# **字符串匹配算法**

1. **朴素算法（Naive Algorithm）**

朴素模式匹配算法，简单的说就是循环把主串的每个字符作为开头，与子串去进行匹配。对主串做大循环，每个字符为开头做子串（要匹配的字符串）的小循环，如果对应字符匹配，则两字符串都向后移位，否则子串又从子串的开头开始与主串前一步比较的字符开头的下一位继续匹配，直到匹配成功或（主串）遍历完成。比如主串为“googldgfegogegoogleglgoogegooglegoo”，子串为“google”。首先从主串头字符‘g’开始与子串匹配，接下来的1,2,3,4都与子串匹配，但第5位不匹配，于是又以主串的下一个字符‘o’开始，与子串匹配。一直这样循环下去。

**2.Knuth-Morris-Pratt 算法（即 KMP Algorithm）**

KMP算法是一种改进的字符串匹配算法,其关键是利用匹配失败后的信息,尽量减少模式串与主串的匹配次数以达到快速匹配的目的。求得模式的特征向量之后，基于特征分析的快速模式匹配算法(KMP模式匹配算法)与朴素匹配算法类似，只是在每次匹配过程中发生某次失配时，不再单纯地把模式后移一位，而是根据当前字符的特征数来决定模式右移的位数。

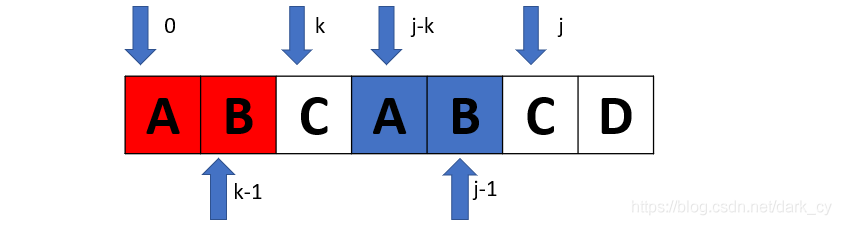
对于每模式串 t 的每个元素 t j，都存在一个实数 k ，使得模式串 t 开头的 k 个字符（t 0 t 1…t k-1）依次与 t j 前面的 k（t j-k t j-k+1…t j-1，这里第一个字符 t j-k 最多从 t 1 开始，所以 k < j）个字符相同。如果这样的 k 有多个，则取最大的一个。模式串 t 中每个位置 j 的字符都有这种信息，采用 next 数组表示，即 next[ j ]=MAX{ k }。

下面咱们分三种情况来讲 next 的求解过程

1.特殊情况

当 j 的值为 0 或 1 的时候，它们的 k 值都为 0，即 next[0] = 0、next[1] =0。但是为了后面 k 值计算的方便，我们将 next[0] 的值设置成 -1。

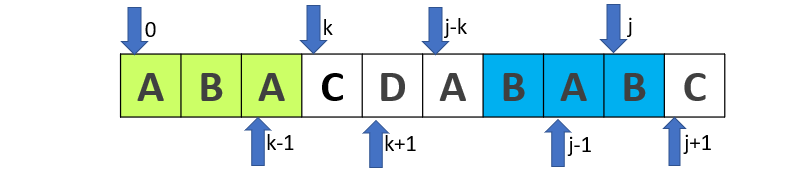
2.当 t[j] == t[k] 的情况



观察上图可知，当 t[j] == t[k] 时，必然有"t[0]…t[k-1]" == " t[j-k]…t[j-1]"，此时的 k 即是相同子串的长度。因为有"t[0]…t[k-1]" == " t[j-k]…t[j-1]"，且 t[j] == t[k]，则有"t[0]…t[k]" == " t[j-k]…t[j]"，这样也就得出了next[j+1]=k+1。

3.当t[j] != t[k] 的情况

关于这种情况，在代码中的描述就是“简单”的一句 k = next[k]；。



　　 由第2中情况可知，当 t[j] == t[k] 时，t[j+1] 的最大子串的长度为 k，即 next[j+1] = k+1。但是此时t[j] != t[k] 了，所以就有 next[j+1] < k，那么求 next[j+1] 就等同于求 t[j] 往前小于 k 个的字符（包括t[j]，看上图蓝色框框）与 t[k] 前面的字符（绿色框框）的最长重合串，即 t[j-k+1] ~ t[j] 与 t[0] ~ t[k-1] 的最长重合串（这里所说“最长重合串”实不严谨，但你知道是符合 k 的子串就行…），那么就相当于求 next[k]（只不过 t[k] 变成了 t[j],但是 next[k] 的值与 t[k] 无关）!!!。所以才有了这句 k = next[k]，如果新的一轮循环（这时 k = next[k] ，j 不变）中 t[j] 依然不等于 t[k] ，则说明倒数第二大 t[0~next[k]-1] 也不行，那么 k 会继续被 next[k] 赋值（这就是所谓的 k 回退…），直到找到符合重合的子串或者 k == -1。

**3.Rabin-Karp 算法**

如果待匹配字符串的长度为M，目标字符串的长度为N（N>M）；首先计算待匹配字符串的hash值，计算目标字符串前M个字符的hash值；比较前面计算的两个hash值，比較次数N-M+1：若hash值不相等，则继续计算目标字符串的下一个长度为M的字符子串的hash值；若hash值同样。则须要使用朴素算法再次推断是否为同样的字串。

**算法实现**

**void Rabin\_Karp\_search(const string &T, const string &P, int d, int q)**

**{**

**int m = P.length();**

**int n = T.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 + P[i])%q;**

**t = (d\*t + T[i])%q;**

**}**

**// Slide the pattern over text one by one**

**for (i = 0; i <= n - m; i++)**

**{**

**// Chaeck 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 (T[i+j] != P[j])**

**break;**

**if (j == m) // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]**

**cout<<"Pattern found at index :"<< i<<endl;**

**}**

**// Calulate hash value for next window of text: Remove leading digit,**

**// add trailing digit**

**if ( i < n-m )**

**{**

**t = (d\*(t - T[i]\*h) + T[i+m])%q;**

**// We might get negative value of t, converting it to positive**

**if(t < 0)**

**t = (t + q);**

**}**

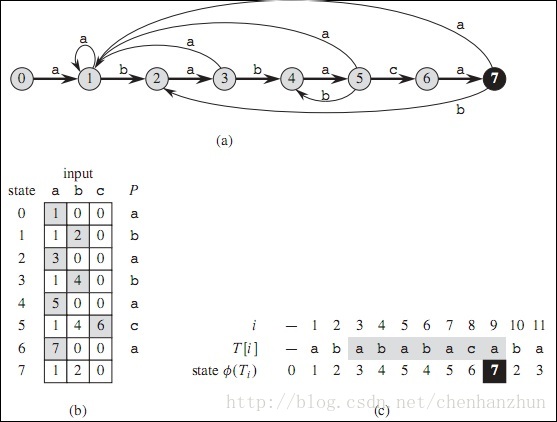
**}**

**}**

**4.有限自动机算法（Finite Automation）**

有限自动机(Finite Automata)字符串匹配算法最主要的是计算出转移函数。即给定一个当前状态k和一个字符x，计算下一个状态；计算方法为：找出模式pat的最长前缀prefix，同时也是pat[0...k-1]x(注意：字符串下标是从0开始)的后缀，则prefix的长度即为下一个状态。匹配的过程是比较输入文本子串和模式串的状态值，若相等则存在，若不想等则不存在。若模式串pat的长度为m，则状态值为0-m，即有m+1个状态，初始状态为0。其中numbers=NO\_OF\_CHARS为输入字符表的个数。

以下是模式串为P=abababaca的自动机执行过程



1. **Boyer-Moore 算法**

常规的字符串匹配算法是从左往右的，这也比较符合我们一贯的思维，但是BM算法是从右往左的。一般匹配我们用的是蛮力匹配，而经典的BM算法其实是对后缀蛮力匹配算法的改进。蛮力后缀匹配每当失匹的时候，就会往后移一位；而BM算法所做的就是改进这一部分，即模式串不在每次只移动一步，而是根据已经匹配的后缀信息，来判断移动的距离，通常80%左右能够移动模式串的长度，从而可以跳过大量不必须比较的字符，大大提高了查找效率。为了实现更快的移动模式串，BM定义了两个规则，坏后缀规则和好后缀规则。这两个规则分别计算我们能够向后移动模式串长度，然后选取这两个规则中移动大的，作为我们真正移动的距离。也就是移动距离不在每次加一，而是加上上面两个规则中移动长度大的。

**6.Simon 算法**

A（x）最小[确定性有限自动机](http://www.darkridge.com/~jpr5/mirror/string/node4.html" \l "SECTION0040)识别IMG_256\*x的经济实现;西蒙注意到A（x）中只有几条重要边) ;

它们是：

从长度 k 的 x 的前缀到长度 k+1 的前缀，对于 0 IMG_260 k < m 的正向边。正好有m这样的边缘;

从长度 k 的 x 的前缀到较小的非零长度前缀的后向边缘。此类边的数量以 m 为界。

其他边通向初始状态，然后可以推导。因此，有效边的数量以 2m 为界。然后，对于自动机的每个状态，只需要存储其重要传出边缘的列表。

每个状态都由其关联前缀的长度减去 1 表示，以便每个指向状态 i 的边（-1 IMG_262 i IMG_263 m-1）由 x[i] 标记，因此不必存储边的标签。正边可以很容易地从图案中推断出来，因此它们不会被存储。它只保留用于存储重要的后向边缘。

我们使用大小为 m-2 的链接列表的表 L。元素 L[i] 给出了从状态 i 开始的边的目标列表。为了避免存储状态 m-1 的列表，在计算此表 L 的过程中，计算整数时，将计算整数IMG_264，使 IMG_265+1 是 x 的最长边界的长度。

西蒙算法的预处理阶段包括计算表 L 和整数IMG_266。它可以在O（m）空间和时间复杂性中完成。

搜索阶段类似于使用自动机的搜索阶段。当找到该模式的出现时，将使用状态 更新当前状态IMG_267。此阶段可以在O（m +n）时间内执行。Simon 算法在搜索阶段最多执行 2个 n-1 文本字符比较。延迟（单个文本字符的最大比较次数）以 min{1+log2（m）， IMG_268} 为界。

**7.Colussi 算法**

模式位置的集合分为两个不相交的子集。然后，每次尝试都分为两个阶段。在第一阶段，从左到右执行比较，文本字符与 kmpNext 函数的值严格大于 -1 的模式位置对齐。这些位置称为空洞;第二阶段包括从右到左比较剩余的位置（称为孔）。此策略具有两个优点：当在第一阶段发生不匹配时，在适当的偏移之后，没有必要比较与上一次尝试中比较的noholes对齐的文本字符;当在第二阶段发生不匹配时，这意味着模式的后缀与文本的因子匹配，在相应的移位后，模式的前缀仍将与文本的因子匹配，则无需再次比较此因子。

1. **Galil-Giancarlo 算法**

加利尔-吉安卡洛算法是[科鲁西](http://www.darkridge.com/~jpr5/mirror/string/node10.html" \l "SECTION00100)算法的变体。此更改会在搜索阶段进行干预。当 x 不是单个字符的幂时，该方法适用。设IMG_259为模式中的最后一个索引，使得对于 0 IMG_260 i IMG_261 IMG_262， x[0]=x[i] 和 x[0] IMG_263 x[IMG_264+1]。假设在上一次尝试中，所有 nohole 都已匹配，并且模式的后缀已匹配，这意味着在相应的移位之后，模式的前缀仍将匹配文本的一部分。因此，窗口位于文本因子 y[j ..j+m-1] 和部分 y[j ..最后]匹配 x[0 ..最后一个-j]。然后在下一次尝试期间，算法将扫描以y[last+1]开头的文本字符，直到到达文本的末尾或找到字符x[0] IMG_265 y[j +k]。

在后一种情况下，可能会出现两个子情况：x[IMG_267+1] IMG_268 y[j+k] 或太小的 x[0] 已被找到 （k IMG_269 IMG_270），然后窗口被移位并定位在文本因子 y[k+1 ..k+m]，文本的扫描恢复（如在Colussi算法中），第一个nohole和模式的记忆前缀是空字。x[IMG_272+1]=y[j+k] 并且已经找到了足够的 x[0]（k >IMG_273），然后窗口被移位并定位在文本因子 y[k-IMG_274-1 ..k-IMG_275+m-2]，文本的扫描恢复（如在 Colussi 算法中），第二个空孔（x[IMG_276+1] 是第一个），并且模式的记忆前缀是 x[0 .。IMG_277+1].预处理阶段与 Colussi 算法完全相同，可以在 O（m） 空间和时间内完成。然后，搜索阶段可以在O（n）时间复杂度内完成。

## **Apostolico-Crochemore 算法**

使徒-克罗切莫尔使用 kmpNext 移位表来计算移位。

设 IMG_257=0，如果 x 是单个字符的幂（x=cm，c inIMG_258），并且IMG_259等于 x 的第一个字符的位置，否则（x=aIMG_260bu 表示 a，b in IMG_261，u in IMG_262\* 和a IMG_263 ）在每次尝试期间，将按以下顺序与形态位置进行比较： IMG_264， IMG_265+1， ... ， m-2， m-1， 0， 1， ... ， IMG_266-1.

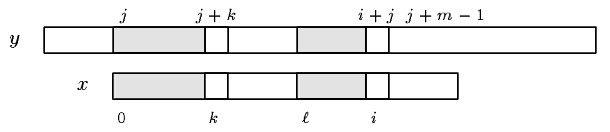
在搜索阶段，我们考虑形式（i，j，k）的三重，其中：

1.窗口位于文本因子 y[j ..j+m-1]

2.0 IMG_269 k IMG_270 IMG_271 和 x[0 ..k-1]=y[j ..j+k-1];

3.IMG_273 IMG_274 i< m 和 x[IMG_275 ..i-1]=y[j+IMG_276 ..i+j-1]。

初始三元组为 （IMG_277，0，0）。

    
图 11.1：在每次尝试使徒-克罗切莫尔算法时，我们都会考虑一个三重（i，j，k）。

我们现在解释如何在计算（i，j，k）后计算下一个三元组。

根据 i 的值，会出现三种情况：

1.i = IMG_280  
如果 x[i] = y[i+j] 则下一个三元组是 （i+1， j， k）。  
如果 x[i] IMG_281 y[i+j] 则下一个三元组是 （IMG_282， j+1， 最大值{0， k-1}）。

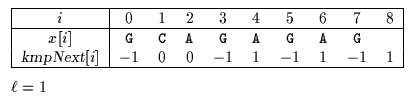
2.IMG_284 < i < m  
如果 x[i] = y[i+j] 则下一个三元组是 （i+1， j， k）。  
如果 x[i] IMG_285 y[i+j] 则根据 kmp 的值出现两种情况：

千米下一个[i] IMG_286 IMG_287： 则下一个三元组是 （IMG_288， i+j-千米下一个 [i]， max{0， 千米下一个 [i]]})

>下一个IMG_289三元组：那么下一个三元组是 （kmpNext[i]， i+j-kmp下一个 [i]，IMG_290)

3.i =m  
如果 k <IMG_292并且 x[k]=y[j+k]，则下一个三元组是 （i， j， k+1）。  
否则，要么 k <IMG_293，要么 x[k] IMG_294 y[j+k]，要么 k=IMG_295。如果 k=IMG_296 报告 x 的出现。在这两种情况下，下一个三元组的计算方式与IMG_297< i < m 的情况相同。

预处理阶段包括计算表 kmpNext 和整数 IMG_298。它可以在O（m）空间和时间中完成。搜索阶段处于O（n）时间复杂度.



**10.Horspool 算法**

Boyer-Moore 算法中使用的坏字符偏移对于小字母表来说不是很有效，但是当字母表与模式的长度相比较大时，就像 ASCII 表和在文本编辑器下进行的普通搜索经常发生的情况一样，它变得非常有用。  
在实践中单独使用它会产生一种非常有效的算法。Horspool建议仅使用窗口最右边字符的坏字符偏移来计算Boyer-Moore算法中的偏移。

预处理阶段采用 O（m+IMG_260） 时间和 O（IMG_261） 空间复杂度。

搜索阶段有一个二次最坏的情况，但可以证明一个文本字符的平均比较次数在1/IMG_262和2/（IMG_263+1）之间。

IMG_256

**11.Shift-Or 算法**

移位或算法使用按位技术。设 R 为大小为 m 的位数组。向量 Rj 是处理文本字符 y[j] 后数组 R 的值（参见图 [5.1](http://www.darkridge.com/~jpr5/mirror/string/node6.html" \l "so)）。它包含有关 x 前缀的所有匹配项的信息，这些前缀在文本中 0 < i IMG_257 m-1 的位置 j 处结束：

IMG_258

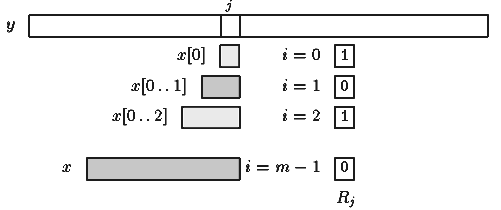


图 5.1：向量 Rj 的含义。

向量 Rj+1 可以在 Rj 之后计算，如下所示。对于每个 Rj[i]=0：  
IMG_260  
和  
IMG_261

如果 Rj+1[m-1]=0，则可以报告完全匹配。

从 Rj 到 Rj+1 的转换可以非常快地计算如下：对于 中的每个 IMG_262c，设 Sc 是大小为 m 的位数组，使得：对于 0 IMG_263 i < m-1，Sc[i]=0 iff x[i]=c。

数组 Sc 表示字符 c 在模式 x 中的位置。每个 Sc 都可以在搜索之前进行预处理。Rj+1 的计算简化为两个运算，移位和或：Rj+1=移位（Rj） 或 Sy[j+1]

假设模式长度不大于机器的内存字大小，则预处理阶段的空间和时间复杂度为O（m+IMG_264）。

搜索阶段的时间复杂度为O（n），因此与字母大小和图案长度无关。

**12.Sunday 算法**

Sunday算法是从前往后匹配，在匹配失败时关注的是主串中参加匹配的最末位字符的下一位字符。

如果该字符没有在模式串中出现则直接跳过，即移动位数 = 模式串长度 + 1；

否则，其移动位数 = 模式串长度 - 该字符最右出现的位置(以0开始) = 模式串中该字符最右出现的位置到尾部的距离 + 1。缘于Sunday算法每一步的移动量都比较大，效率很高。

**算法实现**

**const int maxNum = 1005;**

**int shift[maxNum];**

**int Sunday(const string& T, const string& P) {**

**int n = T.length();**

**int m = P.length();**

**// 默认值，移动m+1位**

**for(int i = 0; i < maxNum; i++) {**

**shift[i] = m + 1;**

**}**

**// 模式串P中每个字母出现的最后的下标**

**// 所对应的主串参与匹配的最末位字符的下一位字符移动到该位，所需要的移动位数**

**for(int i = 0; i < m; i++) {**

**shift[P[i]] = m - i;**

**}**

**// 模式串开始位置在主串的哪里**

**int s = 0;**

**// 模式串已经匹配到的位置**

**int j;**

**while(s <= n - m) {**

**j = 0;**

**while(T[s + j] == P[j]) {**

**j++;**

**// 匹配成功**

**if(j >= m) {**

**return s;**

**}**

**}**

**// 找到主串中当前跟模式串匹配的最末字符的下一个字符**

**// 在模式串中出现最后的位置**

**// 所需要从(模式串末尾+1)移动到该位置的步数**

**s += shift[T[s + m]];**

**}**

**return -1;**

**}**