Strings

Algoritmo de Rabin-Karp

Prof. Edson Alves - UnB/FGA

Sumário

Algoritmo de Rabin-Karp

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987
- A ideia principal do algoritmo é computar o *hash* $h_P = h(P)$ e compará-lo com todas as substrings $h_{ij} = S[i..j]$ de S de tamanho m

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987
- A ideia principal do algoritmo é computar o *hash* $h_P = h(P)$ e compará-lo com todas as substrings $h_{ij} = S[i..j]$ de S de tamanho m
- Caso $h_P \neq h_{ij}$, segue que $P \neq S[i..j]$ e o algoritmo pode prosseguir

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987
- A ideia principal do algoritmo é computar o *hash* $h_P = h(P)$ e compará-lo com todas as substrings $h_{ij} = S[i..j]$ de S de tamanho m
- Caso $h_P \neq h_{ij}$, segue que $P \neq S[i..j]$ e o algoritmo pode prosseguir
- Se $h_P = h_{ij}$, as strings ou são iguais ou houve uma colisão

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987
- A ideia principal do algoritmo é computar o *hash* $h_P = h(P)$ e compará-lo com todas as substrings $h_{ij} = S[i..j]$ de S de tamanho m
- Caso $h_P \neq h_{ij}$, segue que $P \neq S[i..j]$ e o algoritmo pode prosseguir
- Se $h_P = h_{ij}$, as strings ou são iguais ou houve uma colisão
- Esta dúvida pode ser sanada através da comparação direta entre $S[i..j]$ e P

Definição

- O algoritmo de Rabin-Karp é um algoritmo que contabiliza o número de ocorrências da string P, de tamanho m, na string S, de tamanho n
- Ele foi proposto por Michael O. Rabin e Richard M. Karp em 1987
- A ideia principal do algoritmo é computar o *hash* $h_P = h(P)$ e compará-lo com todas as substrings $h_{ij} = S[i..j]$ de S de tamanho m
- Caso $h_P \neq h_{ij}$, segue que $P \neq S[i..j]$ e o algoritmo pode prosseguir
- Se $h_P = h_{ij}$, as strings ou são iguais ou houve uma colisão
- Esta dúvida pode ser sanada através da comparação direta entre $S[i..j]$ e P
- O algoritmo tem complexidade $O(mn)$ no pior caso, por conta do custo do cálculo dos *hashes* e das possíveis comparações diretas entre as strings

Pseudocódigo do algoritmo de Rabin-Karp

Algoritmo 1 Algoritmo de Rabin-Karp – Naive

Input: Duas strings P e S e uma função de *hash* h

Output: O número de ocorrências occ de P em S

```
function RabinKarp(P,S)
    m ← |P|
    n ← |S|
    occ ← 0
    h_P ← h(P)
    for i ← 1 to n − m + 1 do
        h_S ← h(S[i..(i + m − 1)])
        if h_S = h_P then
            if S[i..(i + m − 1)] = P then
                occ ← occ + 1
    return occ
```

Implementação do algoritmo de Rabin-Karp em C++

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

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

int h(const std::string& s)
{
    long long ans = 0, p = 31, q = 1000000007;

    for (auto it = s.rbegin(); it != s.rend(); ++it)
    {
        ans = (ans * p) % q;
        ans = (ans + f(*it)) % q;
    }

    return ans;
}
```

Implementação do algoritmo de Rabin-Karp em C++

```cpp
int rabin_karp(const std::string& s, const std::string& p)
{
    int n = s.size(), m = p.size(), occ = 0, hp = h(p);

    for (int i = 0; i <= n - m; i++)
    {
        auto b = s.substr(i, m);
        occ += (h(b) == hp && b == p) ? 1 : 0;
    }

    return occ;
}
```

Variantes do algoritmo de Rabin-Karp

Diminuição da complexidade para o cálculo dos *hashes*

- Da maneira como foi apresentada, o algoritmo de Rabin-Karp tem complexidade $O(mn)$ no pior caso, o mesmo da busca completa, e com *runtime* maior, por conta do cálculo dos *hashes*

Diminuição da complexidade para o cálculo dos *hashes*

- Da maneira como foi apresentada, o algoritmo de Rabin-Karp tem complexidade $O(mn)$ no pior caso, o mesmo da busca completa, e com *runtime* maior, por conta do cálculo dos *hashes*
- Uma primeira melhoria que pode ser feita é usar o *rolling hash*, e computar $h(S[(i+1)..(i+m)])$ a partir de $h(S[i..(i+m-1)])$ com custo $O(1)$

Diminuição da complexidade para o cálculo dos *hashes*

- Da maneira como foi apresentada, o algoritmo de Rabin-Karp tem complexidade $O(mn)$ no pior caso, o mesmo da busca completa, e com *runtime* maior, por conta do cálculo dos *hashes*
- Uma primeira melhoria que pode ser feita é usar o *rolling hash*, e computar $h(S[(i+1)..(i+m)])$ a partir de $h(S[i..(i+m-1)])$ com custo $O(1)$
- Isto é possível, pois se $h_i(S) = h(S[i..(i+m-1)]$, então

$$
\begin{aligned}
h_{i+1}(S) &= \left(S_{i+1} + S_{i+2}p + \ldots + S_{i+m}p^{m-1}\right) \text{ mod } q \\
&= \left(\frac{S_i + S_{i+1}p + \ldots + S_{i+m-1}p^{n-1} + S_{i+m}p^m - S_i}{p}\right) \text{ mod } q \\
&= \left(\frac{S_i + S_{i+1}p + \ldots + S_{i+m-1}p^{n-1} - S_i}{p} + S_{i+m}p^{m-1}\right) \text{ mod } q \\
&= \left(\frac{h_i(S) - S_i}{p} + S_{i+m}p^{m-1}\right) \text{ mod } q
\end{aligned}
$$

Diminuição da complexidade para o cálculo dos *hashes*

- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo q

Diminuição da complexidade para o cálculo dos *hashes*

- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo q
- Assim,

$$h_{i+1}(S) = \left((h_i - S[i])p^{-1} + S_{i+m}p^{m-1}\right) \bmod q$$

Diminuição da complexidade para o cálculo dos *hashes*

- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo q
- Assim,

$$h_{i+1}(S) = \left((h_i - S[i])p^{-1} + S_{i+m}p^{m-1}\right) \bmod q$$

- Se a constante $k \equiv p^{m-1} \pmod{q}$ for precomputada, cada atualização do *hash* tem custo $O(1)$

Diminuição da complexidade para o cálculo dos *hashes*

- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo q
- Assim,

$$h_{i+1}(S) = \left((h_i - S[i])p^{-1} + S_{i+m}p^{m-1}\right) \mod q$$

- Se a constante $k \equiv p^{m-1} \pmod{q}$ for precomputada, cada atualização do *hash* tem custo $O(1)$
- O inverso $i = p^{-1} \pmod{q}$ também pode ser precomputado, como no caso da constante k

Diminuição da complexidade para o cálculo dos *hashes*

- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo q
- Assim,

$$h_{i+1}(S) = \left((h_i - S[i])p^{-1} + S_{i+m}p^{m-1}\right) \mod q$$

- Se a constante $k \equiv p^{m-1} \pmod{q}$ for precomputada, cada atualização do *hash* tem custo $O(1)$
- O inverso $i = p^{-1} \pmod{q}$ também pode ser precomputado, como no caso da constante k
- O pior caso ainda tem complexidade $O(nm)$, mas o caso médio passa a ter complexidade $O(n+m)$

Pseudocódigo do algoritmo de Rabin-Karp

Algoritmo 2 Algoritmo de Rabin-Karp com *Rolling Hash*

Input: Duas strings P e S e os parâmetros p e q do *rolling hash* h

Output: O número de ocorrências occ de P em S

function $\textsc{RabinKarp}(P,S)$
    $m \leftarrow |P|$, $n \leftarrow |S|$, $occ \leftarrow 0$
    $h_P \leftarrow h(P)$, $h_S \leftarrow h(S[1..m])$
    $k \leftarrow p^{m-1} \pmod{q}$, $i \leftarrow p^{-1} \pmod{q}$

    for $i \leftarrow 1$ **to** $n - m + 1$ **do**
        if $h_S = h_P$ **then**
            if $S[i..(i + m - 1)] = P$ **then**
                $occ \leftarrow occ + 1$
        if $i \neq n - m + 1$ **then**
            $h_S \leftarrow (i(h_S - S[i]) + kS[i + m]) \pmod{q}$
    return occ

Implementação do algoritmo de Rabin-Karp em C++

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

using namespace std;
using ll = long long;
const ll p { 31 }, q { 1000000007 };

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

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

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

    return ans;
}
```

Implementação do algoritmo de Rabin-Karp em C++

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

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

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

    return res;
}
```

Implementação do algoritmo de Rabin-Karp em C++

```cpp
int rabin_karp(const std::string& S, const std::string& P)
{
    int n = S.size(), m = P.size(), occ = 0, hs = h(S, m), hp = h(P, m);

    for (int i = 0; i < n - m + 1; ++i)
    {
        occ += (hs == hp && S.substr(i, m) == P) ? 1 : 0;

        if (i != n - m)
        {
            hs = (hs - f(S[i]) + q) % q;
            hs = (hs * fast_mod_pow(p, q - 2)) % q;
            hs = (hs + f(S[i + m]) * fast_mod_pow(p, m - 1)) % q;
        }
    }

    return occ;
}
```

Rabin-Karp com *hashes* perfeitos

- A complexidade do pior caso não se alterou por conta da comparação direta das substrings $S[i..j]$ com o padrão P no caso de colisão

Rabin-Karp com *hashes* perfeitos

- A complexidade do pior caso não se alterou por conta da comparação direta das substrings $S[i..j]$ com o padrão P no caso de colisão
- Esta comparação pode ser eliminada se a função de hash h for perfeita para o conjunto $\mathcal{P}$ tal que

$$\mathcal{P} = \{P\} \cup \{S[i..j] \mid i \in [1, n], j \in [i, n]\}$$

Rabin-Karp com *hashes* perfeitos

- A complexidade do pior caso não se alterou por conta da comparação direta das substrings $S[i..j]$ com o padrão P no caso de colisão
- Esta comparação pode ser eliminada se a função de hash h for perfeita para o conjunto $\mathcal{P}$ tal que

$$\mathcal{P} = \{P\} \cup \{S[i..j] \mid i \in [1, n], j \in [i, n]\}$$

- O uso de *hash* duplo pode auxiliar na obtenção de uma função de *hash* perfeita, uma vez que

$$|\mathcal{P}| \leq \frac{n(n+1)}{2} + 1$$

Rabin-Karp com *hashes* perfeitos

- A complexidade do pior caso não se alterou por conta da comparação direta das substrings $S[i..j]$ com o padrão P no caso de colisão
- Esta comparação pode ser eliminada se a função de hash h for perfeita para o conjunto $\mathcal{P}$ tal que

$$\mathcal{P} = \{P\} \cup \{S[i..j] \mid i \in [1, n], j \in [i, n]\}$$

- O uso de *hash* duplo pode auxiliar na obtenção de uma função de *hash* perfeita, uma vez que

$$|\mathcal{P}| \leq \frac{n(n+1)}{2} + 1$$

- A eliminação desta comparação reduz a complexidade do pior caso para $O(n + m)$

Pseudocódigo do algoritmo de Rabin-Karp

Algoritmo 3 Algoritmo de Rabin-Karp com *hash* perfeito

Input: Duas strings P e S e os parâmetros p, q do *rolling hash* perfeito h

Output: O número de ocorrências occ de P em S

```
function RabinKarp(P,S)
    m ← |P|, n ← |S|, occ ← 0
    h_P ← h(P), h_S ← h(S[1..m])
    k ← p^(m−1) (mod q), i ← p^(−1) (mod q)

    for i ← 1 to n − m + 1 do
        if h_S = h_P then
            occ ← occ + 1
        if i ≠ n − m + 1 then
            h_S ← (i(h_S − S[i]) + kS[i + m]) (mod q)
    return occ
```

Implementação do algoritmo de Rabin-Karp em C++

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

using namespace std;
const long long p1 { 31 }, p2 { 29 }, q1 { 1000000007 }, q2 { 1000000009 };

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

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

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

    return ans;
}
```

Implementação do algoritmo de Rabin-Karp em C++

```cpp
using ii = pair<long long, long long>;

ii h(const string& s, size_t size) { return { hi(s, p1, q1, size), hi(s, p2, q2, size) }; }

long long fast_mod_pow(long long a, long long n, long long q)
{
    long long res = 1, base = a;

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

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

    return res;
}
```

Implementação do algoritmo de Rabin-Karp em C++

```
ii update(const ii& hs, const string& S, size_t i, size_t m)
{
    auto [x, y] = hs;

    x = (x - f(S[i]) + q1) % q1;
    x = (x * fast_mod_pow(p1, q1 - 2, q1)) % q1;
    x = (x + f(S[i + m]) * fast_mod_pow(p1, m - 1, q1)) % q1;

    y = (y - f(S[i]) + q2) % q2;
    y = (y * fast_mod_pow(p2, q2 - 2, q2)) % q2;
    y = (y + f(S[i + m]) * fast_mod_pow(p2, m - 1, q2)) % q2;

    return { x, y };
}

size_t rabin_karp(const std::string& S, const std::string& P)
{
    size_t n = S.size(), m = P.size(), occ = 0;
    auto hs = h(S, m), hp = h(P, m);
```

```cpp
    for (size_t i = 0; i < n - m + 1; ++i)
    {
        occ += (hs == hp) ? 1 : 0;

        if (i != n - m)
            hs = update(hs, S, i, m);
    }

    return occ;
}

int main()
{
    std::cout << rabin_karp("ababababababa", "aba") << '\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.
4. Wikipédia. Rabin-Karp algorithm, acesso em 08/08/2019.