważ: $15 = 2 \cdot 7 + 1$
- $2^{-1} \equiv 4$, ponieważ $2 \cdot 4 = 8 \equiv 1 \pmod 7$

Twierdzenie:

- $NWD(g, m) = 1 \Rightarrow \forall_{g \in [0,m)} \exists! \, g^{-1}$

  czyli g i m są względnie pierwsze.

Dowód twierdzenie:

lemat Bezout

$NWD(g, m) = 1 \Rightarrow \exists_{s,t} : sg + tm = 1$

$\Downarrow$

$sg + tm \equiv 1 \pmod m$

$\Downarrow$ $tm \equiv 0 \pmod m$

$sg \equiv 1 \pmod m$

Zatem $s = g^{-1}$. □

To dlatego m powinne być liczbą pierwszą.

Jak zależć odwrotność ?

Skorzystamy z Lematu Fermata:

↳ $NWD(g, m) = 1 \Rightarrow g^{\phi(m)} \equiv 1 \pmod m$

, gdzie $\phi(m)$ to tzw. tocjent czyli liczba liczb względnie pierwszych z m i nie większych od m.

Zatem jeśli m jest l. pierwszą $\Rightarrow \phi(m) = m-1$

Więc: $g^{m-1} \equiv 1 \pmod m$

Teraz pomnóżmy obustronnie przez g^{-1}:

$$g^{m-2} \equiv g^{-1} \pmod{m}$$

A zatem w kodzie wystarczy podnieść g do potęgi $m-2$ aby otrzymać $g^{-1} \pmod{m}$.

Zrobimy to za pomocą szybkiego potęgowania w $O(\log m)$:

```
int fast_pow(int a, int b) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = (1LL * res * a) % m;
        a = (1LL * a * a) % m;
        b /= 2;
    }
    return res;
}
```

Uwaga:
Oczywiście musimy pamiętać aby wykonywać mnożenie mod m.

Mając to narzędzie w ręku możemy obliczyć odwrotności dla każdej potęgi p^i, $i \in \{0, 1, \dots, n\}$

```
void precompute_inverses(int n) {
    inv[0] = 1;
    int inverse = fast_pow(p, m-2); // p^-1
    inv[1] = inverse;
    for (int i=2; i<=n; i++)
        inv[i] = (1LL * inv[i-1] * inverse) % m;
}
```

ponieważ:

$$(p^{i-1})^{-1} \cdot p^{-1} = (p^{i-1} \cdot p^{1})^{-1} = (p^{i})^{-1}$$

Minimalizowanie Kolizji:

Jeśli wykonujemy dużo porównań (np. porównując wszystkie podstawy w tekście) to prawdopodobieństwo kolizji może być zbyt wysokie aby program przeszedł wszystkie testy.

Aby wzmocnić hashowanie możemy stworzyć więcej niż jedną funkcję h, każdą z innymi (p, m).

Zastosowanie Hashowanie:

- Algorytm Rabina-Karpa
- Znajdowanie różnych podstaw w tekście
- Znajdowanie najdłuższego palindromu w tekście w $O(n \log n)$
- wiele więcej...

Źródło: cp-algorithms.com