<https://zhuanlan.zhihu.com/p/563551141>

Rabin-Karp 算法（也可以叫 Karp-Rabin 算法），由 Richard M. Karp 和 Michael O. Rabin 在 1987 年发表，它也是用来解决多模式串匹配问题的。

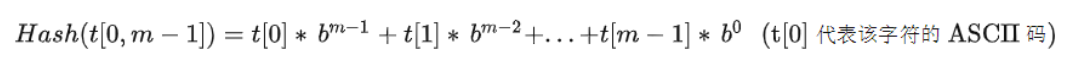
它的实现方式有点与众不同，首先是计算两个字符串的哈希值，然后通过比较这两个哈希值的大小来判断是否出现匹配。

### 算法分析与实现

选择一个合适的哈希函数很重要。假设文本串为 t[0, n)，模式串为 p[0, m)，其中 0 < m < n, 另Hash(t[i, j])代表字符串 t[i, j] 的哈希值。

当 Hash(t[0, m - 1]) != Hash(p[0, m - 1])时，我们很自然的会把 Hash(t[1, m])拿过来继续比较。在这个过程中，若我们重新计算字符串t[1, m]的哈希值，还需要 O(n)的时间复杂度，不划算。观察到字符串 t[0, m - 1]与 t[1, m]中有 m-1 个字符是重合的，因此我们可以选用滚动哈希函数，那么重新计算的时间复杂度就降为 O(1)。

Rabin-Karp 算法选用的滚动哈希函数主要是利用 [Rabin fingerprint](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Rabin_fingerprint) 的思想，举个例子，计算字符串 t[0, m - 1] 的哈希值的公式如下，



Hash(t[0, m - 1]) = t[0] \* b^(m - 1) + t[1] \* b^(m - 2) + ... + t[m - 1] \* b^0, t[0]代表该字符的ASCII码

其中的 b是一个常数，在 Rabin-Karp 算法中，我们一般取值为 256，因为一个字符的最大值不超过 255。上面的公式还有一个问题，哈希值如果过大可能会溢出，因此我们还需要对其取模，这个值应该尽可能大，且是质数，这样可以减小哈希碰撞的概率，在这里我们就取 101。

则计算字符串 t[1, m] 的哈希值公式如下，



  Hash(t[1, m]) = (Hash(t[0, m - 1]) - t[0] \* b^(m - 1)) \* b + t[m] \* b^0 请仔细对比上式

完整代码如下，

#include <iostream>

#include <string.h>

using namespace std;

#define BASE 256

#define MODULUS 101

void RabinKarp(char t[], char p[])

{

int t\_len = strlen(t);

int p\_len = strlen(p);

// 哈希滚动之用

int h = 1;

for (int i = 0; i < p\_len - 1; i++)

h = (h \* BASE) % MODULUS;

int t\_hash = 0;

int p\_hash = 0;

for (int i = 0; i < p\_len; i++)

{

t\_hash = (BASE \* t\_hash + t[i]) % MODULUS;

p\_hash = (BASE \* p\_hash + p[i]) % MODULUS;

}

int i = 0;

while (i <= t\_len - p\_len)

{

// 考虑到哈希碰撞的可能性，还需要用 memcmp 再比对一下

if (t\_hash == p\_hash && memcmp(p, t + i, p\_len) == 0)

cout << p << " is found at index " << i << endl;

// 哈希滚动

t\_hash = (BASE \* (t\_hash - t[i] \* h) + t[i + p\_len]) % MODULUS;

// 防止出现负数

if (t\_hash < 0)

t\_hash = t\_hash + MODULUS;

i++;

}

}

int main()

{

char t[100] = "It is a test, but not just a test";

char p[10] = "test";

RabinKarp(t, p);

return 0;

}

  输出如下，

test is found at index 8

test is found at index 29

### 长度为p\_len(等于模式串长度)的文本串(初始子字符串)的t\_hash代码实现逻辑

int t\_hash = 0;

for (int i = 0; i < p\_len; i++)

{

t\_hash = (BASE \* t\_hash + t[i]) % MODULUS;

}

======================================================================================================================

i = 0 -> t\_hash = (BASE \* 0 + t[0]) % MODULUS = t[0] % MODULUS

i = 1 -> t\_hash = (BASE \* t[0] % MODULUS + t[1]) % MODULUS = BASE \* t[0] % MODULUS + t[1] % MODULUS

i = 2 -> t\_hash = (BASE \* ((BASE \* t[0] % MODULUS + t[1]) % MODULUS) + t[2]) % MODULUS = BASE^2 \* t[0] % MODULUS + BASE \* t[1] % MODULUS + t[2] % MODULUS

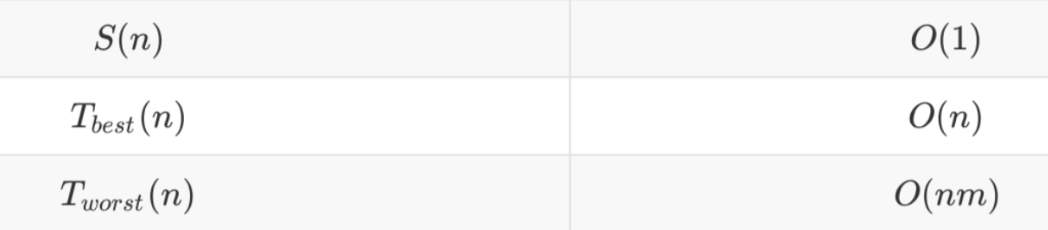
....

i = p\_len - 1 -> t\_hash = BASE^(p\_len - 1) \* t[0] + BASE^(p\_len - 2) \* t[1] + ... + t[p\_len - 1] % MODULUS

### 复杂度分析

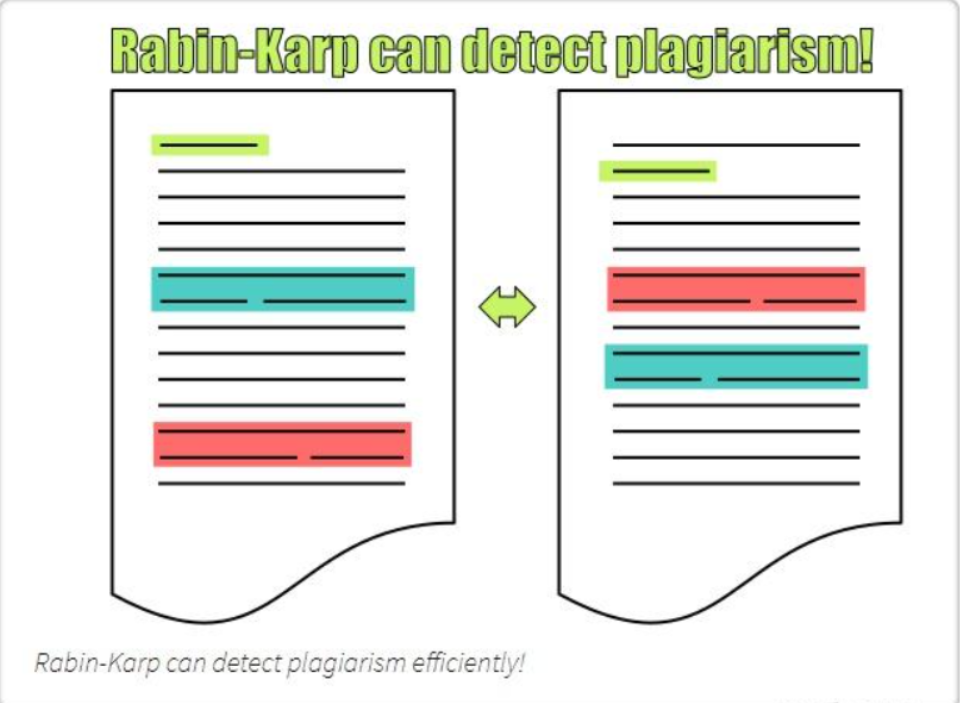
首先看空间复杂度，很容易判断，S(n) = O(1)

再来看时间复杂度，取文本串长度为 n，模式串长度为 m，预处理需要O(m), 在匹配过程中，最佳情况下，未出现哈希碰撞，Tbest(n) = O(n - m)。最坏情况下，每次都出现碰撞，Tworst(n) = O((n - m) \* m)，因为在现实生活中，n 往往远大于 m，因此最后的复杂度表格为，



### 应用分析

Rabin-Karp 算法主要用来检测文章抄袭，比如 [Semantic Scholar](https://link.zhihu.com/?target=https://www.semanticscholar.org/) 的检测系统。



但从上面的复杂度数据来看，Rabin-Karp 算法并无多大优势，用于检测文本抄袭可行么？然而从实际使用中反馈的结果表明，检测抄袭的时间复杂度只有O(n)，我觉得主要是因为以下两点，

* 现实生活中所作的文章，其文本数据并不会出现我们想象中的那么多次的哈希碰撞。
* 很大概率上，被提交的一篇文章，原创的篇幅肯定是远远大于抄袭的篇幅的，也就是说并不会出现我们想象中的那么多次的成功匹配。

### 参考

[Rabin–Karp algorithm](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm)

[Searching for Patterns | Set 3 (Rabin-Karp Algorithm)](https://link.zhihu.com/?target=https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/)

[Computer Algorithms: Rabin-Karp String Searching](https://link.zhihu.com/?target=http://www.stoimen.com/blog/2012/04/02/computer-algorithms-rabin-karp-string-searching/)

**具体例子的Java实现**

**Refer to**

[https://coolcao.com/2020/08/20/rabin-karp/#:~:text=Rabin%20Karp%E7%AE%97%E6%B3%95%E5%8D%B3%E6%98%AF,%E5%88%A4%E6%96%AD%E5%85%B6%E6%98%AF%E5%90%A6%E7%9C%9F%E6%AD%A3%E7%9B%B8%E7%AD%89%E3%80%82](https://coolcao.com/2020/08/20/rabin-karp/" \l ":~:text=Rabin Karp%E7%AE%97%E6%B3%95%E5%8D%B3%E6%98%AF,%E5%88%A4%E6%96%AD%E5%85%B6%E6%98%AF%E5%90%A6%E7%9C%9F%E6%AD%A3%E7%9B%B8%E7%AD%89%E3%80%82)

# 字符串匹配算法-Rabin Karp算法

一个很简单的问题，给定一个字符串txt和一个模式串pat，写一个函数search来输出字符串txt中所有和pat相等的子串。

例如，给定txt=”this is a test text”， pat=”text”，返回 [10]

这个问题非常简单，我们只需要暴力穷举所有txt中所有长度等于len(pat)的子串，并判断其是否和pat相等即可。相等就返回其起始索引。

在判断子串是否和pat相等时，我们需要逐位去比较。这样，这个问题的复杂度就是 O(mn)。这种算法我们称为*朴素字符串匹配算法*。

很显然，这样的算法效率并不高。

Rabin Karp算法即是解决这个问题的更高效的一个算法。它的思想是，不直接逐位对比模式串pat和text是否相等，而是利用哈希算法，计算模式串和子串的哈希值，如果哈希值不相等，那么很明显字符串也不相等，如果相等，由于哈希算法可能会存在哈希冲突的可能，因此我们需要再使用朴素算法判断其是否真正相等。

可能有的人会问，计算字符串的哈希值，不也要逐位计算，然后才能算出一个哈希值，那这样复杂度不还是O(mn)么？

对了，Rabin Karp算法的核心是，将哈希函数使用滚动哈希来计算，这样计算哈希的复杂度是O(1)，整体的复杂度就变成了O(m)了。

这里我们先把Rabin Karp使用的哈希算法放一边，思考一下如果是我们自己，应该怎样选择一个哈希算法呢？

对于一个只含有数字的字符串“123”来说，普通人第一眼的感觉不应该是，这是一个数字，123。

而且字符串”123”和字符串”234”是不相等的，因为其代表的数字123和234是不相等的。

这里就有点上面所说的哈希的意味了，我们不直接对比字符串的每个字符，而是比较其所代表的数字是否相等即可。

我们知道，对于一个只有数字的字符串a[0]a[1]a[2]...a[n]要转换成十进制的数字，公式如下：

num = a[n]\*10^0 + a[n-1]\*10^1 + a[n-2]\*10^2 + ... + a[1]\*10^(n-1) + a[0]\*10^n

在此基础上，我们往外延伸，如果对于一个只有小写英文字符的字符串来说，我们是不是可以当成26进制，然后计算出一个字符串所代表的数字：

num = a[n]\*26^0 + a[n-1]\*26^1 + a[n-2]\*26^2 + ... + a[1]\*26^(n-1) + a[0]\*26^n

其中上面的这个公式，我们就可以作为计算哈希的公式。使用这个计算哈希会把复杂度降到O(1)么?

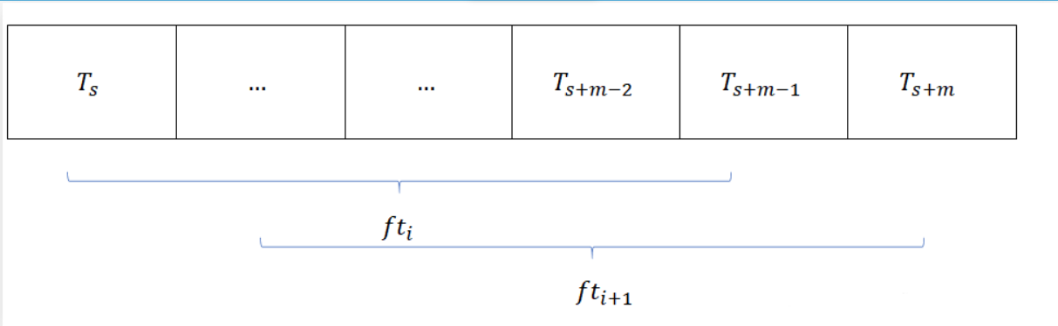
会，我们来看一下为什么。

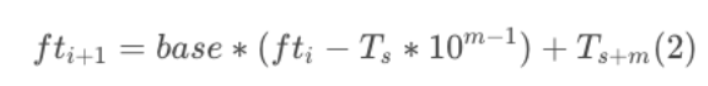
比如对于一个字符串”abcd”，我们要计算一个长度为2的子串的子串的哈希，先计算”ab”的：hash1 = code(a)\*26^1 + code(b)\*26^0

再计算”bc”的：hash2 = code(b)\*26^1 + code(c)\*26^0

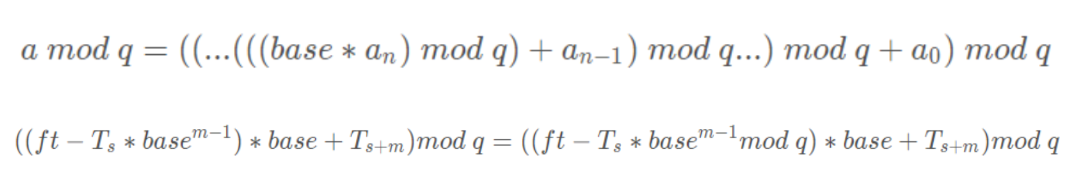
看一下这里的规律，在计算hash2时，我们完全可以复用hash1的值，hash2=(hash1-code(a)\*26^1)\*26+code(c)\*26^0

已知前一个子串的哈希值，计算后一个哈希值的过程，如下：





当然，如果字符串过长，最后计算哈希可能会溢出。为了解决这个问题，在Rabin-Karp算法中，求哈希时，使用取余。



即：

hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q

hash( txt[s .. s+m-1] ) : Hash value at shift s.

hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

**Java代码实现：**

// Following program is a Java implementation

// of Rabin Karp Algorithm given in the CLRS book

public class Main

{

// d is the number of characters in the input alphabet

public final static int d = 256;

/\* pat -> pattern

txt -> text

q -> A prime number

\*/

static void search(String pat, String txt, int q)

{

int M = pat.length();

int N = txt.length();

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

// The value of h would be "pow(d, M-1)%q"

for (i = 0; i < M-1; i++)

h = (h\*d)%q;

// Calculate the hash value of pattern and first

// window of text

for (i = 0; i < M; i++)

{

p = (d\*p + pat.charAt(i))%q;

t = (d\*t + txt.charAt(i))%q;

}

// Slide the pattern over text one by one

for (i = 0; i <= N - M; i++)

{

// Check the hash values of current window of text

// and pattern. If the hash values match then only

// check for characters on by one

if ( p == t )

{

/\* Check for characters one by one \*/

for (j = 0; j < M; j++)

{

if (txt.charAt(i+j) != pat.charAt(j))

break;

}

// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

if (j == M)

System.out.println("Pattern found at index " + i);

}

// Calculate hash value for next window of text: Remove

// leading digit, add trailing digit

if ( i < N-M )

{

t = (d\*(t - txt.charAt(i)\*h) + txt.charAt(i+M))%q;

// We might get negative value of t, converting it

// to positive

if (t < 0)

t = (t + q);

}

}

}

/\* Driver program to test above function \*/

public static void main(String[] args)

{

String txt = "GEEKS FOR GEEKS";

String pat = "GEEK";

int q = 101; // A prime number

search(pat, txt, q);

}

}

// This code is contributed by nuclode