# Strings

## Strings e *Hashes*

Prof. Edson Alves - UnB/FGA

# Sumário

# 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

# *Hash*

- Seja $\mathcal{S}$ o conjunto de todas as strings possíveis e $q$ um número natural
- Denominamos
$$h : \mathcal{S} \to [0, q]$$
uma função de *hash* em $\mathcal{S}$

## *Hash*

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

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

  uma função de *hash* em $\mathcal{S}$
- Observe que, como $h$ é função, se $S = T$ então $h(S) = h(T)$

## *Hash*

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

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

  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 $S \neq T$

# *Hash*

- Seja $\mathcal{S}$ o conjunto de todas as strings possíveis e $q$ um número natural
- Denominamos
$$h : \mathcal{S} \to [0, q]$$
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

# *Hash*

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

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

  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

# *Hash*

- Seja $\mathcal{S}$ o conjunto de todas as strings possíveis e $q$ um número natural
- Denominamos
$$h : \mathcal{S} \to [0, q]$$
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*

# Definição

**Polynomial Rolling Hash**

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

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

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 $\Sigma$ e uma função

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

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

# 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 $\Sigma$ e uma função

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

  então $s_i = f(S[i])$, onde $S[i] \in \Sigma$ para todo $i = 0, 1, 2, \ldots, N-1$
- Um mapeamento possível seria $f(\texttt{'a'}) = 1, f(\texttt{'b'}) = 2, \ldots, f(\texttt{'z'}) = 26$

# 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 $\Sigma$ e uma função

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

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

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

```
int f(char c)
{
    return c - 'a' + 1;
}

int h(const string& s)
{
    int N = s.size();
    long long ans = 0, p = 31, q = 1'000'000'007;

    for (int i = N - 1; i >= 0; --i)
    {
        ans = (ans * p) % q;
        ans = (ans + f(s[i])) % q;
    }

    return ans;
}
```

## 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

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

- Deste modo,

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

# 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 $q \geq 2$,

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

vale 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 $q \geq 2$,

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

  vale 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++

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;
constexpr ll p = 31, q = 1'000'000'007;

int f(char c) { return c - 'a' + 1; }

int h(const string& s)
{
    int N = s.size();
    ll ans = 0;

    for (int i = N - 1; i >= 0; --i) {
        ans = (ans * p) % q;
        ans = (ans + f(s[i])) % q;
    }

    return ans;
}
```

# Contagem das substrings distintas em C++

```
vector<ll> prefixes(const string& s)
{
    int N = s.size();
    vector<ll> ps(N, 0);

    for (int i = 0; i < N; ++i)
        ps[i] = h(s.substr(0, i + 1));

    return ps;
}

ll fast_exp_mod(ll a, ll n)
{
    ll res = 1, base = a;

    while (n)
    {
        if (n & 1)
            res = (res * base) % q;
```

# Contagem das substrings distintas em C++

```
        base = (base * base) % q;
        n >>= 1;
    }

    return res;
}

vector<ll> inverses(ll N)
{
    vector<ll> is(N);
    ll base = 1;

    for (int i = 0; i < N; ++i) {
        is[i] = fast_exp_mod(base, q - 2);
        base = (base * p) % q;
    }

    return is;
}
```

# Contagem das substrings distintas em C++

```cpp
int h(int i, int j, const vector<ll>& ps, const vector<ll>& is)
{
    auto diff = i ? ps[j] - ps[i - 1] : ps[j];
    diff = (diff * is[i]) % q;
    return (diff + q) % q;
}

int unique_substrings(const string& s)
{
    int N = s.size();
    set<ll> hs;
    auto ps = prefixes(s);
    auto is = inverses(s.size());

    for (int i = 0; i < N; ++i) {
        for (int j = i; j < N; ++j) {
            auto hij = h(i, j, ps, is);
            hs.insert(hij);
        }
    }
```

# Contagem das substrings distintas em C++

```cpp

    return hs.size();
}

int main()
{
    cout << unique_substrings("tep") << '\n';
    cout << unique_substrings("banana") << '\n';
    cout << unique_substrings("aaaaa") << '\n';

    return 0;
}
```

# 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++

```cpp
#include <iostream>

using namespace std;

int f(char c) { return c - 'a' + 1; }

int hi(long long pi, long long qi, const string& s)
{
    int N = s.size();
    long long ans = 0;

    for (int i = N - 1; i >= 0; --i)
    {
        ans = (ans * pi) % qi;
        ans = (ans + f(s[i])) % qi;
    }

    return ans;
}
```

# Implementação do *hash* duplo em C++

```cpp
pair<int, int> h(const string& s)
{
    constexpr long long p1 = 31, q1 = 1'000'000'007;
    constexpr long long p2 = 29, q2 = 1'000'000'009;

    return { hi(p1, q1, s), hi(p2, q2, s) };
}

int main()
{
    string s;
    cin >> s;

    auto [h1, h2] = h(s);

    cout << "(" << h1 << ", " << h2 << ")\n";

    return 0;
}
```

# Referências


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.