

Strings

Algoritmo de Rabin-Karp

Prof. Edson Alves - UnB/FGA

2019

1. Algoritmo de Rabin-Karp
2. Variantes do algoritmo de Rabin-Karp

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 e uma função de *hash* h

Output: O número de ocorrências occ de P em S

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

Implementação do algoritmo de Rabin-Karp 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..]
12
13 rabin_karp :: String -> String -> Int
14 rabin_karp s p = sum rs where
15     n = length s
16     m = length p
17     hp = h p
18     xss = [take m (drop i s) | i <- [0..(n - m)]]
19     rs = [fromEnum (h xs == hp && xs == p) | xs <- xss]
20
21 main = print $ rabin_karp "ababababababa" "aba"
```

Implementação do algoritmo de Rabin-Karp em C++

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

Implementação do algoritmo de Rabin-Karp em C++

```
21 int rabin_karp(const std::string& s, const std::string& p)
22 {
23     int n = s.size(), m = p.size(), occ = 0, hp = h(p);
24
25     for (int i = 0; i <= n - m; i++)
26     {
27         auto b = s.substr(i, m);
28         occ += (h(b) == hp && b == p) ? 1 : 0;
29     }
30
31     return occ;
32 }
33
34 int main()
35 {
36     auto s = "ababababababa", p = "aba";
37
38     std::cout << rabin_karp(s, p) << '\n';
39
40     return 0;
41 }
```


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 se $h_i(S) = h(S[i..(i+m-1)])$, então

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

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
- Assim,

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

- Se a constante $k \equiv p^{m-1} \pmod{q}$ for precomputado, 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

```
1: function RABINKARP( $P, S$ )
2:    $m \leftarrow |P|$ ,  $n \leftarrow |S|$ ,  $occ \leftarrow 0$ 
3:    $h_P \leftarrow h(P)$ ,  $h_S \leftarrow h(S[1..m])$ 
4:    $k \leftarrow p^{m-1} \pmod{q}$ ,  $i \leftarrow p^{-1} \pmod{q}$ 
5:
6:   for  $i \leftarrow 1$  to  $n - m + 1$  do
7:     if  $h_S = h_P$  then
8:       if  $S[i..(i + m - 1)] = P$  then
9:          $occ \leftarrow occ + 1$ 
10:    if  $i \neq n - m + 1$  then
11:       $h_S \leftarrow (i(h_S - S[i]) + kS[i + m]) \pmod{q}$ 
12:  return  $occ$ 
```

Implementação do algoritmo de Rabin-Karp em Haskell

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

Implementação do algoritmo de Rabin-Karp em Haskell

```
19 rolling_hash :: String -> Int -> Char -> Int -> [Int]
20 rolling_hash xs _ _ m | length xs < m = []
21 rolling_hash (x:xs) prev c m = hs : rolling_hash xs hs x m where
22     i = fastExpMod p (q - 2)
23     k = fastExpMod p (m - 1)
24     d = xs !! (m - 2)
25     hs = ((prev - f(c))*i + k*f(d)) `mod` q
26
27 rabin_karp :: String -> String -> Int
28 rabin_karp s p = length $ filter validate (zip ys [0..]) where
29     m = length p
30     hp = h p
31     hs = h $ take m s
32     ys = hs : rolling_hash (tail s) hs (head s) m
33     validate (hb, i) = hb == hp && take m (drop i s) == p
34
35 main :: IO ()
36 main = print $ rabin_karp "ababababababa" "aba"
```

Implementação do algoritmo de Rabin-Karp 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) { return c - 'a' + 1; }
9
10 int h(const string& s, int size)
11 {
12     ll ans = 0;
13
14     for (int i = size - 1; i >= 0; i--)
15     {
16         ans = (ans * p) % q;
17         ans = (ans + f(s[i])) % q;
18     }
19
20     return ans;
21 }
```

Implementação do algoritmo de Rabin-Karp em C++

```
22
23 ll fast_mod_pow(ll a, ll n)
24 {
25     ll res = 1, base = a;
26
27     while (n)
28     {
29         if (n & 1)
30             res = (res * base) % q;
31
32         base = (base * base) % q;
33         n >>= 1;
34     }
35
36     return res;
37 }
38
39 int rabin_karp(const std::string& S, const std::string& P)
40 {
41     int n = S.size(), m = P.size(), occ = 0, hs = h(S, m), hp = h(P, m);
42
```


Implementação do algoritmo de Rabin-Karp em C++

```
43     for (int i = 0; i < n - m + 1; ++i)
44     {
45         occ += (hs == hp && S.substr(i, m) == P) ? 1 : 0;
46
47         if (i != n - m)
48         {
49             hs = (hs - f(S[i]) + q) % q;
50             hs = (hs * fast_mod_pow(p, q - 2)) % q;
51             hs = (hs + f(S[i + m]) * fast_mod_pow(p, m - 1)) % q;
52         }
53     }
54
55     return occ;
56 }
57
58 int main()
59 {
60     std::cout << rabin_karp("ababababababa", "aba") << '\n';
61
62     return 0;
63 }
```

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 $|\mathcal{P}| \leq n + 1$
- 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

```
1: function RABINKARP( $P, S$ )
2:    $m \leftarrow |P|, n \leftarrow |S|, occ \leftarrow 0$ 
3:    $h_P \leftarrow h(P), h_S \leftarrow h(S[1..m])$ 
4:    $k \leftarrow p^{m-1} \pmod{q}, i \leftarrow p^{-1} \pmod{q}$ 
5:
6:   for  $i \leftarrow 1$  to  $n - m + 1$  do
7:     if  $h_S = h_P$  then
8:        $occ \leftarrow occ + 1$ 
9:     if  $i \neq n - m + 1$  then
10:       $h_S \leftarrow (i(h_S - S[i]) + kS[i + m]) \pmod{q}$ 
11:   return  $occ$ 
```

Implementação do algoritmo de Rabin-Karp em Haskell

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

Implementação do algoritmo de Rabin-Karp em Haskell

```
19 rolling_hash :: String -> Int -> Char -> Int -> [Int]
20 rolling_hash xs _ _ m | length xs < m = []
21 rolling_hash (x:xs) prev c m = hs : rolling_hash xs hs x m where
22     i = fastExpMod p (q - 2)
23     k = fastExpMod p (m - 1)
24     d = xs !! (m - 2)
25     hs = ((prev - f(c))*i + k*f(d)) `mod` q
26
27 rabin_karp :: String -> String -> Int
28 rabin_karp s p = length $ filter (==hp) ys where
29     m = length p
30     hp = h p
31     hs = h $ take m s
32     ys = hs : rolling_hash (tail s) hs (head s) m
33
34 main :: IO ()
35 main = print $ rabin_karp "ababababababa" "aba"
```

Implementação do algoritmo de Rabin-Karp 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) { return c - 'a' + 1; }
9
10 int h(const string& s, int size)
11 {
12     ll ans = 0;
13
14     for (int i = size - 1; i >= 0; i--)
15     {
16         ans = (ans * p) % q;
17         ans = (ans + f(s[i])) % q;
18     }
19
20     return ans;
21 }
```

Implementação do algoritmo de Rabin-Karp em C++

```
22
23 ll fast_mod_pow(ll a, ll n)
24 {
25     ll res = 1, base = a;
26
27     while (n)
28     {
29         if (n & 1)
30             res = (res * base) % q;
31
32         base = (base * base) % q;
33         n >>= 1;
34     }
35
36     return res;
37 }
38
39 int rabin_karp(const std::string& S, const std::string& P)
40 {
41     int n = S.size(), m = P.size(), occ = 0, hs = h(S, m), hp = h(P, m);
42
```

Implementação do algoritmo de Rabin-Karp em C++

```
43     for (int i = 0; i < n - m + 1; ++i)
44     {
45         occ += (hs == hp) ? 1 : 0;
46
47         if (i != n - m)
48         {
49             hs = (hs - f(S[i]) + q) % q;
50             hs = (hs * fast_mod_pow(p, q - 2)) % q;
51             hs = (hs + f(S[i + m]) * fast_mod_pow(p, m - 1)) % q;
52         }
53     }
54
55     return occ;
56 }
57
58 int main()
59 {
60     std::cout << rabin_karp("ababababababa", "aba") << '\n';
61
62     return 0;
63 }
```


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.