

Strings

Strings e *Hashes*

Prof. Edson Alves - UnB/FGA

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 $n = |S| = |T|$

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 $O(n)$

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 $h(T)$

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 \mathcal{S} o conjunto de todas as strings possíveis e q um número natural

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{S}

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{S}

- Observe que, como h é função, se $S = T$ então $h(S) = h(T)$

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{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 $S \neq T$

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{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

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{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

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

$$h : \mathcal{S} \rightarrow [0, q]$$

uma função de *hash* em \mathcal{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

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 $p = 31$

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 $p = 53$

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 $1/q$

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

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

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 Σ e uma função

$$f : \Sigma \rightarrow \mathbb{N},$$

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

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 Σ e uma função

$$f : \Sigma \rightarrow \mathbb{N},$$

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

- Um mapeamento possível seria $f(\text{'a'}) = 1, f(\text{'b'}) = 2, \dots, f(\text{'z'}) = 26$

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 Σ e uma função

$$f : \Sigma \rightarrow \mathbb{N},$$

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

- Um mapeamento possível seria $f(\text{'a'}) = 1, f(\text{'b'}) = 2, \dots, f(\text{'z'}) = 26$
- 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 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 = 1'000'000'007;
9
10    for (auto c : s)
11    {
12        ans = (ans * p) % q;
13        ans = (ans + f(c)) % 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

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

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

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,

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

Calculo do *hash* das substrings de S

- Para obter o valor do *hash* para $S[i..j]$ é necessário multiplicar a expressão acima pelo inverso multiplicativo $(p^i)^{-1}$ de p^i módulo q

Calculo do *hash* das substrings de S

- Para obter o valor do *hash* para $S[i..j]$ 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

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

Calculo do *hash* das substrings de S

- Para obter o valor do *hash* para $S[i..j]$ 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

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

Calculo do *hash* das substrings de S

- Para obter o valor do *hash* para $S[i..j]$ 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 C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using ll = long long;
5 constexpr ll p = 31, q = 1'000'000'007;
6
7 int f(char c)
8 {
9     return c - 'a' + 1;
10 }
11
12 int h(const string& s)
13 {
14     ll ans = 0;
15
16     for (auto c : s)
17     {
18         ans = (ans * p) % q;
19         ans = (ans + f(c)) % q;
20     }
```

Contagem das substrings distintas em C++

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

Contagem das substrings distintas em C++

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

Contagem das substrings distintas em C++

```
62     return is;
63 }
64
65 int h(int i, int j, const vector<ll>& ps, const vector<ll>& is)
66 {
67     auto diff = i ? ps[j] - ps[i - 1] : ps[j];
68     diff = (diff * is[i]) % q;
69
70     return (diff + q) % q;
71 }
72
73 int unique_substrings(const string& s)
74 {
75     set<ll> hs;
76     int N = s.size();
77
78     auto ps = prefixes(s);
79     auto is = inverses(s.size());
80
81     for (int i = 0; i < N; ++i)
```

Contagem das substrings distintas em C++

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

Redução da probabilidade de colisão

- Dadas duas strings S e T escolhidas aleatoriamente, a probabilidade de colisão entre ambas é de $1/q$

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 $n/q = 10^3$

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

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

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:

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

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 C++

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

Implementação do *hash* duplo em C++

```
20 pair<int, int> h(const string& s)
21 {
22     constexpr long long p1 = 31, q1 = 1'000'000'007;
23     constexpr long long p2 = 29, q2 = 1'000'000'009;
24
25     return { hi(p1, q1, s), hi(p2, q2, s) };
26 }
27
28 int main()
29 {
30     string s;
31     cin >> s;
32
33     auto [h1, h2] = h(s);
34
35     cout << "(" << h1 << ", " << h2 << ")\n";
36
37     return 0;
38 }
```

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.