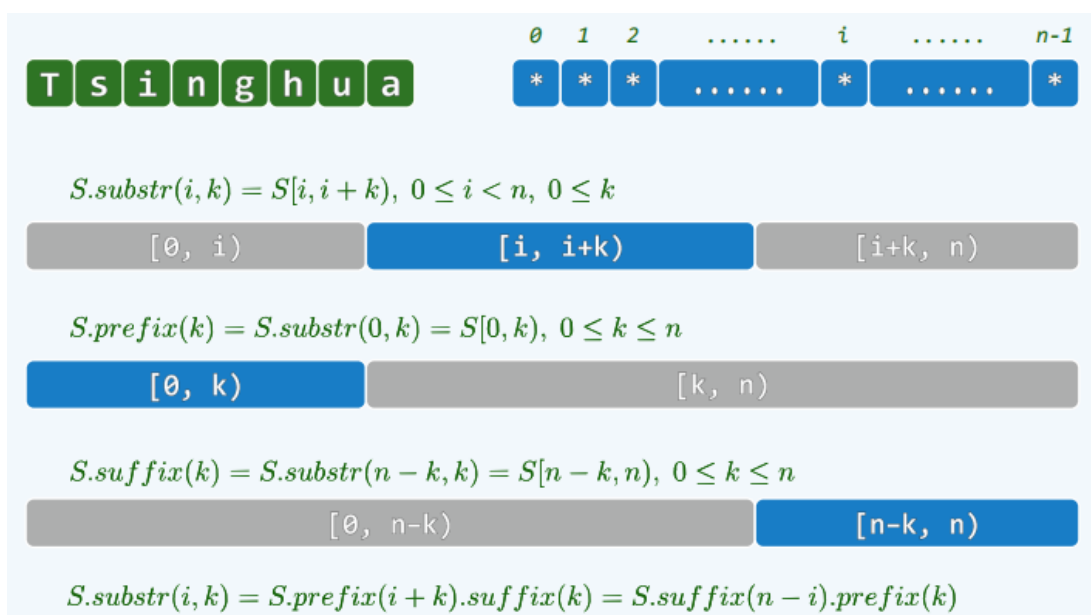


# 串

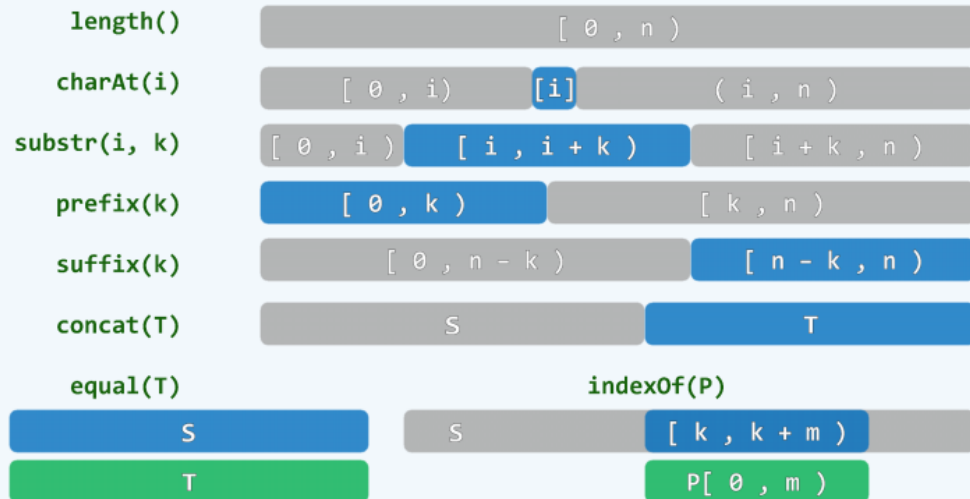
## 串的定义

1. 串（也叫字符串）属于线性结构，是由指定字符表中的字符所组成的有限序列。比如英文文章就是字符串，其字符表就是 {26个字母，标点符号}；计算机中存储的数据也是字符串，其字符表就是 {0, 1}。
2. 一个字符表的大小（即对应字符串的可用字符种类）通常远远小于串的长度（即所含字符总数）。
3. 将长度为n的字符串记作 $S[0, n)$ ，其中秩为k的字符记作 $S[k]$ ：



- 子串：字符串中任一连续的片段，记作 $S[i, i+k)$ ，表明这是起始于i且长度为k的子串；
  - 前缀：起始于秩0、长度为k的子串，记作 $S[0, k)$ ；
  - 后缀：终止于秩n-1、长度为k的子串，记作 $S[n-k, n)$ ；
  - 空串：不包含任何字符（包括空格），长度为零的串；
  - 空串是任何字符串的子串、前缀和后缀；任何字符串都是自己的子串、前缀和后缀。除了字符串本身之外的所有非空子串、前缀和后缀分别称作真子串、真前缀和真后缀。
4. 判等：字符串 $S[0, n)$ 和 $T[0, m)$ 相等，当且仅当二者长度相等（ $m=n$ ），且对应字符分别相同（ $S[i]=T[i]$ ）。

## 串的接口



## 实例

```
❖ "data structures".length() = 15
"data structures".charAt(5) = 's'
"data structures".prefix(4) = "data"
"data structures".suffix(10) = "structures"
"data structures".concat(" & algorithms") = "data structures & algorithms"
"algorithms".equal("data structures") = false
"data structures and algorithms".indexOf("string") = -1
"data structures and algorithms".indexOf("algorithm") = 20
❖ 以下，直接利用字符数组实现字符串，转而重点讨论串匹配算法
```

## 串匹配定义

如何在字符串数据中，检测和提取以字符串形式给出的某一局部特征

这类操作都属于串模式匹配（string pattern matching）范畴，简称串匹配。一般地，即：

对基于同一字符表的任何文本串  $T$  ( $|T| = n$ ) 和模式串  $P$  ( $|P| = m$ )：

判定  $T$  中是否存在某一子串与  $P$  相同

若存在（匹配），则报告该子串在  $T$  中的起始位置

串的长度  $n$  和  $m$  本身通常都很大，但相对而言  $n$  更大，即满足  $2 \ll m \ll n$ 。比如，若：

$T = \text{"Now is the time for all good people to come"}$

$P = \text{"people"}$

则匹配的位置应该是  $T.\text{indexOf}(P) = 29$ 。

## 串匹配四个层次

1. detection：匹配字符串是否出现在目标字符串中？
2. location：首次出现的位置在哪里？
3. counting：总共出现了多少次？
4. enumeration：各出现在哪里？

## 串匹配算法的评价标准

### 1. 随机产生文本串+随机产生模式串

这使得无法考虑到串匹配成功的情况，因为此时串匹配的成功率极低：

以基于字符表  $\Sigma = \{ 0, 1 \}$  的二进制串为例。任给长度为  $n$  的文本串，其中长度为  $m$  的子串不过  $n - m + 1$  个 ( $m \ll n$  时接近于  $n$  个)。另一方面，长度为  $m$  的随机模式串多达  $2^m$  个，故匹配成功的概率为  $n / 2^m$ 。以  $n = 100,000$ 、 $m = 100$  为例，这一概率仅有

$$100,000 / 2^{100} < 10^{-25}$$

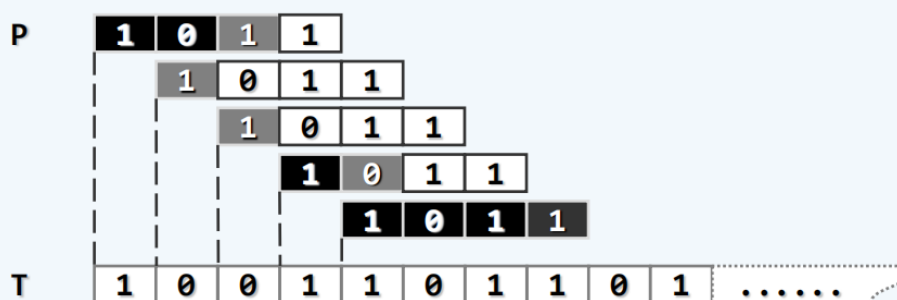
### 2. 随机产生文本串，并分别测试成功和失败的情况

- 成功情况：在文本串中随机取出长度为  $m$  的子串作为模式串，统计平均复杂度；
- 失败情况：随机产生模式串，统计平均复杂度。

## 蛮力匹配

### 算法原理

❖ 自左向右，以字符为单位，依次移动模式串  
直到在某个位置，发现匹配



### 实现版本一

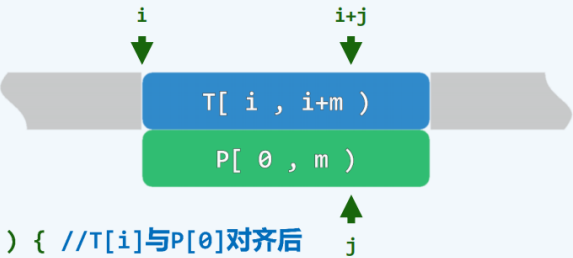
#### 版本1

```
❖ int match( char * P, char * T ) {  
    size_t n = strlen(T), i = 0;  
    size_t m = strlen(P), j = 0;  
    while ( j < m && i < n ) //自左向右逐次比对  
    {  
        if ( T[i] == P[j] ) { i++; j++; } //若匹配，则转到下一对字符  
        else { i -= j-1; j = 0; } //否则，T回退、P复位  
    }  
    return i-j; //如何通过返回值，判断匹配结果？  
}
```

### 实现版二

## 版本2

```
❖ int match( char * P, char * T ) {  
    size_t n = strlen(T), i = 0;  
    size_t m = strlen(P), j;  
    for ( i = 0; i < n - m + 1; i ++ ) { //T[i]与P[0]对齐后  
        for ( j = 0; j < m; j ++ ) //逐次比对  
            if ( T[i+j] != P[j] ) break; //失配，转下一对齐位置  
        if ( m <= j ) break; //完全匹配  
    }  
    return i; //如何通过返回值，判断匹配结果？  
}
```



Data Structures & Algorithms, Tsinghua University

4

## 性能分析

### 蛮力：复杂度

❖ 最好情况（只经过一轮比对，即可确定匹配）：#比对 =  $m = O(m)$

❖ 最坏情况（每轮都比对至P的末字符，且反复如此）

每轮循环：#比对 =  $m - 1(\text{成功}) + 1(\text{失败}) = m$

循环次数 =  $n - m + 1$

一般地有  $m \ll n$

故总体地，#比对 =  $m \times (n - m + 1) = O(n \times m)$

如果字符表足够大，那么每次  
比对可以只需常数时间，使得最  
坏情况下的复杂度降低至 $O(n)$

❖ 最坏情况，真会出现？

是的！



❖  $|Σ|$  越小，最坏情况出现的概率越高

$m$  越大，最坏情况的后果更严重

Data Structures & Algorithms (Fall 2013), Tsinghua University

8

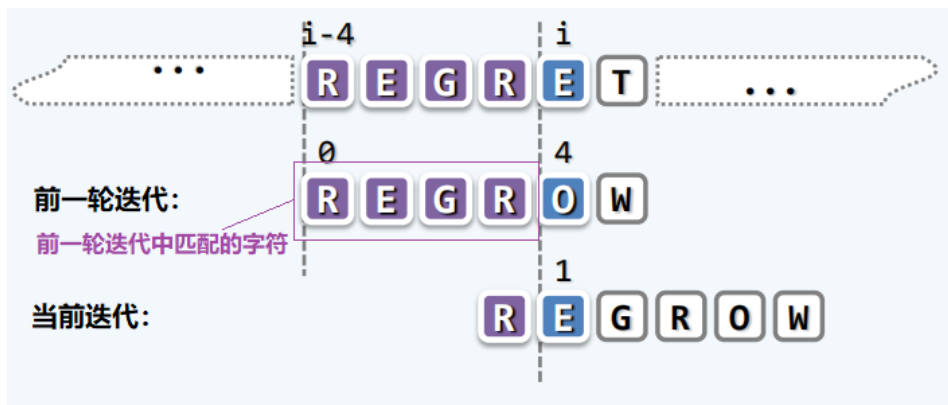
## KMP算法

### 算法原理

在蛮力实现的版本1中，每轮比对都要使文本串的指针回退到下一个对齐位置，而模式串的指针回退到首字符。总体看来，就像是模式串在文本串上一步一步的前进，每前进一步都要重头开始匹配。KMP算法就是蛮力实现的一种改进，它使得模式串可以每次前进多步，而且每前进一次都无需重头开始匹配。实质上讲，就是每轮比对的文本串指针不必回退，而模式串的指针也无须回退到首字符。

阮一峰讲解KMP算法：[http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Pratt\\_algorithm.html](http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Pratt_algorithm.html)

每轮迭代的对齐位置应从文本串和模式串的某个前缀匹配的子串开始，因为凡是和模式串前缀不匹配的所有文本子串是不可能和模式串匹配的。而这部分匹配的字符也不应重新匹配，因此有必要直接略过这些已匹配的字符而从未匹配的字符开始匹配，如下图所示：



由上图可以发现，在这两者的共同作用下，仅仅需要移动模式串的指针，而不需要移动文本串的指针。所以问题转化为模式串的指针需要回退到什么位置？实际上，这一位置恰恰可以由前一轮迭代的已匹配字符中获取。我们看一下“部分匹配值”这个概念：

“部分匹配值”就是“前缀”和“后缀”的最长的共有元素的长度。以“ABCDABD”为例，

- “A”的前缀和后缀都为空集，共有元素的长度为0；
- “AB”的前缀为[A]，后缀为[B]，共有元素的长度为0；
- “ABC”的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度为0；
- “ABCD”的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；
- “ABCD A”的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为“A”，长度为1；
- “ABCDAB”的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为“AB”，长度为2；
- “ABCDABD”的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

**字符串的前缀和后缀存在共有元素也称为自匹配。**

可以发现，模式串指针的回退位置恰恰是前一轮迭代中已匹配字符的部分匹配值。由于前一轮迭代中已匹配字符恰恰就是模式串的某个真前缀，因此我们可以针对模式串的每一个真前缀计算出其部分匹配值并制成一个表（称为next表），然后在每次迭代中根据前一轮迭代的已匹配字符属于哪一个模式串的真前缀，从表中相应地取出部分匹配值作为模式串指针的回退位置。

## 算法实现

由以上分析，可以将KMP算法实现如下：

## KMP算法

```
❖ int match( char * P, char * T ) {
    int * next = buildNext(P); //构造next表 即上述模式串各个真前缀的部分匹配值做成的表
    int n = (int) strlen(T), i = 0; //主串指针
    int m = (int) strlen(P), j = 0; //模式串指针
    while ( j < m && i < n ) //自左向右，逐个比对字符
        if ( 0 > j || T[i] == P[j] ) { //若匹配
            i++; j++; //则携手共进
        } else //否则，P右移，T不回退
            j = next[j];
    return i - j;
}
```

匹配成功则两个指针一起前进

匹配不成功仅需回退模式串的指针

delete [] next; //释放next表

return i - j;

$T[i] \neq P[j]$

$P[0, n(j)]$  ?  $P(n(j), m)$

$P[0, j]$  Y  $P(j, m)$

$T[0, i]$  X  $T(i, n)$

Data Structures & Algorithms (Fall 2013), Tsinghua University

## 实现next表

1. next表实际是一个数组，其下标对应每个模式串真前缀的长度，因此每个下标与真前缀一一对应。

2. 下标为0说明真前缀长度为0，即前一轮迭代中无任何匹配的字符，此时其部分匹配值规定为-1。这时在KMP算法中对应的情况就是 $j < 0$ ，相当于匹配成功，使得双指针共同前进一步，因此可以认为next表下标为0的真前缀是一个通配符哨兵。

3. 长度为j的真前缀的部分匹配值必为以下其中一个：

(通过向下递推验证，取第一个条件成立的为部分匹配值。如果没有一个条件成立，则只能取为0)

长度为j-1的真前缀的部分匹配值+1，仅当  $P[j-1]=P[n(j-1)]$  ；

长度为  $n[j-1]$  的真前缀的部分匹配值+1，仅当  $P[j-1]=P[n[n(j-1)]]$  ；

长度为  $n[n[j-1]]$  的真前缀的部分匹配值+1，仅当  $P[j]=P[n[n[n(j-1)]]]$  ；

..... ；

长度为0的真前缀的部分匹配值+1 = 0。

4. 详细说明上述过程：

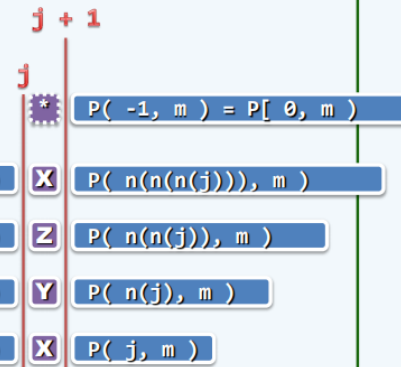
假设求长度为j的真前缀的部分匹配值，我们先看该前缀除了最后一个字符 $P[j-1]$ 外的前缀的最长自匹配，如果该自匹配最后一个字符和 $P[j-1]$ 相等，那么我们可以断定该真前缀的部分匹配值就是该自匹配长度加1；如果不相等，那么我们继续从该自匹配中找到其最长自匹配，其必定也是该真前缀除了最后一个字符 $P[j-1]$ 外的一个自匹配，那么我们再考察该自匹配最后一个字符和 $P[j-1]$ 是否相等，如果相等，那么我们可以断定该真前缀的部分匹配值就是该自匹配长度加1；如果不相等，以此类推。

## next[]的构造：算法

```

❖ int * buildNext( char * P ) { //构造模式串P的next[]表
    size_t m = strlen(P), j = 0; //“主”串指针
    int * N = new int[m]; //next[]表
    int t = N[0] = -1; //模式串指针 ( P[-1]通配符 )
    while ( j < m - 1 )
        if ( 0 > t || P[j] == P[t] ) //匹配
            N[ ++j ] = ++t;
        else //失配
            t = N[t];
    return N;
}

```



## 性能分析

### 分摊分析！

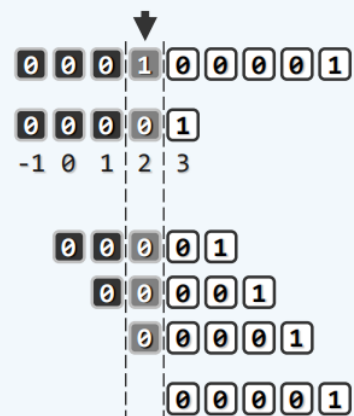
- ❖ 考察整个匹配过程，while循环累计执行 $O(n)$ 次
- ❖ 令  $k = 2*i - j$ ，观察k在算法过程中的变化趋势
  - 初始时， $k = 0$
  - 每经过一次循环，k至少增加1
  - 若if判断为true，则i、j同时加1，故k加1
  - 否则，i不变，j至少减1，故k也至少加1
- ❖ 同理，建立next[]也只需 $O(m)$ 时间
- ❖ 空间： $O(n + m)$

## 问题和改进

❖ 例 T = 0 0 0 1 0 0 0 0 1

P = 0 0 0 0 1

- ❖ T[3]：与 P[3] 比对，失败
  - 与 P[2] = P[next[3]] 继续比对，失败
  - 与 P[1] = P[next[2]] 继续比对，失败
  - 与 P[0] = P[next[1]] 继续比对，失败
- ❖ 最终，才前进到T[4]

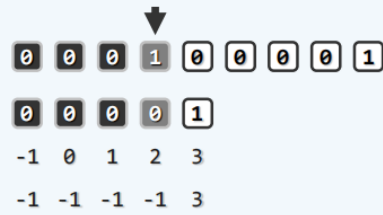


以上可以发现，即使已经知道后续比对的值还是0，但它依然还是继续比对。为了杜绝这个现象，就可以对next进行改进，让其可以在第一次比对不匹配时，就略过后续相同的值：



## 再改进：算法

```
❖ int * buildNext( char * P ) {
    size_t m = strlen(P), j = 0; //“主” 串指针
    int * N = new int[m]; //next表
    int t = N[0] = -1; //模式串指针
    while ( j < m - 1 )
        if ( 0 > t || P[j] == P[t] ) { //匹配
            j++; t++; N[j] = P[j] != P[t] ? t : N[t];
        } else //失配
            t = N[t];
    return N;
}
```



## BM算法

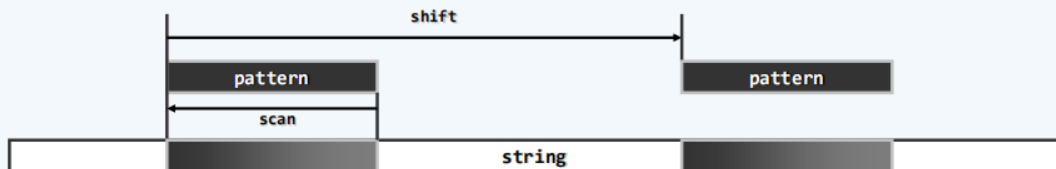
BM算法和KMP算法得过程基本相似：每次迭代比对文本串和模式串，当出现失配字符时就进行查表，找到并将模式串移动到下一次迭代的对齐位置。与KMP算法不同的是，BM算法是自右向左进行比对，且每次需要查两张表（BC表和GS表）。

❖ 预处理：根据模式串P，预先构造gs[]表和bc[]表

迭代：自右向左依次比对字符，找到极大的匹配后缀

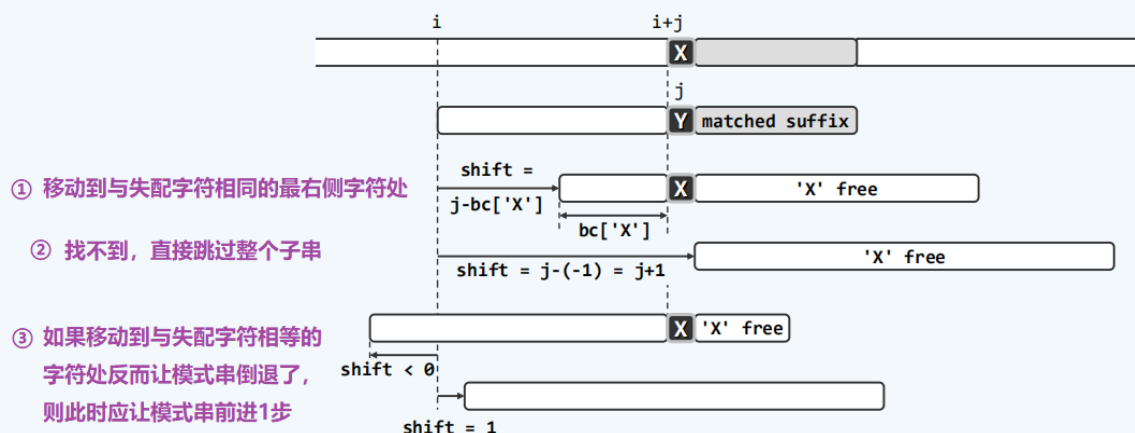
若完全匹配，则返回位置

否则，根据gs[]和bc[]，P适当右移，并重新自右向左比对



## BC表（坏字符规则）

BC表记录了模式串每个字符在串中的最大秩，它认为如果失配字符（坏字符）想要有再次匹配的可能，就必须在模式串中找到与之相同的字符，如果存在则将其与之对齐并重新进行比对，如果不存在则可以直接跳过当前比对的整个文本子串。具体细分下来有以下三种情况：





BC表的构建算法实现：

### BC[]表的构造：算法

```
❖ int * buildBC( char * P ) {  
    int * bc = new int[ 256 ]; //bc[]表，与字母表等长  
    for ( size_t j = 0; j < 256; j++ ) bc[j] = -1; //初始化  
    for ( size_t m = strlen(P), j = 0; j < m; j++ ) //画家算法，自左向右扫描  
        bc[ P[ j ] ] = j; //刷新P[j]的出现位置记录  
    return bc;  
} //第二个循环，通过引入临时变量m，避免反复调用strlen()  
  
❖ 与KMP同理，凡未出现之字符，BC值均取作-1 //相当于在P的左侧添加通配符  
  
❖ 附加空间 = | bc[] | =  $O(|\Sigma|)$  =  $O(|S|)$   
❖ 时间 =  $O(|\Sigma| + m)$  =  $O(|S| + m)$  //可改进至 $O(m)$ 
```

Data Structures & Algorithms (Fall 2013), Tsinghua University

8

举例说明：



如果仅用BC表实现BM算法，其复杂度如下所示：

### BC策略：最好情况

```
❖  $O(n / m)$  —— 除法？没错！  
  
❖ 比如：T = 

|   |   |   |   |   |
|---|---|---|---|---|
| x | x | x | x | 1 |
|---|---|---|---|---|



|   |   |   |   |   |
|---|---|---|---|---|
| x | x | x | x | 1 |
|---|---|---|---|---|



|   |   |   |   |   |
|---|---|---|---|---|
| x | x | x | x | 1 |
|---|---|---|---|---|

 . . . 

|   |   |   |   |   |
|---|---|---|---|---|
| x | x | x | x | 1 |
|---|---|---|---|---|



|   |   |   |   |   |
|---|---|---|---|---|
| x | x | x | x | 1 |
|---|---|---|---|---|

  
P = 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|

  
  
❖ 只要T的当前字符未 在P中出现，即可直接移动m个字符  
此时，仅需单次比较，即可排除m个对齐位置  
  
❖ 单次匹配概率越小 ( $|\Sigma|$  越大) 的场合 //ASCII + UniCode  
相对于蛮力算法的性能优势越明显  
  
❖ P越长，这类移动的效果越明显
```

Data Structures & Algorithms (Fall 2013), Tsinghua University

9

## BC策略：最差情况

❖ 最坏 =  $O(n \times m)$  —— 等效于蛮力算法？是的！

❖ 比如：T = [0][0][0][0][0] . . . [0][0][0]

P = [1][0][0][0][0]

❖ 每轮迭代，都要在扫过整个P之后，方能确定右移一个字符

此时，须经m次比较，方能排除单个对齐位置

❖ 单次匹配概率越大（ $|\Sigma|$ 越小）的场合，性能退化至接近于蛮力算法 //Bitmap + DNA

❖ 反思：借助以上bc表，仅仅利用了失配比对提供的信息（教训）！

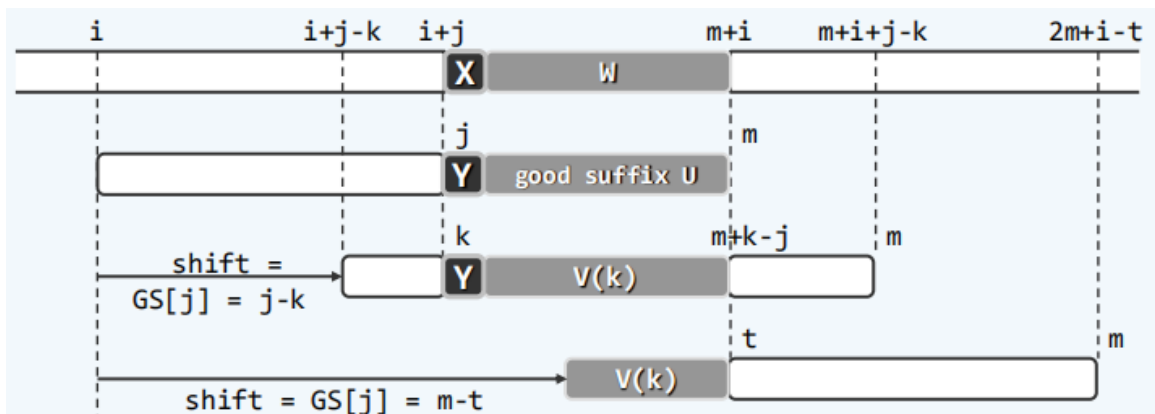
类比：可否仿照KMP，同时利用起匹配比对提供的信息（经验）？

Data Structures & Algorithms (Fall 2013), Tsinghua University

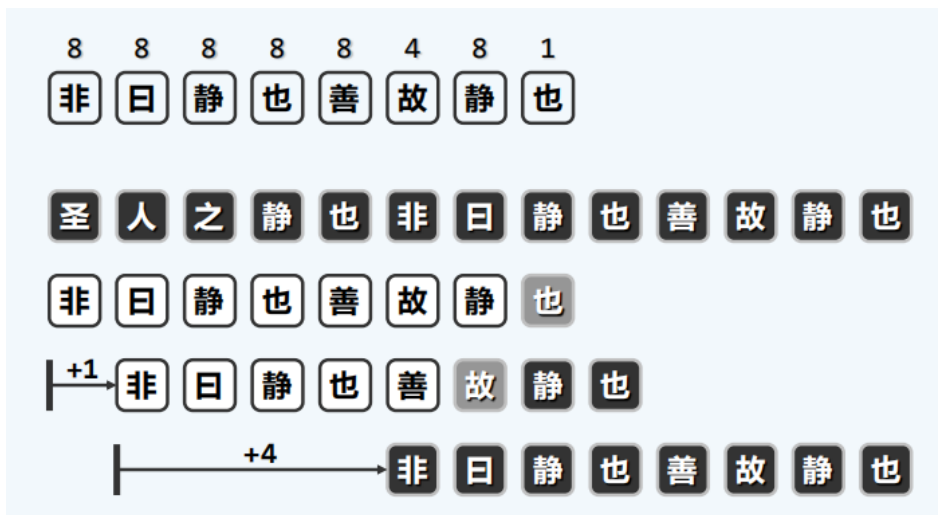
10

## GS表（好后缀规则）

为了避免BC表的最差情况（使模式串反向移动的情况），就创建了GS表。将一次迭代中比对到失配字符之前所匹配成功的模式串后缀称为好后缀，GS表认为好后缀具有完全匹配成功的潜能，只要能在模式串中找到另一个与之匹配的子串或局部匹配的前缀。如果能够找到，就应该将其移动到好后缀的位置上重新与文本串中与好后缀对应的子串进行比对；如果没有找到，则说明模式串不可能与其完全匹配，应该直接略过该部分字符。如下所示：



举例说明：



GS表的构建：

## 1. 构建MS字符串和SS表

**MS[] → ss[]**

❖ 令： $|MS[j]| = \max_{0 \leq s \leq j+1} \{P(j-s, j) = P[m-s, m]\}, 0 \leq j < m$   
 即  $P[0, j]$  所有后缀中，与  $P$  的某一后缀匹配的最长者

❖ 令： $ss[j] = |MS[j]| \leq j + 1, 0 \leq j < m$

❖ 实际上,  $ss[]$  表中蕴含了  $gs[]$  表的所有信息 // 无非两种情况...

Data Structures & Algorithms, Tsinghua University

## 2. 从SS表到GS表

**ss[] → gs[]**

a) 若  $ss[j] = j + 1$ , 则对于任何  $i < m - j - 1$ ,  $m - j - 1$  必是  $gs[i]$  的一个候选

b) 若  $ss[j] \leq j$ , 则  $m - j - 1$  必是  $gs[m - ss[j] - 1]$  的一个候选

Data Structures & Algorithms, Tsinghua University

## 使用哪一个

BC表和GS表可以同时使用，然后取使模式串向前移动最大者。

总结起来就是，在失配时从模式串中分别找到坏字符或好后缀，找得到就对齐移动距离最大者，找不到就跳过。

## 性能分析

## BC + GS : 性能分析

### ❖ 空间

$$|BC[]| + |GS[]|$$

$$= O(|\Sigma| + m)$$

### ❖ 预处理

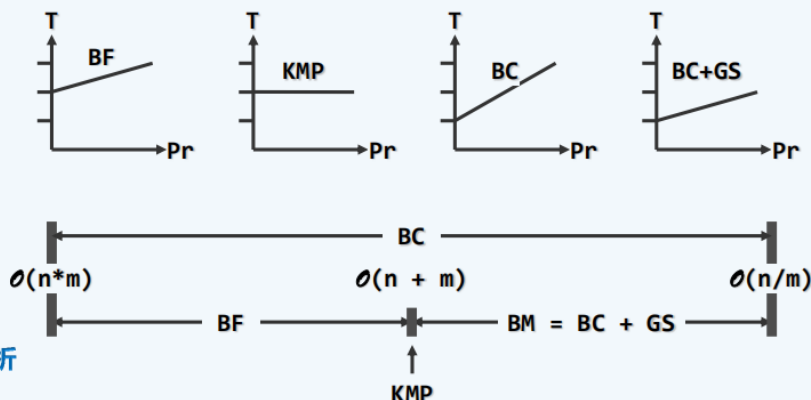
$$O(|\Sigma| + m)$$

### ❖ 查找

$$\text{最好} = O(n / m)$$

$$\text{最差} = O(n + m)$$

// 参照KMP算法的分析



## Karp-Rabin算法

利用散列函数将模式串和文本串转化为数值，然后逐一进行比对：

❖ P = 8 2 8 1 8 //hash( P ) = hash( 82818 ) = 77

T = 2 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5 2 3 5 3 6

2 7 1 8 2 //22

7 1 8 2 8 //48

1 8 2 8 1 //45

8 2 8 1 8 //77

散列冲突的解决：经过数值比对后，如果相等则还需进行各字符的逐一比对。

P = 1 8 2 8 4 //仍取M = 97, 则hash(18284) = 48

T = 2 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5 2 3 5 3 6

2 7 1 8 2 //22

7 1 8 2 8 //48

...

1 8 2 8 4 //48