

Strings

Strings e *Hashes*

Prof. Edson Alves - UnB/FGA

2019

1. Strings e *Hashes*
2. *Polynomial Rolling Hash*

Strings e *Hashes*

Comparação de strings

- Duas strings S e T são iguais se $S[i] = T[i]$, para $i \in [1, n]$, com $n = |S| = |T|$
- A comparação entre os caracteres de posições correspondentes faz com que esta verificação tem complexidade $O(n)$
- Uma maneira de realizar esta comparação de forma mais eficiente é utilizar uma função de *hash* h , que transforma uma string S em um inteiro $h(S)$, e comparar $h(S)$ com $h(T)$
- Como a comparação de inteiros, em geral, é feita em $O(1)$, a complexidade da comparação dependerá apenas do custo de se computar $h(S)$

- Seja S o conjunto de todas as strings possíveis e q um número natural
- Denominamos

$$h : S \rightarrow [0, q]$$

uma função de *hash* em S

- Observe que, como h é função, se $S = T$ então $h(S) = h(T)$
- A recíproca não é necessariamente verdadeira: pode acontecer $h(S) = h(T)$ com $S \neq T$
- Isto ocorre porque o número de strings possíveis é, em geral, muito maior do que o intervalo $[0, q]$, de modo que h não é injetiva
- Esta situação é denominada colisão
- O desafio é definir h de modo a minimizar o número de colisões

Polynomial Rolling Hash

Polynomial Rolling Hash

Seja S uma string de tamanho n , cujos elementos são indexados de 0 a $n - 1$. A função

$$\begin{aligned} h(S) &= \left(\sum_{i=0}^{n-1} S_i p^i \right) \bmod q \\ &= (S_0 + S_1 p + S_2 p^2 + \dots + S_{n-1} p^{n-1}) \bmod q, \end{aligned}$$

onde p e q são dois inteiros positivos, é denominada *polynomial rolling hash*.

Escolha dos parâmetros

- Em geral, p é um número primo aproximadamente igual ao tamanho do alfabeto
- Para um alfabeto de 26 letras, uma escolha razoável seria $p = 31$
- Para maiúsculas e minúsculas pode-se adotar $p = 53$
- O valor de q deve ser grande, pois a chance de colisão entre duas strings sorteadas aleatoriamente é de $1/q$
- Usar um número primo para q também é uma boa escolha, no sentido de evitar colisões
- O valor $q = 10^9 + 7$ tem a vantagem de ser fácil de lembrar e digitar, e também de permitir a multiplicação sem *overflow* usando variáveis do tipo **long long**

Mapeamento de caracteres

- Na definição da função h o valor s_i corresponde ao mapeamento do caractere $S[i]$ da string para um inteiro
- Em termos formais, dado um alfabeto \mathcal{A} e uma função

$$f : \mathcal{A} \rightarrow \mathbb{N},$$

então $s_i = f(S[i])$, onde $S[i] \in \mathcal{A}$ para todo $i = 0, 1, 2, \dots, N - 1$

- Um mapeamento possível seria $f(a) = 1, f(b) = 2, \dots, f(z) = 25$
- Veja que o caractere ‘a’ não é mapeado para zero, e sim para um, para evitar que todas as strings compostas por repetições deste caractere tenham o mesmo *hash* h

Implementação do *rolling hash* em Haskell

```
1 import Data.Char
2
3 f :: Char -> Int
4 f c = (ord c) - (ord 'a') + 1
5
6 h :: String -> Int
7 h s = sum (zipWith (*) fs ps) `mod` q where
8     p = 31
9     q = 10^9 + 7
10    fs = map f s
11    ps = map (\x -> p ^ x) $ take (length s) [0..]
```

Implementação do *rolling hash* em C++

```
1 int f(char c)
2 {
3     return c - 'a' + 1;
4 }
5
6 int h(const string& s)
7 {
8     long long ans = 0, p = 31, q = 1000000007;
9
10    for (auto it = s.rbegin(); it != s.rend(); ++it)
11    {
12        ans = (ans * p) % q;
13        ans = (ans + f(*it)) % q;
14    }
15
16    return ans;
17 }
```

Calculo do *hash* das substrings de S

- Dada uma string S , a definição de h permite computar o valor de $h(S[i..j])$, para qualquer par $i \leq j$ de índices válidos, em $O(1)$, se conhecidos os valores de h para todos os prefixos $S[0..i]$ de S
- A função h é definida por

$$h(S) = \left(\sum_{i=0}^{n-1} S_i p^i \right) \bmod q$$

- Deste modo,

$$\begin{aligned} h(S[i..j]) p^i &= \left(\sum_{k=i}^j S_k p^i \right) \bmod q \\ &= (h(S[0..j]) - h(S[0..(i-1)])) \bmod q \end{aligned}$$

Calculo do *hash* das substrings de S

- Para obter o valor de $S[i..j]$, é necessário multiplicar a expressão acima pelo inverso multiplicativo $(p^i)^{-1}$ de p^i módulo q
- Este pode ser obtido pelo Pequeno Teorema de Fermat: se q é primo e $(a, q) = 1$, então

$$a^{q-1} \equiv 1 \pmod{q}$$

- Assim, como $q \geq 2$,

$$a \cdot a^{q-2} \equiv 1 \pmod{q},$$

de modo que

$$a^{-1} \equiv a^{q-2} \pmod{q}$$

- Se os inversos de p^i também forem precomputados, juntamente com os *hashes* dos prefixos $S[0..i]$, os valores $h(S[i..j])$ podem ser calculados em $O(1)$

Contagem das substrings distintas em Haskell

```
1 import Data.Char
2 import Data.Set
3
4 p = 31
5 q = 10^9 + 7
6
7 f :: Char -> Int
8 f c = (ord c) - (ord 'a') + 1
9
10 h :: String -> Int
11 h s = sum (zipWith (*) fs ps) `mod` q where
12     fs = Prelude.map f s
13     ps = Prelude.map (\x -> p ^ x) $ take (length s) [0..]
14
15 prefixes :: String -> [Int]
16 prefixes s = [h $ take k s | k <- [1..n]] where n = length s
17
18 fastExpMod :: Int -> Int -> Int
19 fastExpMod _ 0 = 1
20 fastExpMod a n = (b * fastExpMod (a^2 `mod` q) (n `div` 2)) `mod` q where
21     b = if n `mod` 2 == 1 then a else 1
```

Contagem das substrings distintas em Haskell

```
22
23 inverses :: Int -> [Int]
24 inverses n = [fastExpMod (fastExpMod p i) (q - 2) | i <- [0..(n-1)]]
25
26 hsubs i j ps is
27     | i == 0 = ps !! j
28     | otherwise = (ps !! j - ps !! (i - 1)) * is !! i `mod` q
29
30 unique_substrings s = length us where
31     n = length s
32     xs = [(i, j) | i <- [0..(n-1)], j <- [i..(n-1)]]
33     ps = prefixes s
34     is = inverses n
35     hs = [hsubs i j ps is | (i, j) <- xs]
36     us = fromList hs           -- us :: Data.Set
37
38 main = do
39     print $ unique_substrings "tep"
40     print $ unique_substrings "banana"
41     print $ unique_substrings "aaaaa"
```

Contagem das substrings distintas em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using ll = long long;
5
6 const ll p = 31, q = 1000000007;
7
8 int f(char c)
9 {
10     return c - 'a' + 1;
11 }
12
13 int h(const string& s)
14 {
15     ll ans = 0;
16
17     for (auto it = s.rbegin(); it != s.rend(); ++it)
18     {
19         ans = (ans * p) % q;
20         ans = (ans + f(*it)) % q;
21     }
```


Contagem das substrings distintas em C++

```
22
23     return ans;
24 }
25
26 vector<ll> prefixes(const string& s)
27 {
28     int N = s.size();
29     vector<ll> ps(N, 0);
30
31     for (int i = 0; i < N; ++i)
32         ps[i] = h(s.substr(0, i + 1));
33
34     return ps;
35 }
36
37 ll fast_exp_mod(ll a, ll n)
38 {
39     ll res = 1, base = a;
40
```

Contagem das substrings distintas em C++

```
41     while (n) {
42         if (n & 1)
43             res = (res * base) % q;
44
45         base = (base * base) % q;
46         n >>= 1;
47     }
48
49     return res;
50 }
51
52 vector<ll> inverses(ll N)
53 {
54     vector<ll> is(N);
55     ll base = 1;
56
57     for (int i = 0; i < N; ++i)
58     {
59         is[i] = fast_exp_mod(base, q - 2);
60         base = (base * p) % q;
61     }
```

Contagem das substrings distintas em C++

```
62
63     return is;
64 }
65
66 int h(int i, int j, const vector<ll>& ps, const vector<ll>& is)
67 {
68     auto diff = i ? ps[j] - ps[i - 1] : ps[j];
69     diff = (diff * is[i]) % q;
70
71     return (diff + q) % q;
72 }
73
74 int unique_substrings(const string& s)
75 {
76     set<ll> hs;
77     int N = s.size();
78
79     auto ps = prefixes(s);
80     auto is = inverses(s.size());
81
```

Contagem das substrings distintas em C++

```
82     for (int i = 0; i < N; ++i)
83     {
84         for (int j = i; j < N; ++j)
85         {
86             auto hij = h(i, j, ps, is);
87             hs.insert(hij);
88         }
89     }
90
91     return hs.size();
92 }
93
94 int main()
95 {
96     cout << unique_substrings("tep") << '\n';
97     cout << unique_substrings("banana") << '\n';
98     cout << unique_substrings("aaaaa") << '\n';
99
100    return 0;
101 }
```

Redução da probabilidade de colisão

- Dadas duas strings S e T escolhidas aleatoriamente, a probabilidade de colisão entre ambas é de $1/q$
- Assim, com $q = 10^9 + 7$, se S for comparado com $n = 10^6$ strings distintas, a probabilidade de acontecer uma colisão é igual a $n/q = 10^3$
- Um modo de diminuir esta probabilidade é utilizar o *hash* duas vezes
- Em termos mais preciso, seja h_i a função de *rolling hash* que utiliza os parâmetros p_i e q_i
- O *hash* duplo h_{ij} associa uma string S a um par de inteiros da seguinte maneira:

$$h_{ij}(S) = (h_i(S), h_j(S))$$

- Se $q_i, q_j > 10^9$, a função h_{ij} produz mais de 10^{18} pares distintos, de forma que a comparação de S com $n = 10^6$ strings distintas passa a ter probabilidade de colisão igual a $n/(q_i q_j) = 1/10^{12}$

Implementação do *hash* duplo em Haskell

```
1 import Data.Char
2
3 f :: Char -> Int
4 f c = (ord c) - (ord 'a') + 1
5
6 hi :: String -> Int -> Int -> Int
7 hi s p q = sum (zipWith (*) fs ps) `mod` q where
8     fs = map f s
9     ps = map (\x -> p ^ x) $ take (length s) [0..]
10
11 h :: String -> (Int, Int)
12 h s = (hi s p1 q1, hi s p2 q2) where
13     p1 = 31
14     q1 = 109 + 7
15     p2 = 29
16     q2 = 109 + 9
17
18 main :: IO ()
19 main = do
20     s <- getLine
21     print $ h s
```

Implementação do *hash* duplo em C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 int f(char c)
6 {
7     return c - 'a' + 1;
8 }
9
10 int hi(long long pi, long long qi, const string& s)
11 {
12     long long ans = 0;
13
14     for (auto it = s.rbegin(); it != s.rend(); ++it)
15     {
16         ans = (ans * pi) % qi;
17         ans = (ans + f(*it)) % qi;
18     }
19
20     return ans;
21 }
```

Implementação do *hash* duplo em C++

```
22
23 pair<int, int> h(const string& s)
24 {
25     const long long p1 = 31, q1 = 1000000007;
26     const long long p2 = 29, q2 = 1000000009;
27
28     return make_pair(hi(p1, q1, s), hi(p2, q2, s));
29 }
30
31 int main()
32 {
33     string s;
34     cin >> s;
35
36     auto hs = h(s);
37
38     cout << "(" << hs.first << ", " << hs.second << ")\n";
39
40     return 0;
41 }
```


1. CP-Algorithms. [String Hashing](#), acesso em 06/08/2019.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.