BM模式匹配算法

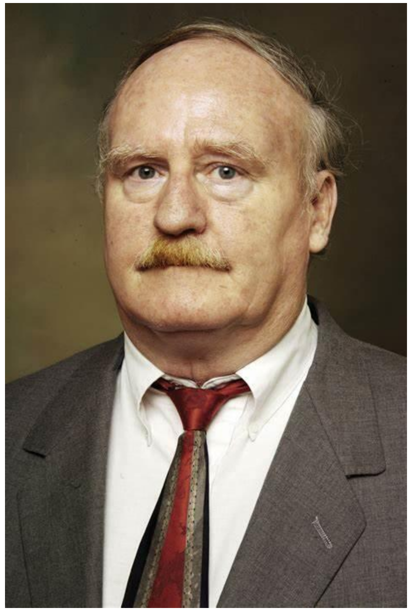
- 基本思想
- 坏字符策略及实现
- 好后缀策略及实现

Last updated on 2024.11

zhuyungang@jlu.edu.cn

BM算法

BM算法：以终为始，方得始终

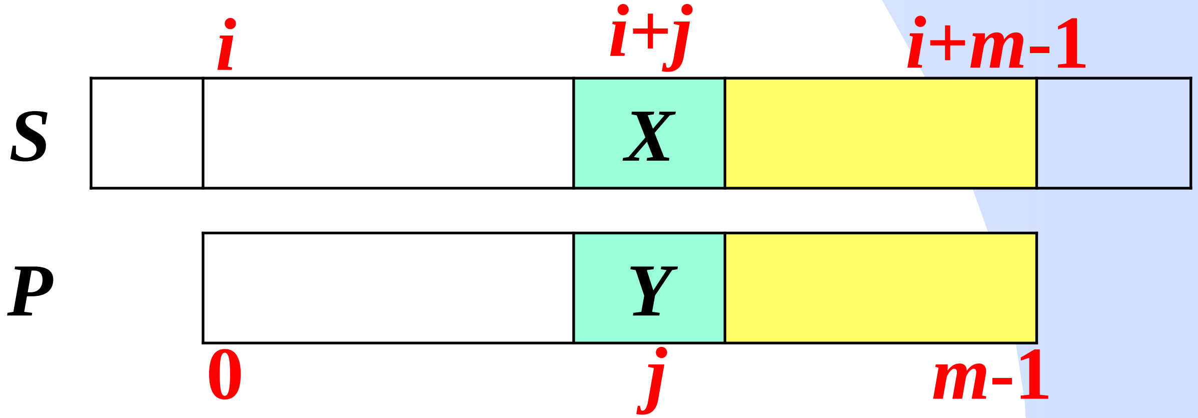


Robert S. Boyer
德克萨斯大学奥斯汀分校

J Strother Moore
德克萨斯大学奥斯汀分校
美国工程院院士

朴素模式匹配的另一写法

```
int BF(char *s, char *p, int n, int m){
    int i=0; //用指针i扫描主串s
    while (i <= n - m){ //看p是否在s的第i个位置
        int j=0; //用指针j扫描模式串p
        while (j < m && p[j] == s[i+j])
            j++;
        if (j == m)  return i;     //匹配成功
        i++;    //p不在s的第i个位置，继续看p在不在s的第i+1个位置
    }
    return -1;
}
```



BM算法

每一趟比对，按照模式串下标从大到小（自右向左）的顺序进行比对

坏字符

S	*b*	*a*	*b*	*a*	*x*	*a*

P	*b*	*a*	*a*
	③	②	①

← 匹配顺序

BM算法——坏字符策略

➢ 若*P*中不含坏字符：

坏字符

S	*A*	*B*	*A*	*C*	*T*	*L*	*E*	*E*	*D*	*L*	*E*
P		*L*	*E*	*E*	*D*	*L*	*E*				

该坏字符与模式串*P*中的任何字符都不可能匹配，则*P*整体移过失配位置，即移动到坏字符后面的位置

S	*A*	*B*	*A*	*C*	*T*	*L*	*E*	*E*	*D*	*L*	*E*
P						*L*	*E*	*E*	*D*	*L*	*E*

BM算法——坏字符策略

➢ 若P中含有1个坏字符：

S	A	B	A	C	N	L	E	D	L	E	E
P		N	L	E	D	L	E				

还能将P移过坏字符么？至少坏字符本身应得以匹配。

找出P中的坏字符，让其与主串中失配位置对齐。

S	A	B	A	C	N	L	E	D	L	E	E
P					N	L	E	D	L	E	

BM算法——坏字符策略

➢ 若P中包含多个坏字符：

S	A	B	C	A	A	B	C	B	C	E	E
P		A	A	B	C	B	C				

让P中**最右边**的坏字符与主串中失配位置对齐。

S	A	B	C	A	A	B	C	B	D	E	E
P				A	A	B	C	B	D		

BM算法——坏字符策略

➢ *P*中包含多个坏字符，最右边的坏字符过于靠右，在失配位置的右侧：

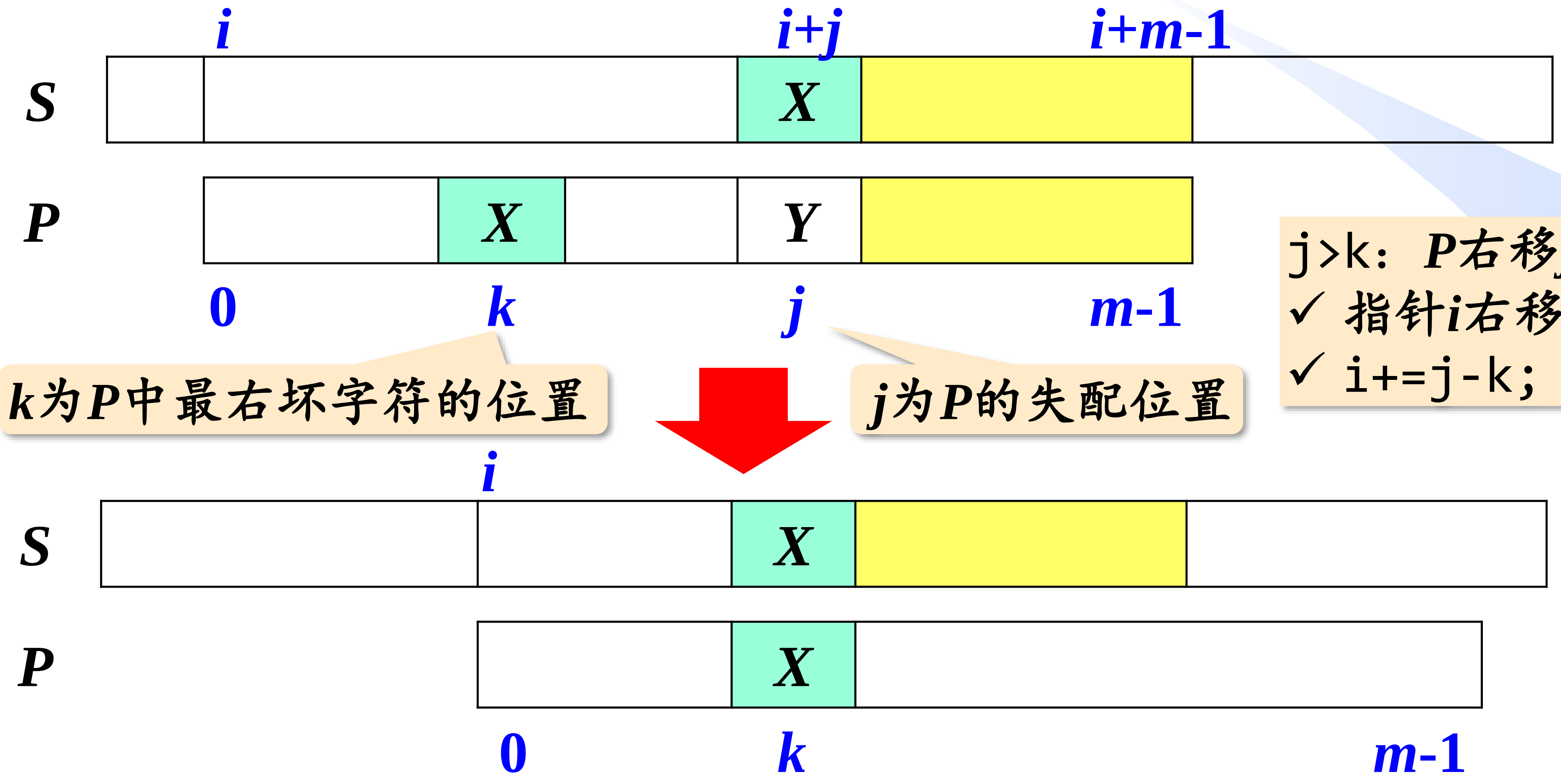
S	A	B	N	E	E	L	E	D	L	E	E
P		N	E	E	L	L	E				

此时将*P*右移1个字符。

S	A	B	N	E	E	L	E	D	L	E	E
P			N	E	E	L	E	D			

BM算法——坏字符策略

①**左端出现规则**：*P*包含坏字符，且在失配位置的左端，让*P*中最右边的坏字符（失配位置左侧且与失配位置最近的坏字符）与失配位置对齐。

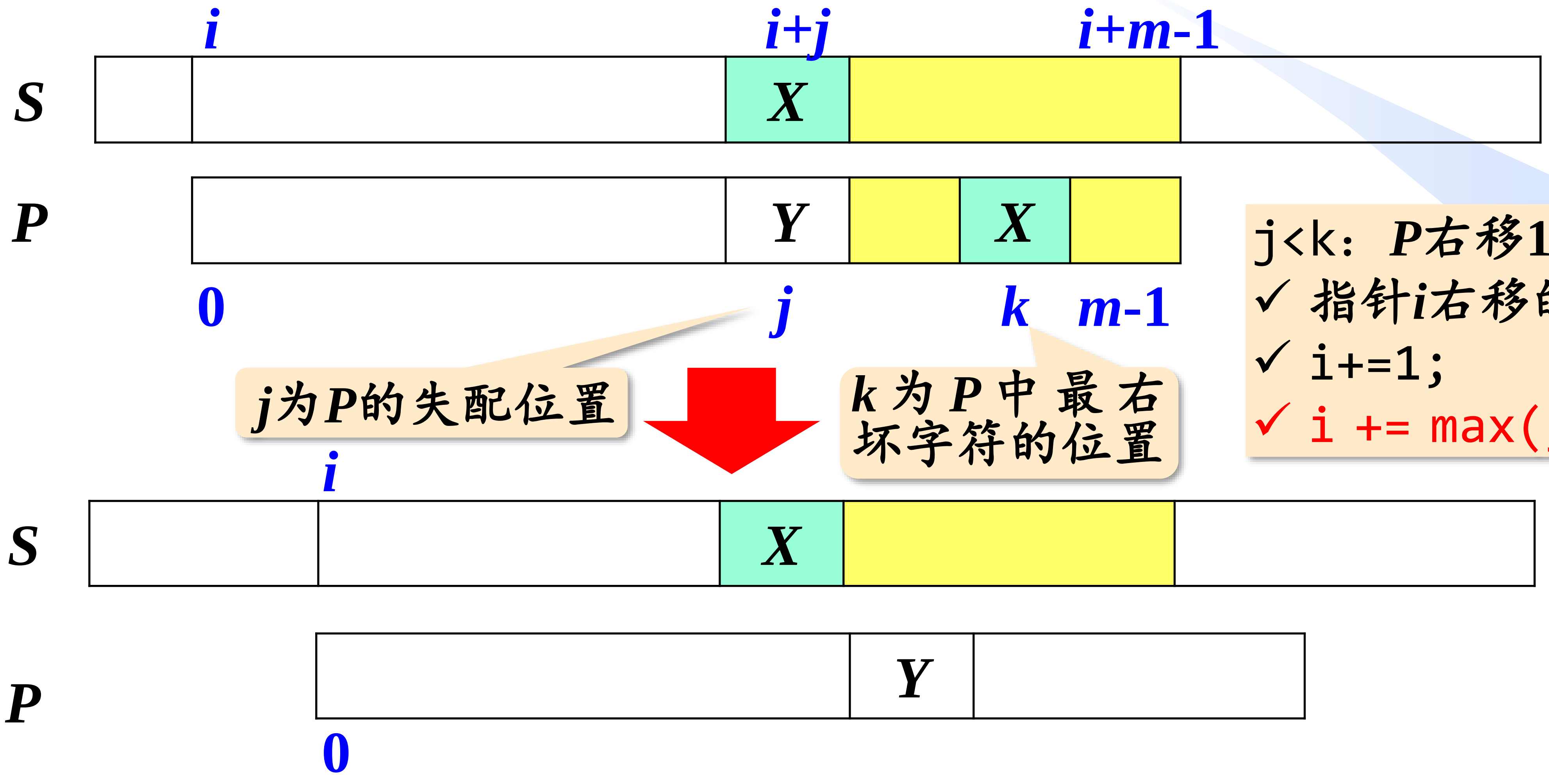


j>k：*P*右移*j-k*位

- ✓ 指针*i*右移的位数和*P*相同
- ✓ `i+=j-k;`

BM算法——坏字符策略

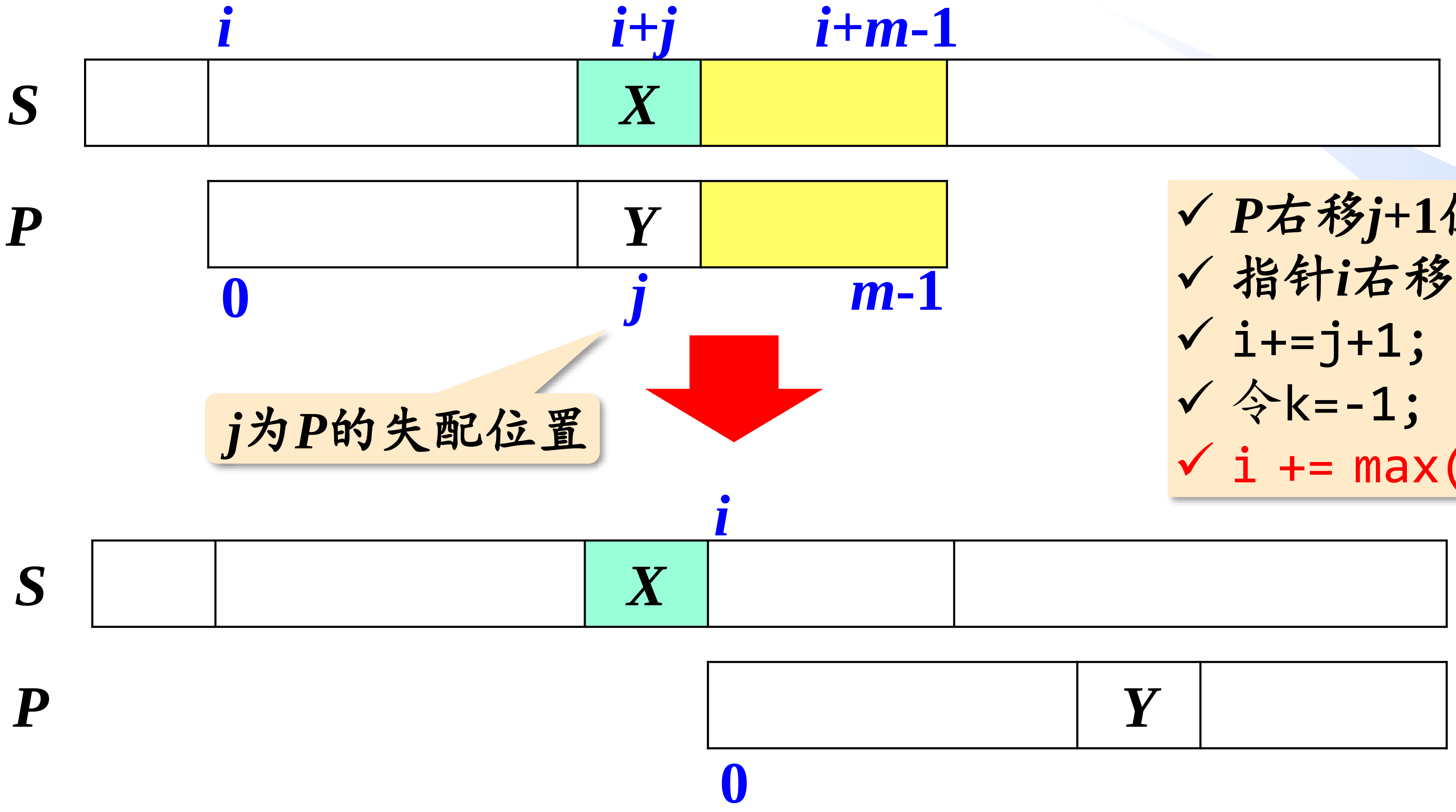
②**右端出现规则**：*P*包含坏字符，但最右边的坏字符在失配位置的右端，则*P*右移1位。



j<k：*P*右移1位
- ✓ 指针*i*右移的位数和*P*相同
- ✓ i+=1;
- ✓ i += max(j - k, 1)

BM算法——坏字符策略

③未出现规则：模式串*P*不包含坏字符，则*P*整体移过失配位置。



- ✓ *P*右移*j*+1位
- ✓ 指针*i*右移的位数和*P*相同
- ✓ i+=j+1;
- ✓ 令k=-1;
- ✓ i += max(j - k, 1)

坏字符策略实现——BC（Bad Character）表

BC表：存储每个字符在*P*中出现的最右位置，BC[*ch*]就是字符*ch*在*P*的最右位置。

P

a	*c*	*a*	*a*	*f*
0	1	2	3	4

BC表

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	*u*	*v*	*w*	*x*	*y*	*z*

```
const int charsize = 256;
void buildBC(char *p, int m, int BC[]){
    for(int i=0; i<charsize; i++)
        BC[i] = -1;
    for(int i=0; i<m; i++){
        char ch = p[i];
        BC[ch] = i; // 字符ch在P的最右位置是i
    }
}
```

坏字符策略实现——BC（Bad Character）表

BC表：存储每个字符在*P*中出现的最右位置，BC[*ch*]就是字符*i*在*P*的最右位置。

P

a	*c*	*a*	*a*	*f*
0	1	2	3	4

***BC*表**

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	*u*	*v*	*w*	*x*	*y*	*z*

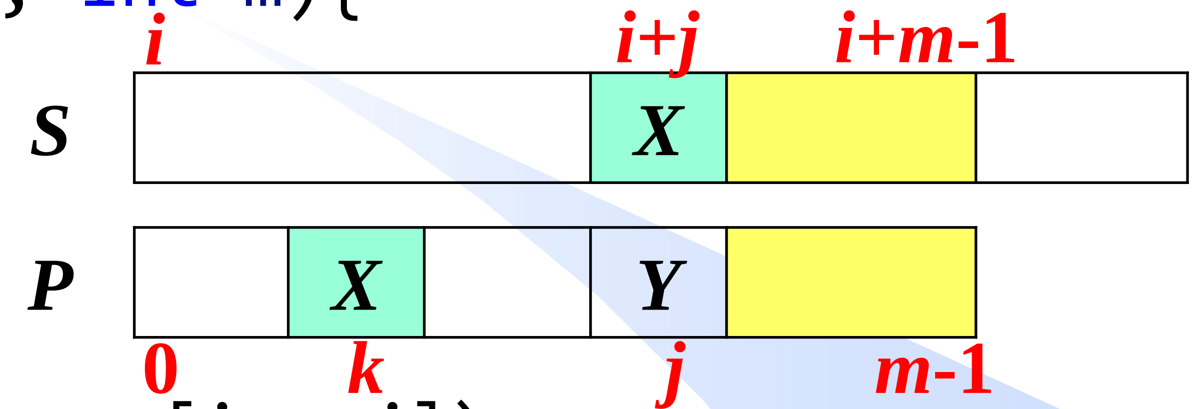
```
const int charsize = 256;
void buildBC(char *p, int m, int BC[]){
    for(int i=0; i<charsize; i++)
        BC[i] = -1;
    for(int i=0; i<m; i++)
        BC[p[i]] = i; //字符p[i]在P的最右位置是i
}
```

时间复杂度 O(*m*)

基于坏字符策略的BM算法

```
const int charsize = 256;
int BM(char *s, char *p, int n, int m){
    int BC[charsize], i=0;
    buildBC(p, m, BC);
    while (i <= n - m){
        int j = m - 1;
        while (j >= 0 && p[j] == s[i + j])
            j--;
        if (j < 0)  return i;
        int k=BC[s[i+j]];//坏字符在p的最右位置
        i += max(j-k, 1); //在Pj处失配

    }
    return -1;
}
```



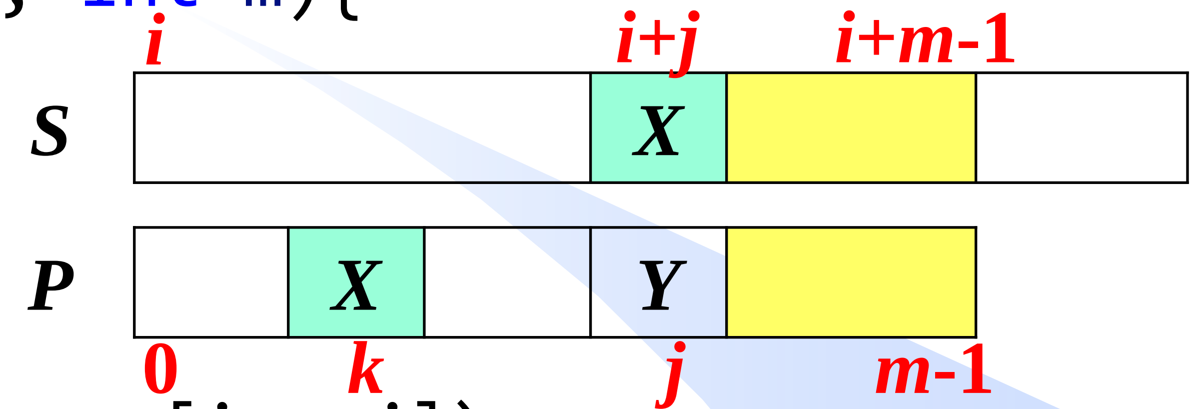
坏字符

BC[s[i+j]]

坏字符在P的最右位置

基于坏字符策略的BM算法

```
const int charsize = 256;
int BM(char *s, char *p, int n, int m){
    int BC[charsize], i=0;
    buildBC(p, m, BC);
    while (i <= n - m){
        int j = m - 1;
        while (j >= 0 && p[j] == s[i + j])
            j--;
        if (j < 0)  return i;
        i += max(j-BC[s[i+j]], 1); //在Pj处失配
    }
    return -1;
}
```



$$\underbrace{\mathrm{BC}[\overbrace{s[i+j]}^{\text{坏字符}}]}_{\text{坏字符在}P\text{的最右位置}}$$

基于坏字符策略的BM算法时间复杂度

- 最好情况O(n/m)【每比较1次右移m位】

S	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				

- 字符集越大（如ACSII、汉字），单词比对成功概率越小，因比对失败的概率增加，更可能大跨度地向右移动，性能优势越明显。

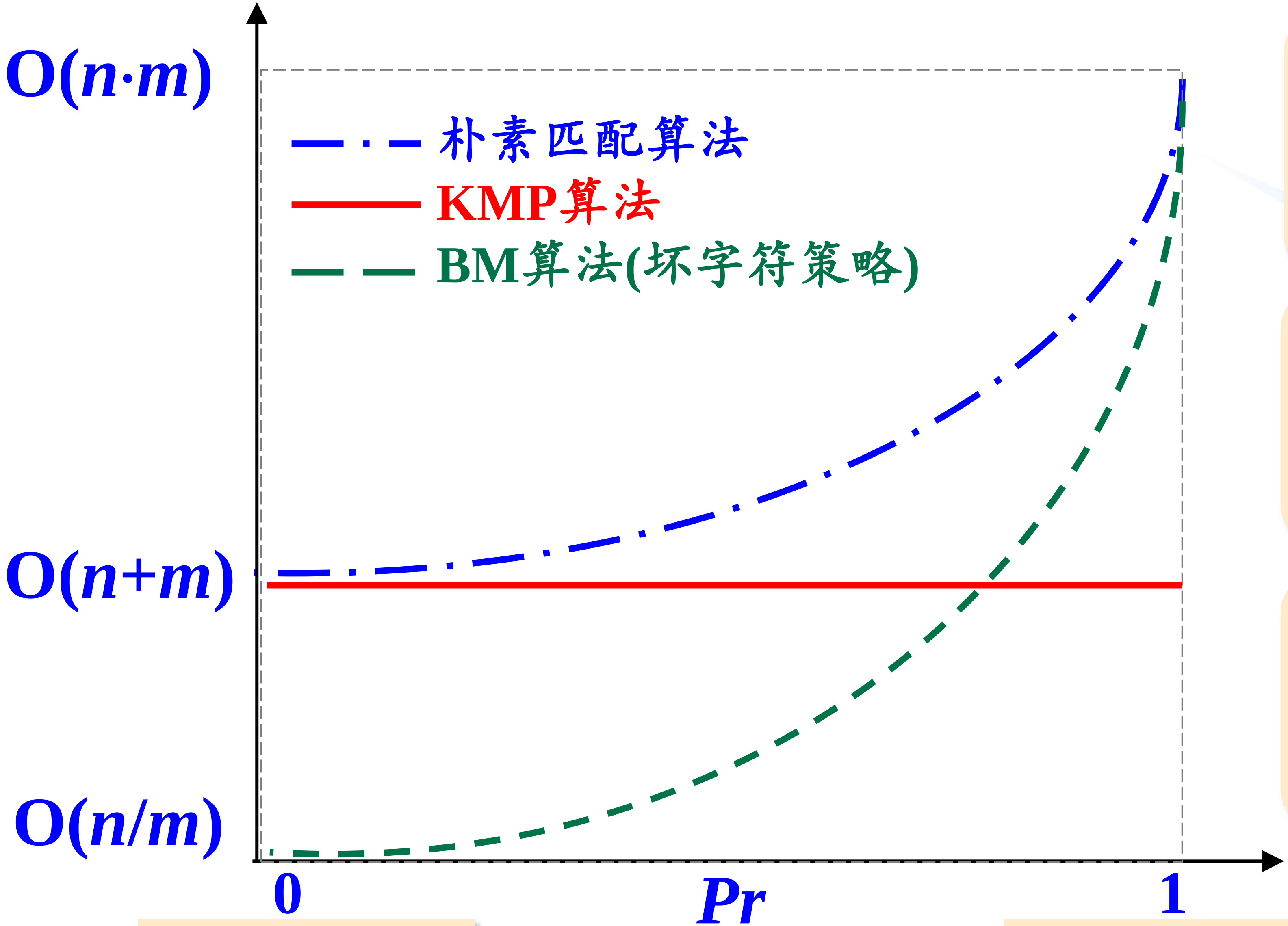
基于坏字符策略的BM算法时间复杂度

- 最坏情况O($n \cdot m$)【每比较m次右移1位】

S	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
P	1 0 0 0 0

- 退化成朴素匹配算法。
- 字符集越小（如ATCG基因序列），单词比对成功概率越高，难以大跨度的向右移动，性能越接近朴素算法。

基于坏字符策略的BM算法总结



字符集越大

字符集越小

字符单次成功比对概率$Pr = 1/k$（k为字符集大小）

字符集越大，字符单次成功比对概率越低，性能优势越明显

字符集越小，字符单次成功比对概率越高，越接近朴素算法

练习

设目标串*S*和模式串*P*的长度分别为2023和17，且*S*不包含*P*，则仅使用**基于坏字符策略的BM算法**进行模式匹配，最好情况下需进行________次字符比较。【清华大学2023年考研题】

判断：相对于KMP算法而言，BM算法更适合于大字符集的应用场合。【清华大学期末考试题】

BM模式匹配算法

- 基本思想
- 坏字符策略及实现
- **好后缀策略及实现**

zhuyungang@jlu.edu.cn

BM算法的好后缀

好后缀：尾部匹配的子串

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*	…
P				*F*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*			

BM算法的好后缀策略

好后缀：尾部匹配的子串

① 若*P*失配位置左方有与好后缀相等的子串：

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*X*	*Y*	*A*	*A*	*B*	*C*	*A*	*B*	…
P				*F*	*X*	*Y*	*A*	*A*	*B*	*C*	*A*	*B*				

在*P*失配位置左方选取与好后缀相等的子串，让该子串与主串的好后缀对齐。

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*X*	*Y*	*A*	*A*	*B*	*C*	*A*	*B*	…
P							*F*	*X*	*Y*	*A*	*A*	*B*	*C*	*A*	*B*	

BM算法的好后缀策略

好后缀：尾部匹配的子串

① 若*P*失配位置左方有与好后缀相等的子串：

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*	…
P				*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*				

漏掉成功匹配位置

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*	…							
P										*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*						

BM算法的好后缀策略

好后缀：尾部匹配的子串

① 若P失配位置左方有与好后缀相等的子串：

S	A	B	A	A	B	A	F	A	B	A	A	B	C	A	B	…
P				F	A	B	A	A	B	C	A	B				

在P失配位置左方选取与好后缀相等的、最靠右的子串，让该子串与主串的好后缀对齐。

S	A	B	A	A	B	A	F	A	B	A	A	B	C	A	B	…
P							F	A	B	A	A	B	C	A	B	

BM算法的好后缀策略

好后缀：尾部匹配的子串

① 若*P*失配位置左方有与好后缀相等的子串：

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*A*	*B*	*A*	*A*	*B*	*C*	…
P	*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*		

下趟比对必然失败

S	*A*	*B*	*A*	*A*	*B*	*A*	*F*	*A*	*B*	*A*	*A*	*B*	*C*	…		
P				*F*	*A*	*B*	*A*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	

BM算法的好后缀策略

好后缀：尾部匹配的子串

① 若P失配位置左方有与好后缀相等的子串：

S	A	B	A	A	B	A	F	A	B	A	A	B	C	…
P	F	A	B	A	A	B	C	A	B	C	A	B		

在P失配位置左方选取与好后缀相等的、最靠右的子串（且其左侧字符与P的失配字符不相等）,让该子串与主串的好后缀对齐。

S	A	B	A	A	B	A	F	A	B	A	A	B	C	…			
P							F	A	B	A	A	B	C	A	B	C	A	B

BM算法的好后缀策略

② 若P失配位置左方没有与好后缀相等的子串，则找P的最长前缀，该前缀与**好后缀的某个后缀**相等：

S	A	B	L	C	A	B	X	C	A	B
P	A	B	X	C	A	B				

好后缀

让该前缀与主串的好后缀对齐。

S	A	B	L	C	A	B	X	C	A	B
P				A	B	X	C	A	B	

BM算法的好后缀策略

③ 若*P*失配位置左方没有与好后缀相等的子串，且找不到与好后缀的某个后缀相等的前缀，即*P*的左侧没有与*S*的好后缀匹配的子串。

S	*A*	*B*	*E*	*C*	*A*	*B*	*D*	*E*	*F*	*C*	*A*	*B*
P	*D*	*E*	*F*	*C*	*A*	*B*						

好后缀

将*P*整体右移至*S*的好后缀后面。

S	*A*	*B*	*E*	*C*	*A*	*B*	*D*	*E*	*F*	*C*	*A*	*B*
P							*D*	*E*	*F*	*C*	*A*	*B*

GS（Good Suffix）表

- ***GS*(*j*): 在P_j处失配后，*P*移动的位数。**
- ***GS*表可以在O(*m*)的时间构建。**

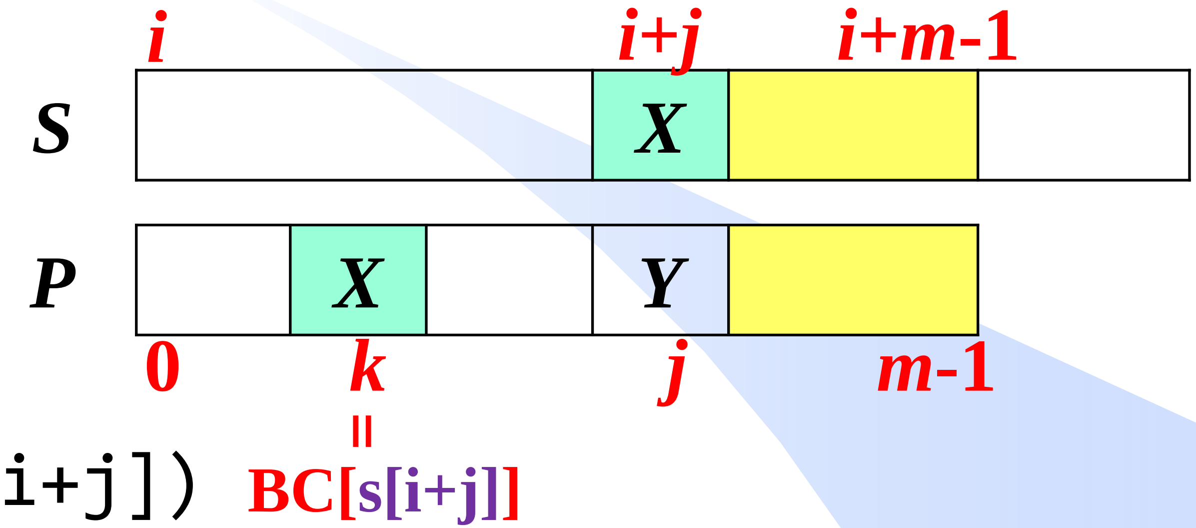
基于坏字符+好后缀策略的BM算法

每次失配后，

$$P\text{右移的位数} = \max\left\{\begin{matrix}\text{坏字符策略}\\\text{算出的右移位数}\end{matrix},\ \begin{matrix}\text{好后缀策略}\\\text{算出的右移位数}\end{matrix}\right\}$$

基于坏字符+好后缀策略的BM算法

```
const int charsize=256, maxm=1e4+10;
int BM(char *s, char *p, int n, int m){
   int BC[charsize], GS[maxm];
   buildBC(p, m, BC); //建BC表
   buildGS(p, m, GS); //建GS表
   int i = 0;
   while(i <= n - m){
      int j = m - 1;
      while(j >= 0 && p[j] == s[i+j])
         j--;
      if(j < 0) return i;
      i += max(j-BC[s[i+j]], GS[j]); //在Pj处失配，需右移的位数
   }
   return -1;
}
```



坏字符策略
算出的右移位数

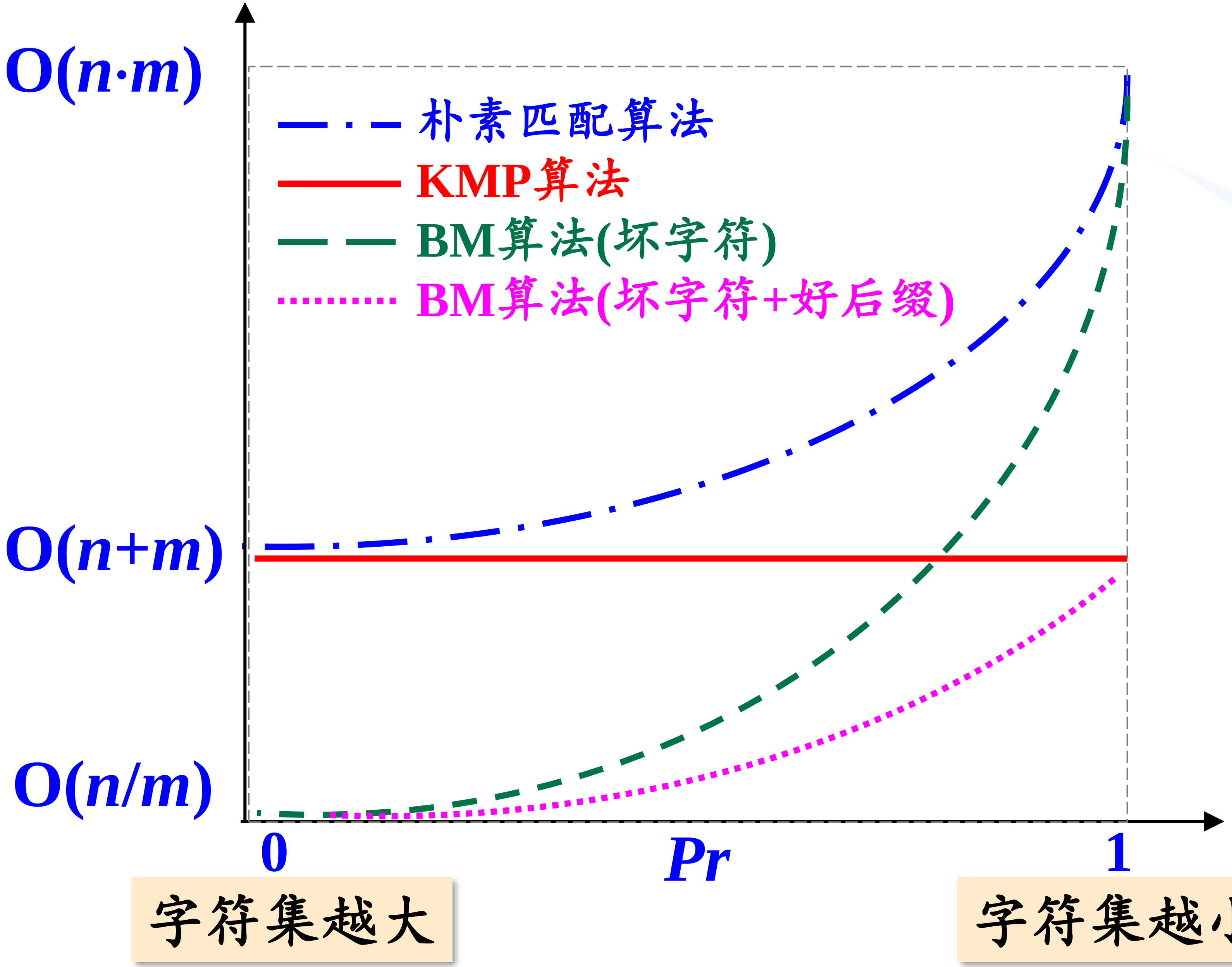
好后缀策略
算出的右移位数

时间复杂度

可以证明，结合坏字符和好后缀策略的BM算法，在最坏情况下时间复杂度为O(n+m)[1,2]。

1. Guibas L, Odlyzko A. A New Proof of the Linearity of the Boyer-Moore String Search Algorithm. ***SIAM Journal on Computing***. 1980,9(4):672-682.
2. Cole R. Tight Bounds on the Complexity of the Boyer-Moore Pattern Matching Algorithm. ***SIAM Journal on Computing***. 1994,23(5):1075-1091.

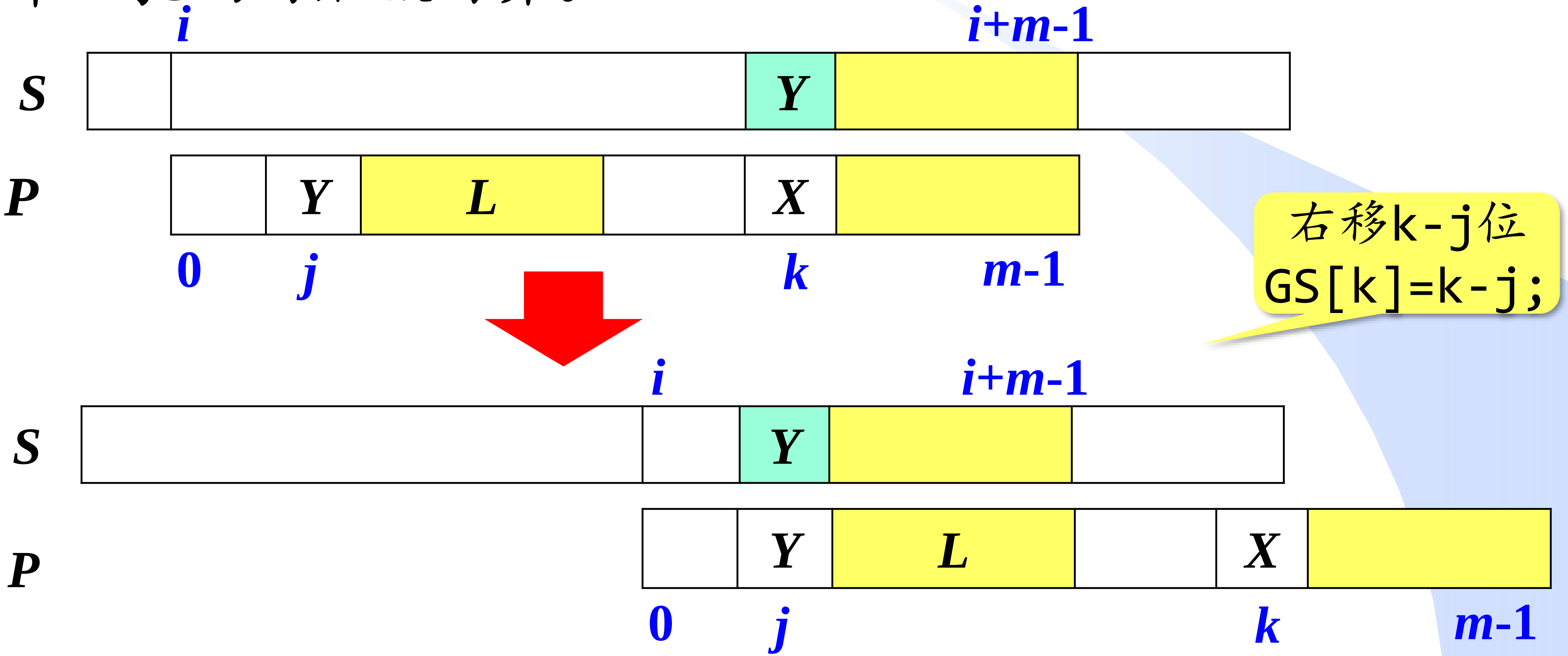
时间复杂度



字符单次成功比对概率$Pr = 1/k$（k为字符集大小）

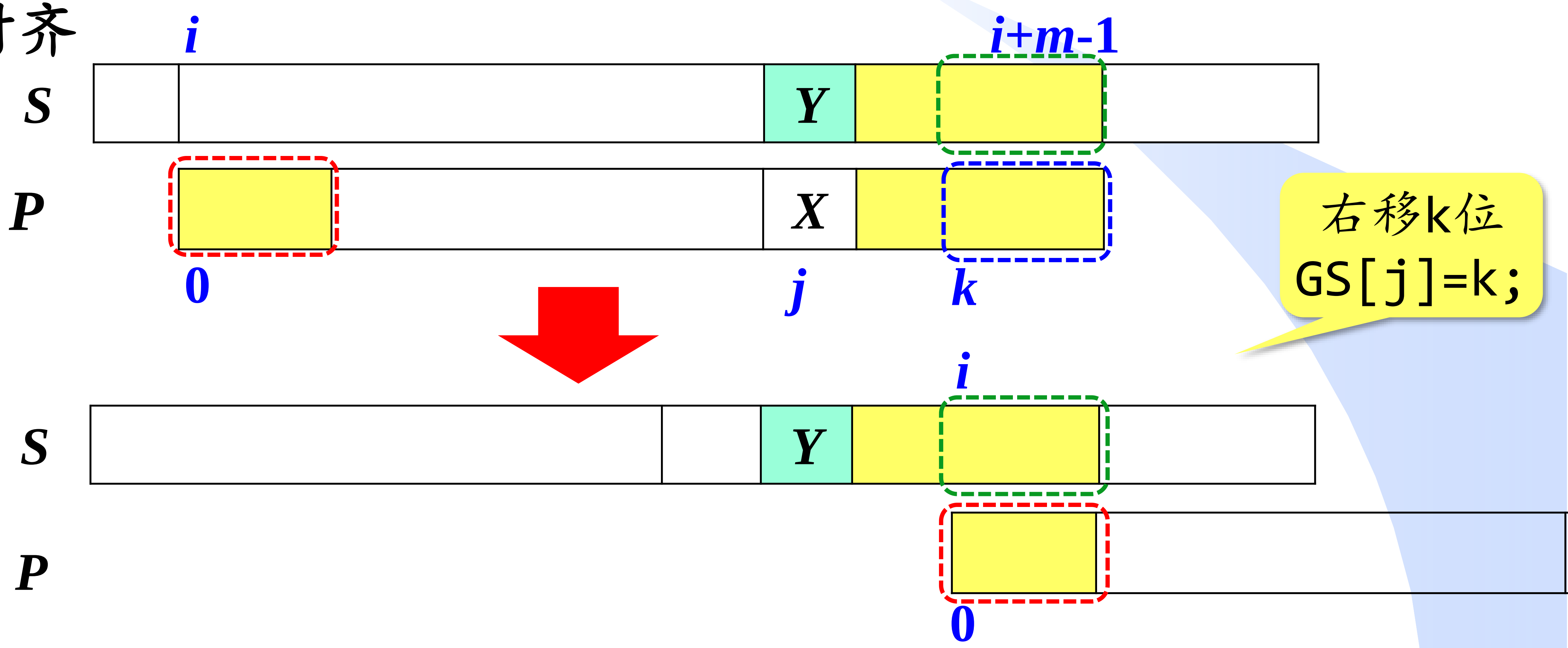
好后缀策略——实现

① P失配位置k左方有与好后缀相等且$P_j \neq P_k$的子串L，移动P使子串L与S的好后缀对齐。



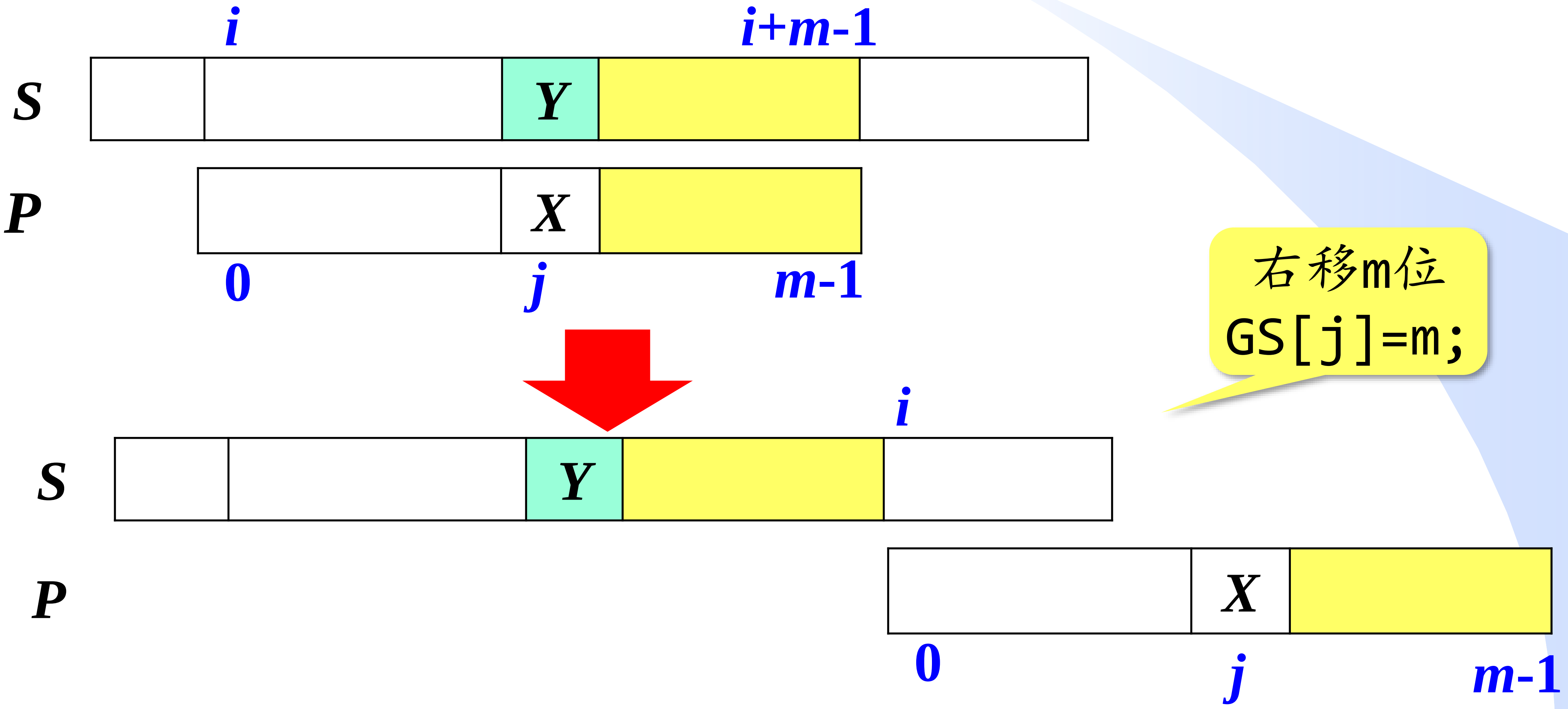
好后缀策略——实现

② 若P失配位置j左侧没有与好后缀相等的子串，但P的某个最长前缀与好后缀的某个后缀$P_k…P_{m-1}$相等，让该前缀与S的好后缀对齐



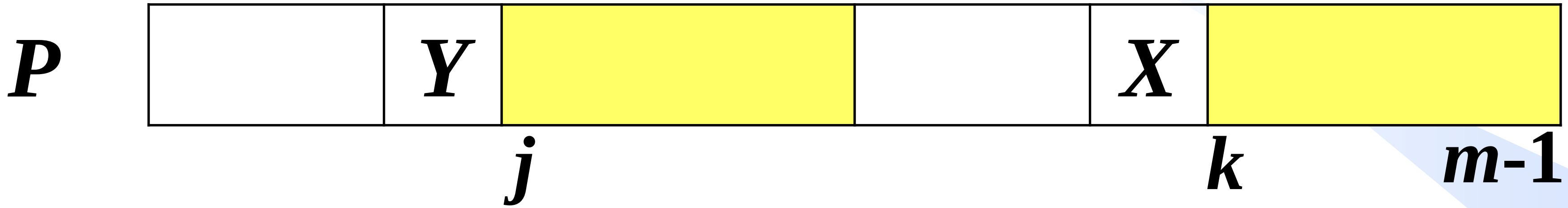
好后缀策略——实现

③ 若P失配位置j左侧没有与S的好后缀或其后缀匹配的子串，将P整体右移至S的好后缀后面。



GS表的构建

在P失配位置左方与好后缀相等的子串：

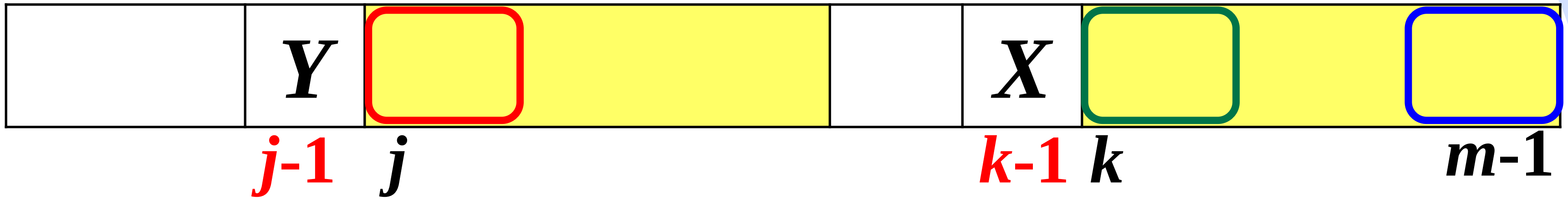


$$suffix(j)=\begin{cases} P_j \ldots P_{m-1}\text{的最长相等前后缀中，后缀的起点下标} \\ m, \quad P_j \ldots P_{m-1}\text{无相等前后缀} \\ m+1, \quad j=m \end{cases}$$

GS(i): 在P_i处失配后，P移动的位数

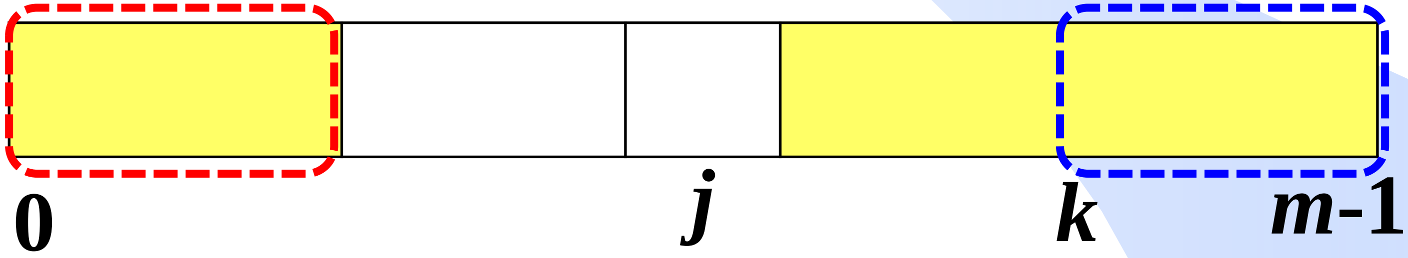
GS表的构建

```
void buildGS(char *p, int m, int GS[]) {
  for(int i=0; i<m; i++) GS[i]=0;
  int suffix[maxm], k = suffix[m] = m+1;
  for(int j=m; j>0; j--){ //由suffix[j]推出suffix[j-1]
    while(k <= m && p[j-1] != p[k-1]){
      if(GS[k-1]==0) GS[k-1] = k-j; //若在k-1处失配, P右移k-j位
      k = suffix[k];
    }
    suffix[j-1] = --k;
  }
  //未完待续
```



GS表的构建

```
void buildGS(char *p, int m, int GS[]) {
  for(int i=0; i<m; i++) GS[i]=0;
  int suffix[maxm], k = suffix[m] = m+1;
  for(int j=m; j>0; j--){ //由suffix[j]推出suffix[j-1]
    while(k <= m && p[j-1] != p[k-1]){
       if(GS[k-1]==0) GS[k-1] = k-j; //若在k-1处失配, P右移k-j位
       k = suffix[k];
    }
    suffix[j-1] = --k;
  }
  k = suffix[0];
  for(int j = 0; j < m; j++) {
     if(GS[j] == 0) GS[j] = k;
     if(j==k-1) k = suffix[k];
  }
}
```



时间复杂度
O(*m*)