Strings

Algoritmo de Rabin-Karp

Prof. Edson Alves - UnB/FGA

2019

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
- 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, enquanto strings, 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

Output: O número de ocorrências occ de P em S

```
1:  function RabinKarp(P,S)
2:      m ← |P|
3:      n ← |S|
4:      occ ← 0
5:      h_P ← h(P)
6:      for i ← 1 to n − m + 1 do
7:          h_S ← h(S[i..(i + m − 1)])
8:          if h_S = h_P then
9:              if S[i..(i + m − 1)] = P then
10:                 occ ← occ + 1
11:     return occ
```

Implementação do algoritmo de Rabin-Karp em Haskell

```haskell
import Data.Char

f :: Char -> Int
f c = (ord c) - (ord 'a') + 1

h :: String -> Int
h s = sum (zipWith (*) fs ps) `mod` m where
    p = 31
    m = 10^9 + 7
    fs = map f s
    ps = map (\x -> p ^ x) $ take (length s) [0..]

rabin_karp :: String -> String -> Int
rabin_karp s p = sum rs where
    n = length s
    m = length p
    hp = h p
    xss = [take m (drop i s) | i <- [0..(n - m)]]
    rs = [fromEnum (h xs == hp && xs == p) | xs <- xss]

main = print $ rabin_karp "abababababab" "aba"
```

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, m = 1000000007;

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

    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;
}

int main()
{
    auto s = "abababababab", p = "aba";

    std::cout << rabin_karp(s, p) << '\n';

    return 0;
}
```

Pseudocódigo do algoritmo de Rabin-Karp

Algoritmo 2 Algoritmo de Rabin-Karp – Naive

Input: Duas strings P e S

Output: O número de ocorrências occ de P em S

```
1:  function RabinKarp(P,S)
2:      m ← |P|
3:      n ← |S|
4:      occ ← 0
5:      h_P ← h(P)
6:      for i ← 1 to n − m + 1 do
7:          h_S ← h(S[i..(i + m − 1)])
8:          if h_S = h_P then
9:              if S[i..(i + m − 1)] = P then
10:                 occ ← occ + 1
11:     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*
- 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

$$
\begin{aligned}
h(S[(i+1)..(i+m)]) &= S_{i+1} + S_{i+2}p + \ldots + S_{i+m}p^{m-1} \\
&= \frac{S_i + S_{i+1}p + \ldots + S_{i+m-1}p^{n-1} + S_{i+m}p^m - S_i}{p} \\
&= \frac{S_i + S_{i+1}p + \ldots + S_{i+m-1}p^{n-1} - S_i}{p} + S_{i+m}p^{m-1} \\
&= \frac{h(S[i..(i+m-1)]) - S_i}{p} + S_{i+m}p^{m-1}
\end{aligned}
$$

Diminuição da complexidade para o cálculo dos *hashes*

- Como $S_i < p$, para todo i, então

$$h(S[(i+1)..(i+m)]) = \left\lfloor \frac{h(S[i..(i+m-1)])}{p} \right\rfloor + S_{i+m}p^{m-1}$$

- Se a constante $k \equiv p^{m-1} \pmod{m}$ for precomputado, cada atualização do *hash* tem custo $O(1)$
- Observe que a divisão deve ser feita por meio da multiplicação pelo inverso multiplicativo de p módulo m
- Este inverso 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 3 Algoritmo de Rabin-Karp – Naive

Input: Duas strings P e S

Output: O número de ocorrências occ de P em S

```
1:  function RabinKarp(P,S)
2:      m ← |P|
3:      n ← |S|
4:      occ ← 0
5:      h_P ← h(P)
6:      for i ← 1 to n − m + 1 do
7:          h_S ← h(S[i..(i + m − 1)])
8:          if h_S = h_P then
9:              if S[i..(i + m − 1)] = P then
10:                 occ ← occ + 1
11:     return occ
```

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.