西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

C程序设计
Programming in C

1011014

主讲：姜学锋，计算机学院

字符串查找与匹配算法

1、字符串查找与匹配

2、朴素查找算法

3、KMP算法

4、BM算法

7.5.4 字符串查找与匹配算法

- 一、字符串查找
- 在Word、Visual Studio、Codeblocks等编辑器中都有字符串查找功能。

- 字符串查找算法是一种搜索算法，目的是在一个长的字符串中找出是否包含某个子字符串。

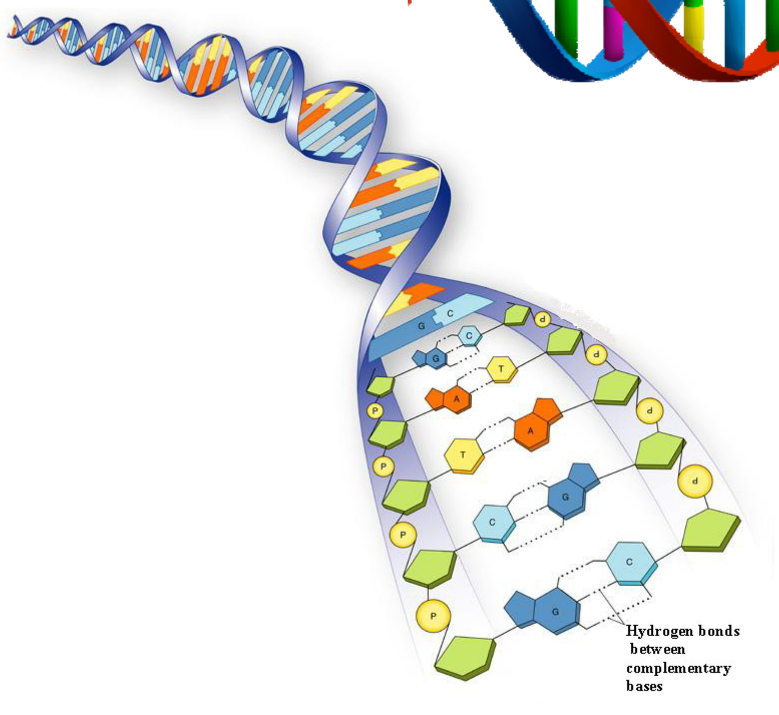
- 二、字符串匹配
- 一个字符串是一个定义在有限字母表Σ上的字符序列。例如，ATCTAGAGA是字母表Σ={A,C,G,T}上的一个字符串。

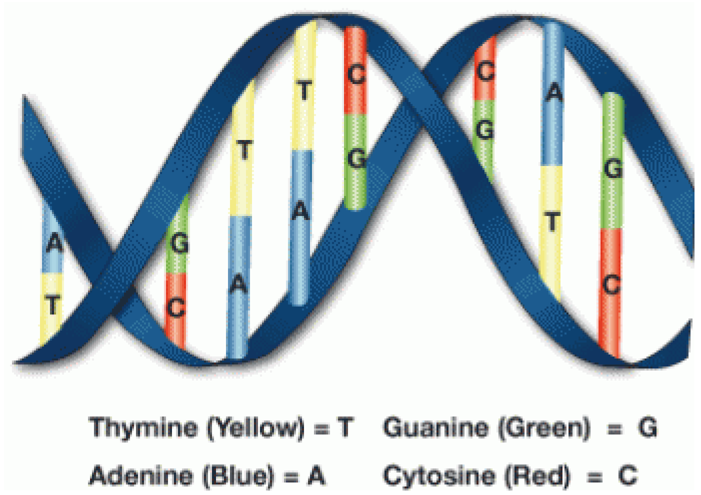
- 字符串匹配算法就是在一个大的字符串T中搜索某个字符串P的所有出现位置。其中，T称为文本，P称为模式，T和P都定义在同一个字母表Σ上。

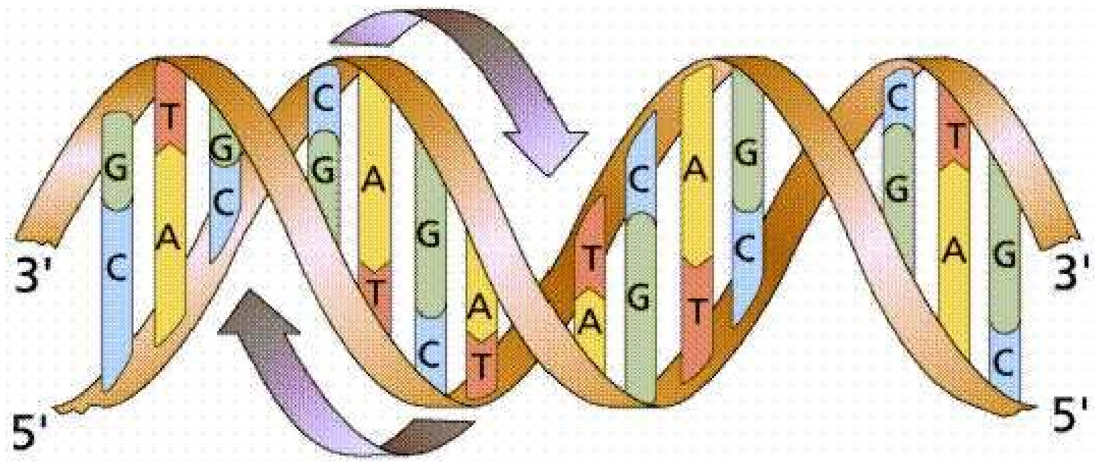
7.5.4 字符串查找与匹配算法

- 字符串匹配的应用包括生物信息学、信息检索、拼写检查、语言翻译、数据压缩、网络入侵检测。

- 在生物信息学中，研究ACGT序列中快速定位某序列的起始位置。

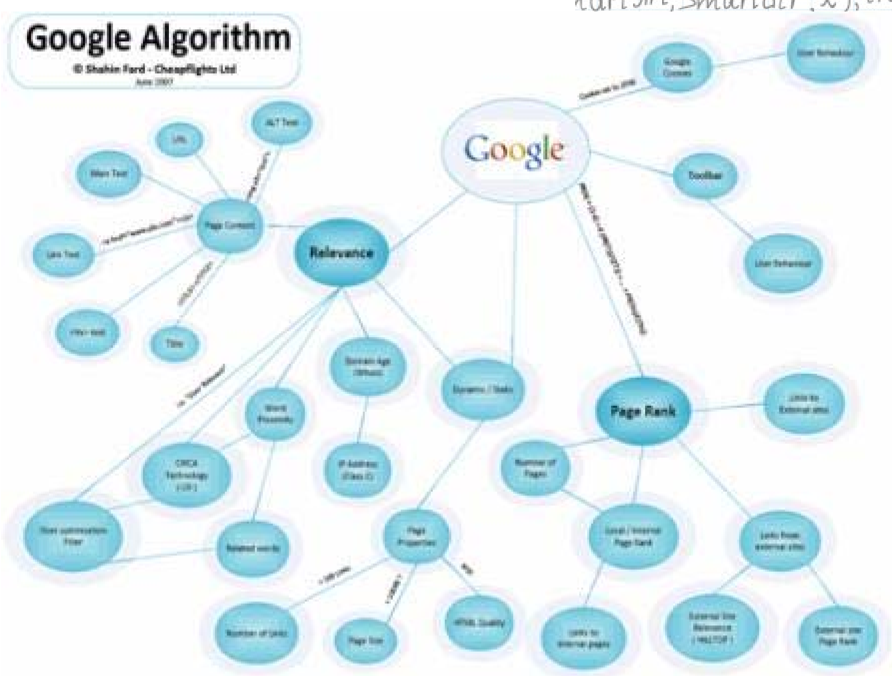


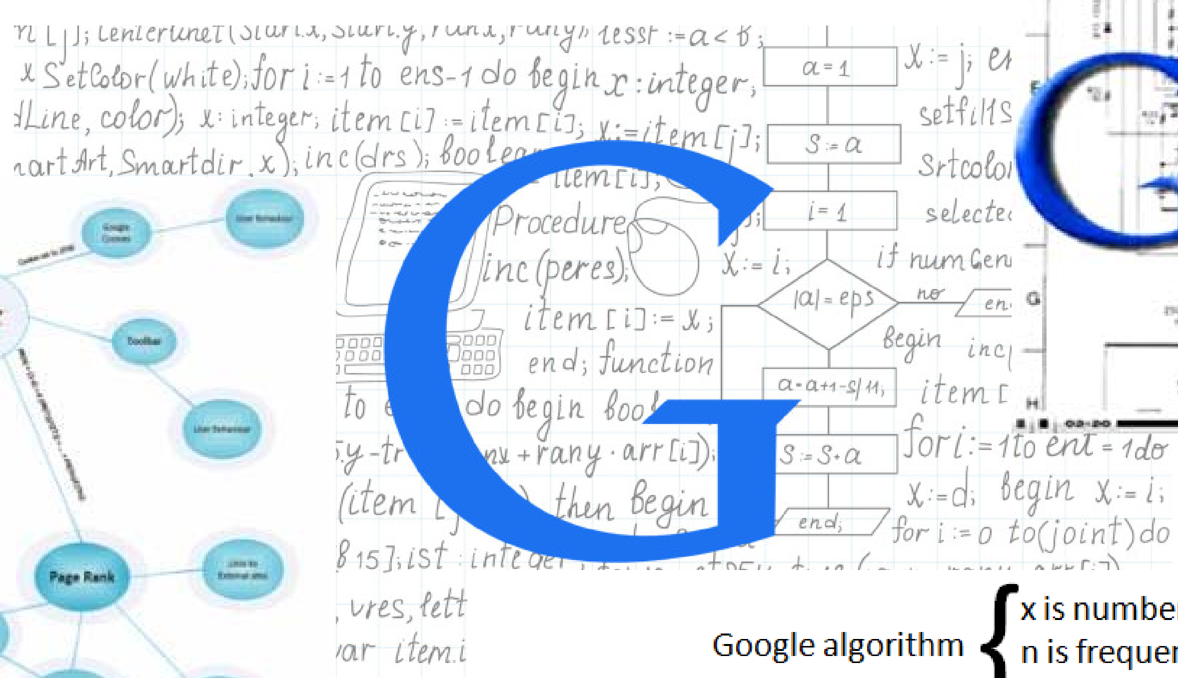


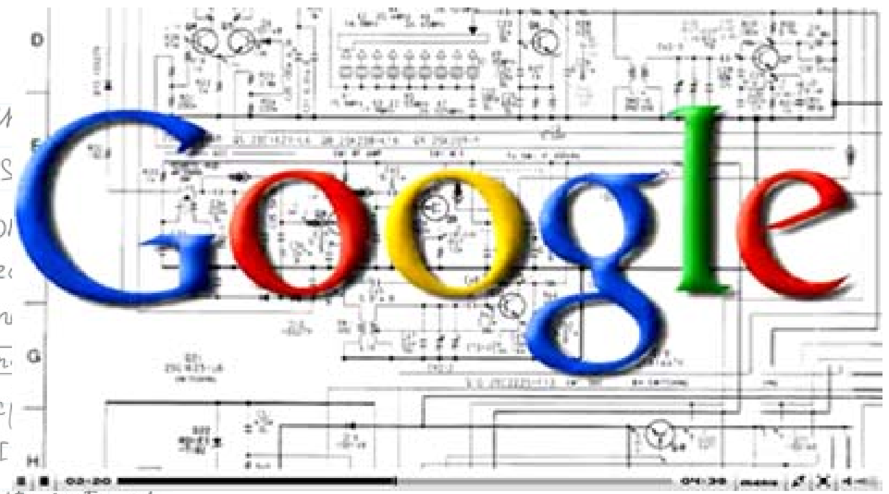


7.5.4 字符串查找与匹配算法

- 在信息检索中，一个挑战性的任务是，搜索出由用户自定义的模式对应在文本中的匹配位置。







Google algorithm $\begin{cases} \text{x is number of links} \\ \text{n is frequency} \\ \text{if keyword density < .05} \end{cases}$

$$\text{google}(x,n)= x + x\left(1+\frac{+1\text{'s}}{n}\right)^{n} + x\left(1+\frac{\text{fb likes}}{2n}\right)^{2n} + x\left(1+\frac{\text{tweets}}{2n}\right)^{2n} + \frac{\text{time on site}}{\text{bounce rate}} + \frac{\text{keyword density}}{\text{words on-page}}$$

- 三、字符串查找与匹配算法
  - 1．朴素查找算法（Naive string search algorithm）
  - 2．KMP算法（Knuth–Morris–Pratt algorithm）
  - 3．BM算法（Boyer–Moore string search algorithm）

7.5.4 字符串查找与匹配算法

- 问题：
- 给定一个文本T字符串和一个模式P字符串，查找P在T中的位置。

- 记号：
- m、n：分别表示T和P的长度。
- Σ：T和P的字符都定义在同一个字母集Σ上。
- T_i、P_j: T的第i个字符和P的第j个字符（以0起始）。其中 $0 \leqslant i < m$，$0 \leqslant j < n$

- 1．朴素查找算法（Naive string search algorithm）

- （1）求解原理：使用暴力匹配（Brute Force algorithm）的思路。

```
B B C  A B C D A B  A B C D A B C D A B D E
A B C D A B D
```

- 1．朴素查找算法（Naive string search algorithm）

- （2）求解方法：假定当前匹配中，文本T匹配到 i 位置，模式P匹配到 j 位置，则：
- ①如果当前字符匹配失败（称为失配，即T[i] != P[j]），令i=i-(j-1)，j=0，即每次失配时，i要回溯，j被置为0。
- ②如果当前字符匹配成功（即T[i] == P[j]），则i++，j++，继续匹配下一个字符；
- ③若j≥n，则T包含P一次，匹配位置=i-j。

7.5.4 字符串查找与匹配算法

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

1）开始时，i=0，j=0

T[0]为B，P[0]为A，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=0-(0-1)=1，j=0。

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

2）接下来，i=1，j=0

T[1]为B，P[0]为A，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=1-(0-1)=2，j=0。

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

3）接下来，i=2，j=0

T[2]为C，P[0]为A，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=1-(0-1)=3，j=0。……

▶ （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

4）接下来，i=4，j=0

T[4]为A，P[0]为A，匹配。执行②：即T[i]==P[j]时，i++，j++。

因此，i=5，j=1。

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23
P					A	B	C	D	A	B	D	\0													n=7
j					0	1	2	3	4	5	6	7													

5）接下来，i=5，j=1

T[5]为B，P[1]为B，匹配。执行②：即T[i]==P[j]时，i++，j++。

因此，i=6，j=2。……

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

6）接下来，i=10，j=6

T[10]为空格，P[6]为D，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=10-(6-1)=5，j=0。

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

7）接下来，i=5，j=0

T[5]为B，P[0]为A，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=5-(0-1)=6，j=0。……

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

8）接下来，i=10，j=2

T[10]为空格，P[2]为C，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=10-(2-1)=9，j=0。……

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

9）接下来，i=17，j=6

T[17]为C，P[6]为D，失配。执行①：即T[i]!=P[j]时，i=i-(j-1)，j=0。

因此，i=17-(6-1)=12，j=0。……

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

10）接下来，i=15，j=0

T[15]为A，P[0]为A，匹配。执行②：即T[i]==P[j]时，i++，j++。

因此，i=16，j=1。……

- （3）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

11）接下来，i=21，j=6

T[21]为D，P[6]为D，匹配。执行②：即T[i]==P[j]时，i++，j++。

因此，i=22，j=7。此时j≥n，执行③：T包含P一次，匹配位置=i-j=15。

7.5.4 字符串查找与匹配算法

- 1．朴素查找算法（Naive string search algorithm）

- （4）算法性能：时间复杂度是O(m(n-m+1))=O(mn)
- 最坏情况：
- T = aaa ... ah
- P = aaah

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

- （5）算法特点：每次比较总是从起点开始，即T的下一个字符、P的起始字符。

- 1．朴素查找算法（Naive string search algorithm）
- （6）算法代码

```
Algorithm BruteForceMatch(T, P)
    Input 文本串T和模式串P
    Output P在T的起始位置或者–1（若T没有包含P）
    for i ← 0 to m – n
        { 移位模式串下标比较}
        j ← 0
        while j < n ∧ T[i + j] = P[j]
            j ← j + 1
        if j = n
            return i {匹配位置i}
        else
            break while loop {不匹配}
    return -1 {不匹配}
```

【例7.53】

编写字符串查找程序。

A B C A B C D A B A B C D A B C D A B D E

A B C D A B D

7.5.4 字符串查找与匹配算法

例7.53

```c
#include <stdio.h>
#include <string.h>
int BruteForceMatch(char T[], char P[])
{
  int m,n,i=0,j=0;
  m = strlen(T);
  n = strlen(P);
  while (i<m && j<n) {
    if (T[i]==P[j])
      i++,j++; //①如果当前字符匹配成功，则i++, j++
    else { //②如果失配，则i=i-(j-1), j=0
      i=i-j+1;
      j=0;
    }
  }
```

7.5.4 字符串查找与匹配算法

例7.53

```
  if (j>=n)
    return i-j; //匹配成功，返回P在T中第1次出现的位置
  else
    return -1; //否则返回-1
}
int main()
{
  char T[]="ABC ABCDAB ABCDABCDABDE";
  char P[]="ABCDABD";
  printf("%s在%s",P,T);
  if (BruteForceMatch(T,P)>=0)
    printf("找到!");
  else
    printf("未找到!");
  return 0;
```

- 2．KMP算法（Knuth–Morris–Pratt algorithm）

- （1）算法原理：在前面的暴力匹配BF算法中：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

i=10，j=6时失配，则i要回溯到i=10-(6-1)=5，j=0，重新比较。

7.5.4 字符串查找与匹配算法

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

BF算法

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

新算法

P	A	B	C	D	A	B	D	\0
j	0	1	2	3	4	5	6	7

T[5]肯定与P[0]失配，因为在前面的步骤，已经得知T[5]=P[1]=B，而P[0]=A，即P[1]!=P[0]，故T[5]必定不等于P[0]，所以回溯过去必然会导致失配。依此类推，T[6]、T[7]也是如此，故从T[8]开始比较会更好。

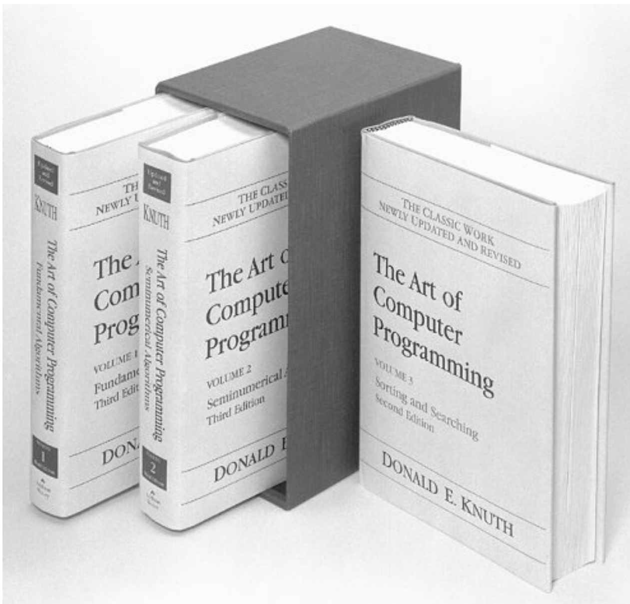
- 2．KMP算法（Knuth–Morris–Pratt algorithm）

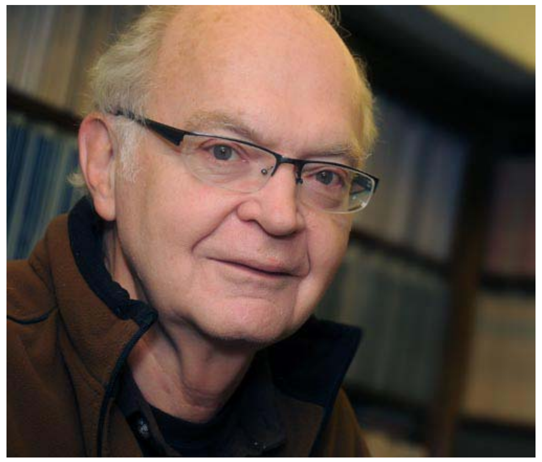
- （1）算法原理：KMP算法是一种高效的前缀匹配算法，在暴力匹配BF算法的基础上改进而来。它利用之前已经部分匹配的有效信息，保持i不回溯，通过修改 j 的位置，让模式串P尽量地移动到有效的位置，每次移动的距离可以不是1而是更大。

```
BBC ABCDAB ABCDABCDABDE
 ABCDABD
```

7.5.4 字符串查找与匹配算法

- 2．KMP算法（Knuth–Morris–Pratt algorithm）
- KMP算法由Donald Knuth(唐纳德·克努特)、Vaughan Pratt(沃恩·普拉特)、James H. Morris(杰姆斯·莫里斯)三人于1977年联合发表，取3人的姓氏命名此算法。



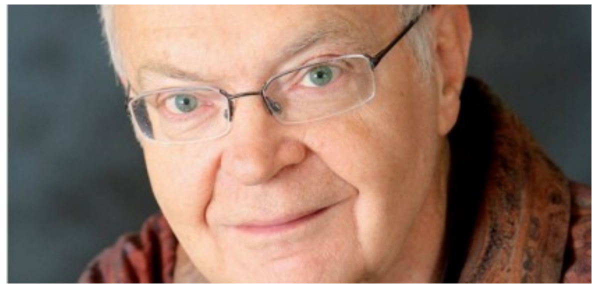


Donald Knuth

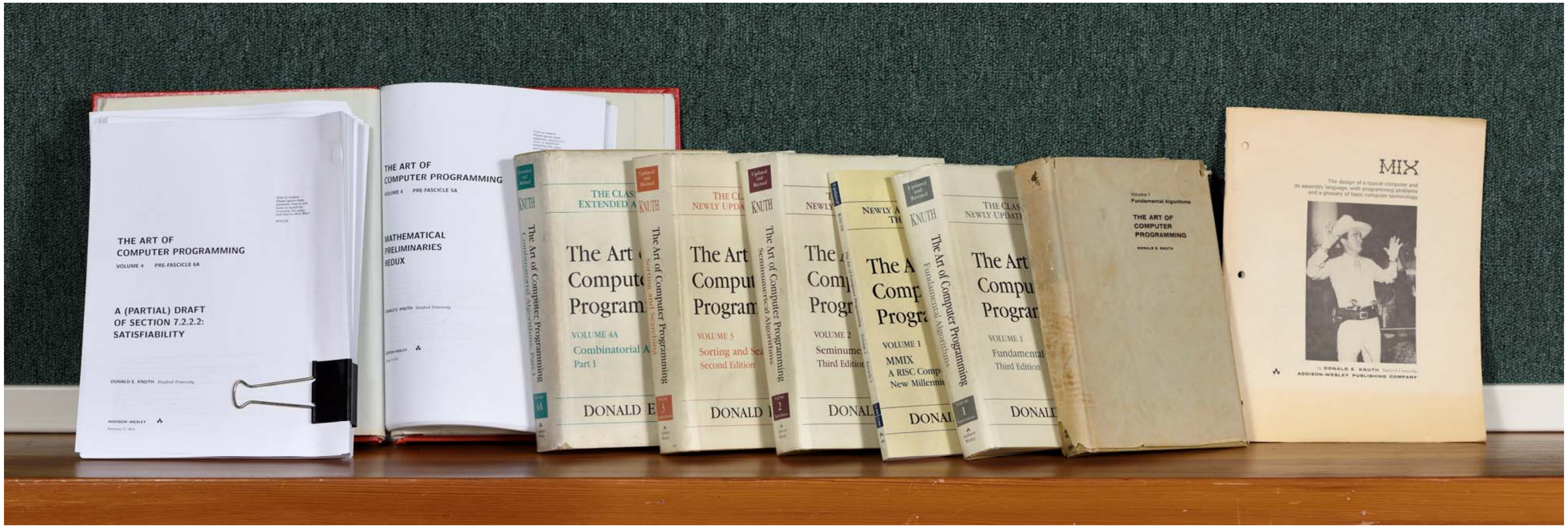
Vaughan Pratt

James H. Morris

7.5.4 字符串查找与匹配算法



- Donald Knuth(唐纳德·克努特)。
- 《计算机程序设计艺术（The Art of Computer Programming）》，简称TAOCP

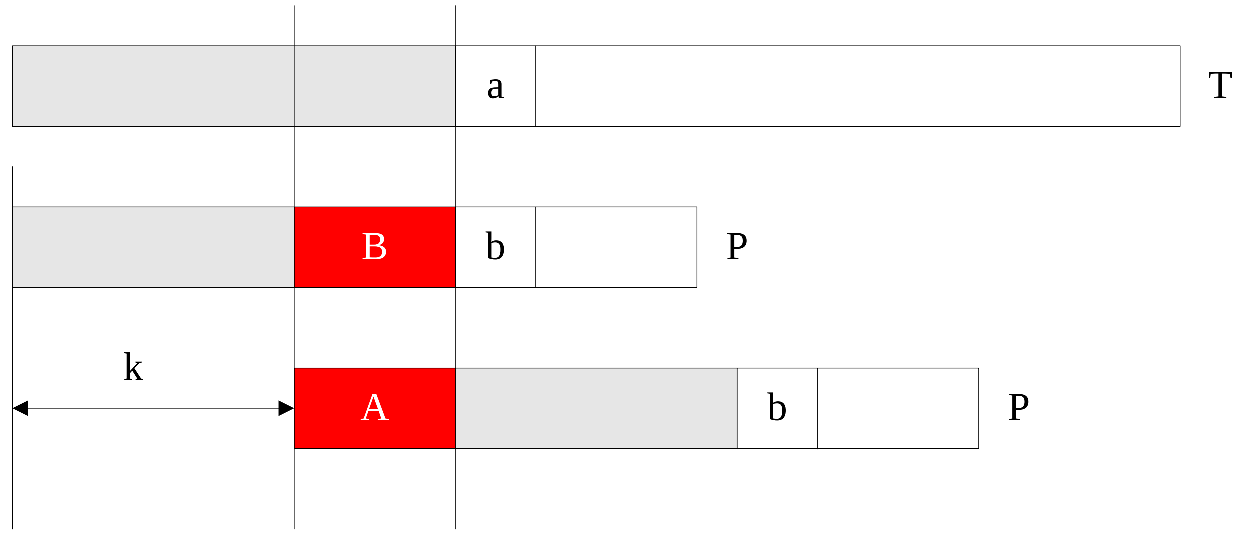


7.5.4 字符串查找与匹配算法

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

- （2）算法思想：
- 假设根据已经获得的信息知道可以前移k位，我们分析移位前后P有什么特点。可以得到如下的结论：
  - A段字符串是P的一个前缀。
  - B段字符串是P的一个后缀。
  - A段字符串和B段字符串相等。



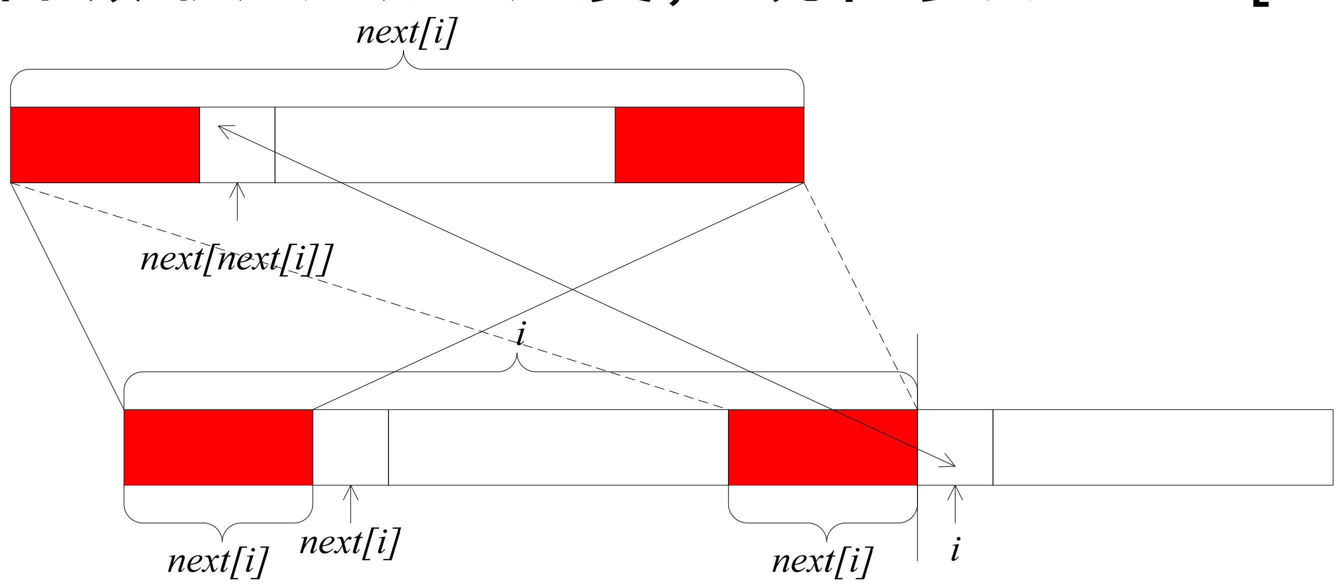
7.5.4 字符串查找与匹配算法

- KMP算法的核心是计算模式串P每一个位置之前的字符串的前缀和后缀公共部分的最大长度（不包括字符串本身，否则最大长度始终是字符串本身）。
- 获得P每一个位置的最大公共长度之后，就可以利用该最大公共长度快速和文本串T比较。当每次比较到两个字符串的字符不同时，我们就可以根据最大公共长度将模式串P向前移动(已匹配长度-最大公共长度)位，接着继续比较下一个位置。事实上，模式串P的前移只是概念上的前移，只要我们在比较的时候从最大公共长度之后比较P和T即可达到字符串P前移的目的。

A B C　A B C D A B　A B C D A B C D A B D E

A B C D A B D

- next数组计算
- 理解了KMP算法的基本原理，下一步就是要获得模式串P每一个位置的最大公共长度，记为next数组。假设我们现在已经求得next[1]、next[2]、......next[i]，分别表示长度为1到i的字符串的前缀和后缀最大公共长度，现在要求next[i+1]。



7.5.4 字符串查找与匹配算法

- 由上图我们可以看到，如果位置i和位置next[i]处的两个字符相同，则next[i+1]等于next[i]加1。如果两个位置的字符不相同，我们可以将长度为next[i]的字符串继续分割，获得其最大公共长度next[next[i]]，然后再和位置i的字符比较。这是因为长度为next[i]前缀和后缀都可以分割成上部的构造，如果位置next[next[i]]和位置i的字符相同，则next[i+1]就等于next[next[i]]加1。如果不相等，就可以继续分割长度为next[next[i]]的字符串，直到字符串长度为0为止。

- 2．KMP算法（Knuth–Morris–Pratt algorithm）

- （3）求解方法：假定当前匹配中，文本T匹配到 i 位置，模式P匹配到 j 位置，则：
- ①如果j!=-1，且当前字符匹配失败（即T[i]!=P[j]），令 i 不变，j=next[j]。此举意味着失配时，模式串P相对于文本串T向右移动了j-next[j] 位；
- ②如果j=-1，或者当前字符匹配成功（即T[i]==P[j]），令i++，j++，继续匹配下一个字符。
- ③若j≥n，则T包含P一次，匹配位置=i-j。

- 2．KMP算法（Knuth–Morris–Pratt algorithm）

- 换言之，当匹配失败时，模式串P向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值，即移动的实际位数为：j - next[j]，且此值大于等于1。
- 其中，next 数组各值的含义代表当前字符之前的字符串中，有多大长度的相同前缀后缀。例如若next[j]=k，代表 j 之前的字符串中有最大长度为 k 的相同前缀后缀。

▸ 2．KMP算法（Knuth–Morris–Pratt algorithm）

```
A B C   A B C D A B   A B C D A B C D A B D E
A B C D A B D
```

- 2．KMP算法（Knuth–Morris–Pratt algorithm）

- 此意味着在某个字符失配时，该字符对应的next值会确定下一步匹配中，模式串应该跳到哪个位置（跳到next [j] 的位置）。如果next [j] 等于0或-1，则跳到模式串P的起始字符，若next [j] = k 且 k > 0，代表下次匹配跳到 j 之前的某个字符，而不是跳到起始，且具体跳过了 k 个字符。

- 1）最长前缀后缀：给定模式串P="ABCDABD"，从左至右遍历整个P，其各个子串的前缀后缀分别如下表所示：

P的子串	前缀	后缀	最大公共元素长度
A	空	空	**0**
AB	A	B	**0**
ABC	A,AB	C,BC	**0**
ABCD	A,AB,ABC	D,CD,BCD	**0**
ABCDA	**A**,AB,ABC,ABCD	**A**,DA,CDA,BCDA	**1**
ABCDAB	A,**AB**,ABC,ABCD,ABCDA	B,**AB,**DAB,CDAB,BCDAB	**2**
ABCDABD	A,AB,ABC,ABCD,ABCDA,ABCDAB	D,BD,ABD,DABD,CDABD,BCDABD	**0**

- 2）最大长度表：模式串P子串对应的各个前缀后缀的公共元素最大长度表如下：

模式串P	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0

- 失配时，模式串P向右移动的位数为：
- 已匹配字符数 - 失配字符上一位字符所对应的最大长度值

- 当匹配到一个字符失配时，其实没必要考虑当前失配的字符，只看失配字符的上一位字符对应的最大长度值。如此，便得出next 数组。

模式串P	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**
索引	0	1	2	3	4	5	6

- **next数组**：寻找最大对称长度的前缀后缀，然后整体右移一位，初值赋为-1。

- 当失配时，模式串P向右移动的位数为：
- **失配字符所在位置 - 失配字符对应的next值**

```
A B C   A B C D A B   A B C D A B C D A B D E
A B C D A B D
```

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

- （4）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	
j	0	1	2	3	4	5	6	7	n=7

1）开始时，i=0，j=0
P[0]=A与T[0]=B失配。执行①：如果j!=-1，且当前字符匹配失败（即T[i]!=P[j]），令 i 不变，j=next[j]。得到j=-1。转而执行②：如果j=-1，或者当前字符匹配成功（即T[i]==P[j]），令i++，j++，得到i=1，j=0，即P[0]继续跟T[1]匹配。

7.5.4 字符串查找与匹配算法

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

- （4）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

2）接下来，i=1，j=0
P[0]与T[1]失配。执行①：j=next[j]，j=-1。执行②：i++，j++，得到i=2，j=0，即P[0]继续跟T[2]匹配。……

7.5.4 字符串查找与匹配算法

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

- （4）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23
P					A	B	C	D	A	B	D	\0													n=7
j					0	1	2	3	4	5	6	7													

3）接下来，i=4，j=0

P[0]与T[4]匹配。执行②：i++，j++，得到i=5，j=1。

- （4）求解分析：

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

4）接下来，i=10，j=6
P[6]与T[10]失配。执行①： i 不变，j=next[j]，得到j=2。所以下一步用P[2]继续跟T[10]匹配，相当于P向右移动：j-next[j]=6-2=4位。

i	0	1	2	3	4	5	6
next	-1	0	0	0	0	1	2

- （4）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	
j	0	1	2	3	4	5	6	7	n=7

5）接下来，i=10，j=2
P[2]与T[10]失配。执行①： i 不变，j=next[j]，得到j=0。所以下一步用P[0]继续跟T[10]匹配，相当于P向右移动：j-next[j]=2-0=2 位。

- （4）求解分析：

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

6）接下来，i=10，j=0
P[0]与T[10]失配。执行①：j=next[j]，j=-1。执行②：i++，j++，得到i=11，j=0，即P[0]继续跟T[11]匹配。

- （4）求解分析：

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

7）接下来，i=17，j=6

P[6]与T[17]失配。执行①： i 不变，j=next[j]，得到j=2。所以下一步用P[2]继续跟T[10]匹配，相当于P向右移动：j-next[j]=6-2=4位。

▶ （4）求解分析：

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23

P	A	B	C	D	A	B	D	\0	n=7
j	0	1	2	3	4	5	6	7	

8）接下来，i=17，j=2
P[2]与T[17]匹配。执行②：i++，j++，得到i=18，j=3。……

7.5.4 字符串查找与匹配算法

i	0	1	2	3	4	5	6
next	**-1**	**0**	**0**	**0**	**0**	**1**	**2**

- （4）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
T	B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E	\0	m=23
P																A	B	C	D	A	B	D	\0		n=7
j																0	1	2	3	4	5	6	7		

9）接下来，i=21，j=6

P[6]与T[21]匹配。执行②：i++，j++，得到i=21，j=7。此时j≥n，执行③：T包含P一次，匹配位置=i-j=15。

- （5）next优化：

i	0	1	2	3	4	5	6	7	8	9	
T	A	B	A	C	A	B	A	B	C	\0	m=9
P	A	B	A	B	\0						n=4
j	0	1	2	3	4						

P的子串	前缀	后缀	最大公共元素长度
A	空	空	**0**
AB	A	B	**0**
ABA	**A**,AB	**A**,BA	**1**
ABAB	A,**AB**,ABA	B,**AB**,BAB	**2**

i	0	1	2	3
next	**-1**	**0**	**0**	**1**

i	0	1	2	3
next	**-1**	**0**	**0**	**1**

- （5）next优化：

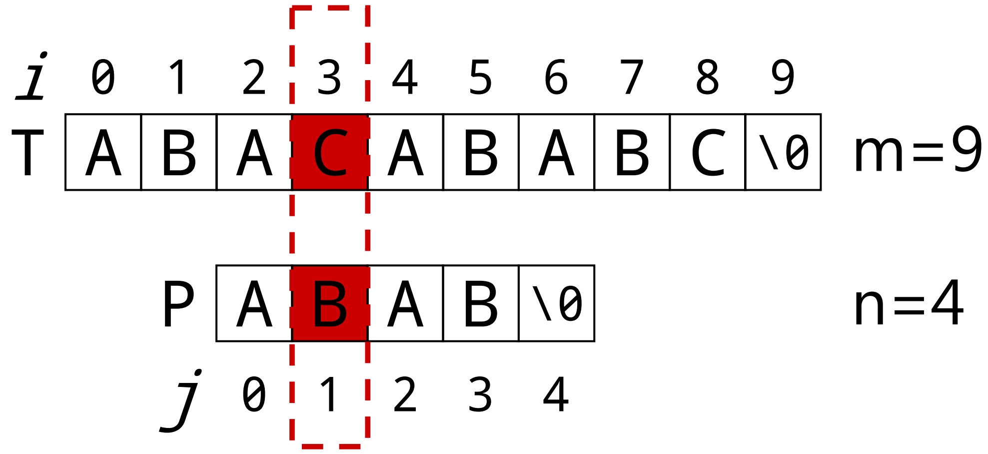
i	0	1	2	3	4	5	6	7	8	9	
T	A	B	A	C	A	B	A	B	C	\0	m=9
P	A	B	A	B	\0						n=4
j	0	1	2	3	4						

1）当进行P跟T匹配时

发现P[3]跟T[3]失配，于是模式串P右移 j-next[j]=3-1=2位。

i	0	1	2	3
next	**-1**	**0**	**0**	**1**

- （5）next优化：



2）右移2位后，P[1]又跟T[3]失配。事实上，在上一步的匹配中，已经得知P[3]=B，与T[3]=C失配，而右移2位之后，让P[next[3]]=P[1]=B再跟T[3]匹配时，必然失配。问题出在哪呢？

i	0	1	2	3
next	**-1**	**0**	**0**	**1**

- （5）next优化：

i	0	1	2	3	4	5	6	7	8	9	
T	A	B	A	C	A	B	A	B	C	\0	m=9

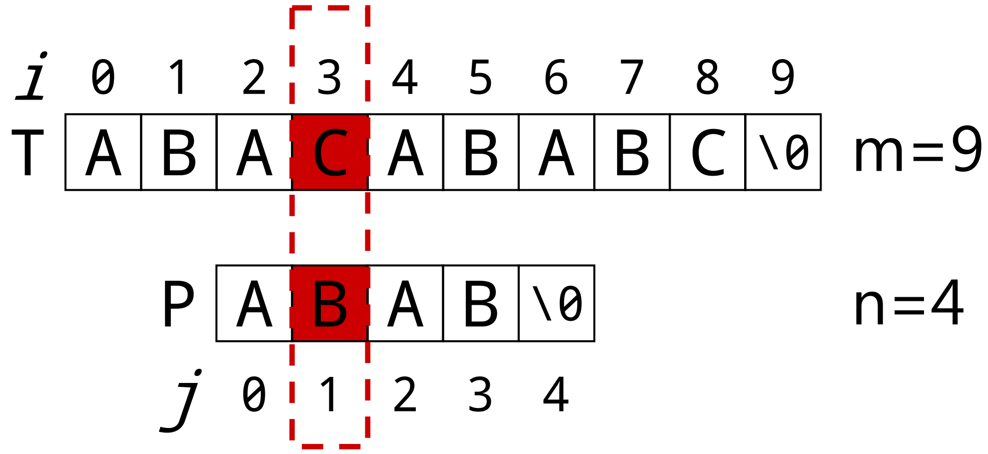
P	A	B	A	B	\0	n=4
j	0	1	2	3	4	

3）问题出在不该出现 P[j]=P[next[j]]。为什么呢？理由是：当P[j] !=T[i]时，下次匹配必然是P[next [j]] 跟T[i]匹配，如果P[j] = P[next[j]]，必然导致后一步匹配失败。所以不允许 P[j]=P[next[j]]。如果出现了 P[j] = P[next[j]]咋办呢？如果出现了，则需要再次递归，即令next[j] = next[next[j]]。

7.5.4 字符串查找与匹配算法

i	0	1	2	3
next	**-1**	**0**	**0**	**1**
优化next	**-1**	**0**	**-1**	**0**

- （5）next优化：



4）只要出现了P[next[j]] = P[j]的情况，则把next[j]的值再次递归。

next[2] = next[next[2]] = next[0] = -1

next[3] = next[next[3]] = next[1] = 0

i	0	1	2	3
next	**-1**	**0**	**0**	**1**
优化next	**-1**	**0**	**-1**	**0**

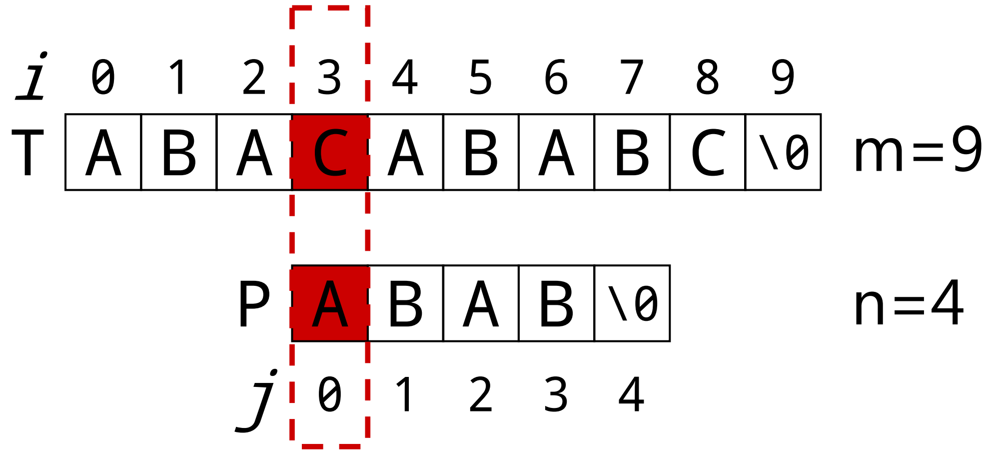
- （5）next优化：

i	0	1	2	3	4	5	6	7	8	9	
T	A	B	A	C	A	B	A	B	C	\0	m=9
P	A	B	A	B	\0						n=4
j	0	1	2	3	4						

5）P[3]与T[3]失配。

i	0	1	2	3
next	**-1**	**0**	**0**	**1**
优化next	**-1**	**0**	**-1**	**0**

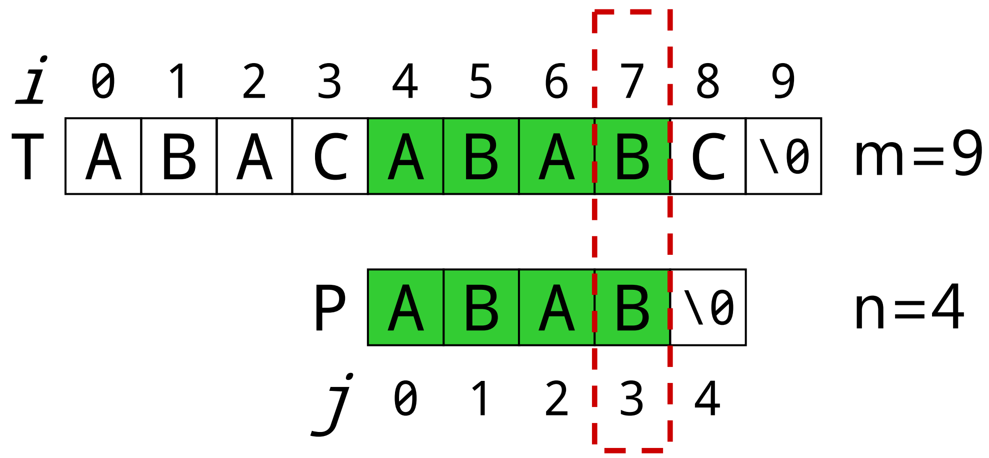
- （5）next优化：



6）T[3]保持不变，P的下一个匹配位置是P[next[3]]，而next[3]=0，所以P[next[3]]=P[0]与T[3]匹配。

- （5）next优化：

i	0	1	2	3
next	**-1**	**0**	**0**	**1**
优化next	**-1**	**0**	**-1**	**0**



7）由于P[0]与T[3]不匹配。此时i=3，j=next [0]=-1，执行②：i++，j++，得到i=4，j=0。......

最后j≥n，执行③：T包含P一次，匹配位置=i-j=4。

- 2．KMP算法（Knuth–Morris–Pratt algorithm）

- （6）算法性能：BF算法的时间复杂度是O(m*n)，KMP算法的时间复杂度是O(m+n)。

- 2．KMP算法（Knuth–Morris–Pratt algorithm）
- （7）算法代码：

```
Algorithm KMPMatch(T, P)
    F ←失配函数next(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { 匹配 }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { 不匹配 }
```

【例7.54】

编写字符串查找程序，使用KMP算法。

```
A B C   A B C D A B   A B C D A B C D A B D E
A B C D A B D
```

7.5.4 字符串查找与匹配算法

例7.54

```c
#include <stdio.h>
#include <string.h>
void preNext(char P[], int next[])
{
  int m, k=-1, j=0;
  m = strlen(P);
  next[0] = -1;
  while (j<m-1) { //P[k]表示前缀，P[j]表示后缀
    if (k==-1 || P[j]==P[k]) {
      ++j, ++k;
      if (P[j]!=P[k]) next[j]=k;
            else
        next[j]=next[k]; //避免p[j]=p[next[j]]，需要递归
    }
    else k=next[k];
```

7.5.4 字符串查找与匹配算法

例7.54

```
  }
}
int KMP(char T[], char P[])
{
 int i=0, j=0, m,n;
 int next[1000];
 m = strlen(P);
 n = strlen(T);
 preNext(P,next);
 while (i<n && j<m)
 {
  //①如果j=-1，或者当前字符匹配成功（即T[i]==P[j]）,令i++，j++
  if (j==-1 || T[i]==P[j])
   i++, j++;
```

7.5.4 字符串查找与匹配算法

例7.54

```
  else
   //②如果j!=-1，且当前字符匹配失败（即T[i]!=P[j]），令i不变，j=next[j]
   j=next[j];
 }
 if (j>=m)
  return i-j; //匹配成功，返回P在T中第1次出现的位置
 else
  return -1; //否则返回-1
}
int main()
{
 char T[]="ABC ABCDAB ABCDABCDABDE";
 char P[]="ABCDABD";
 printf("%s在%s",P,T);
```

7.5.4 字符串查找与匹配算法

例7.54

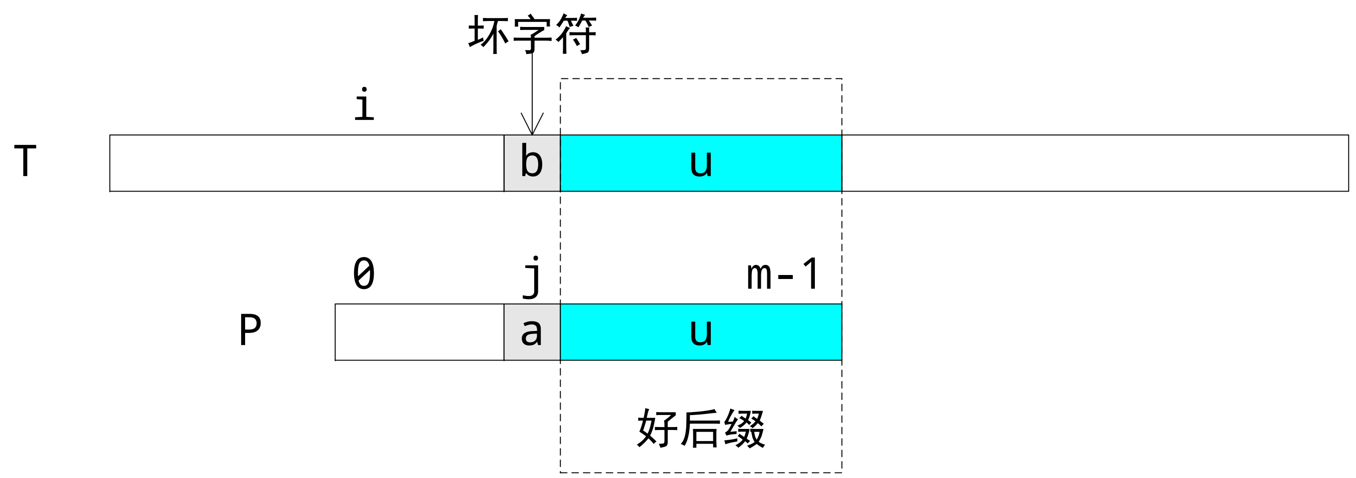
```
  if (KMP(T,P)>=0)
    printf("找到!");
  else
    printf("未找到!");
  return 0;
}
```

- 3．BM算法（Boyer–Moore string search algorithm）

- （1）算法原理：BM算法是一种基于后缀匹配的模式串匹配算法，后缀匹配就是模式串从右向左开始比较，但模式串的移动还是从左到右的。字符串匹配的关键是模式串如何移动，BM为了做到这点定义了两个规则：好后缀规则和坏字符规则。

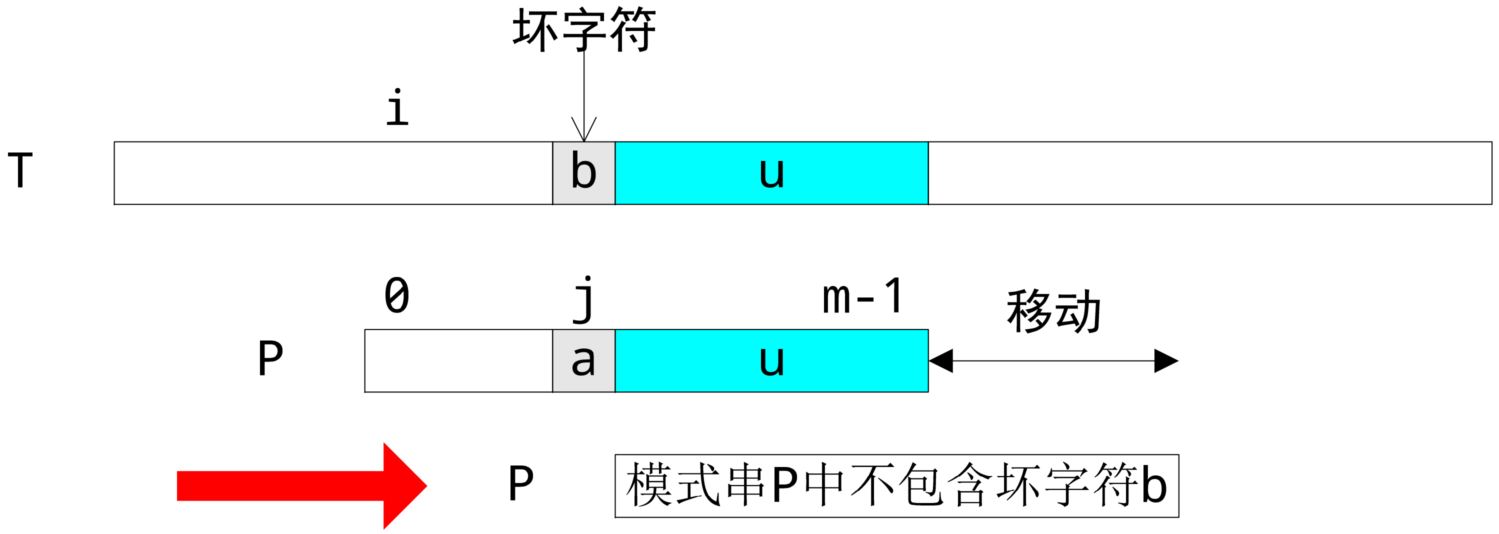
```
B B C   A B C D A B   A B C D A B C D A B D E
A B C D A B D
```

- 假定失配发生在模式串P[j]=a和文本串T[i+j]=b之间（从文本串 i 开始比较），即P[j+1 .. m-1]=T[i+j+1 .. j+m-1]=u，且P[j]!=T[i+j]。

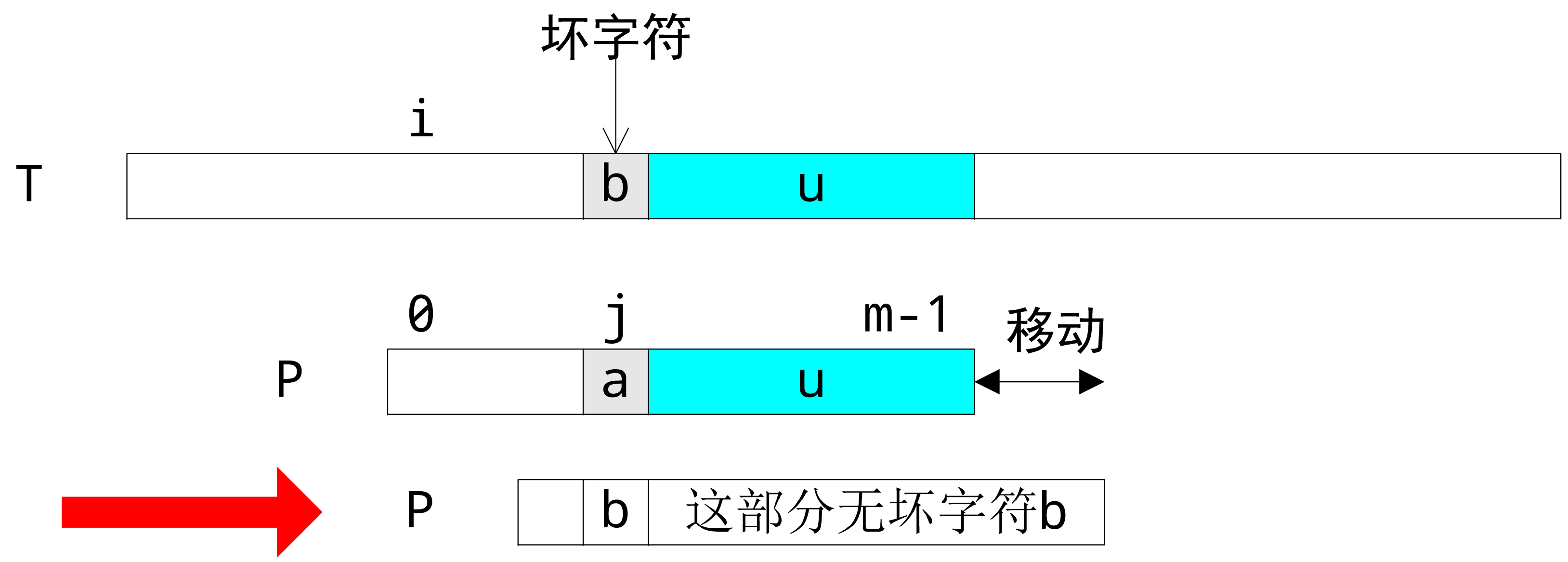


- 则T[i+j]为坏字符（bad-character），u为好后缀（good-suffix），并且好后缀子串也是好后缀。

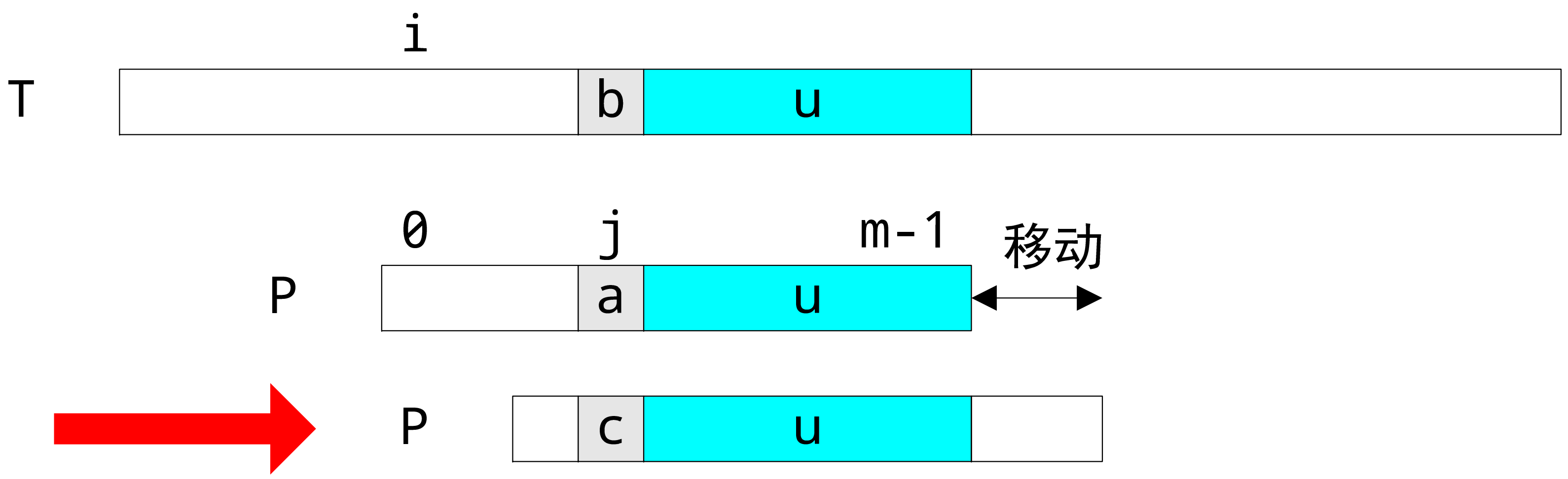
- 坏字符规则
- 1）如果坏字符没有出现在模式串P中，则直接将P移到坏字符的后一个字符（即T[i+j+1]）位置上。



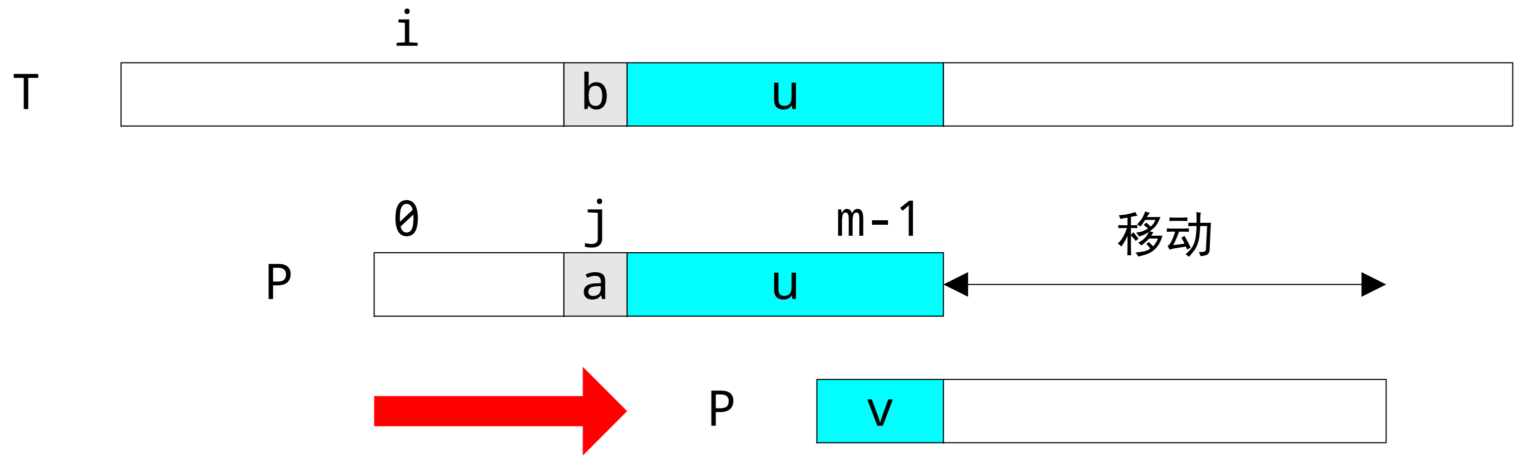
- 坏字符规则
- 2）如果坏字符出现在模式串P中，则将P中最靠近好后缀u的坏字符与T的坏字符（即T[i+j]）对齐。



- 好后缀规则
- 1）模式串P中有子串匹配上好后缀u，此时移动P，让该子串和T的好后缀u对齐即可；如果P中有超过一个子串匹配上好后缀u，则选择P最靠近好后缀的子串对齐。

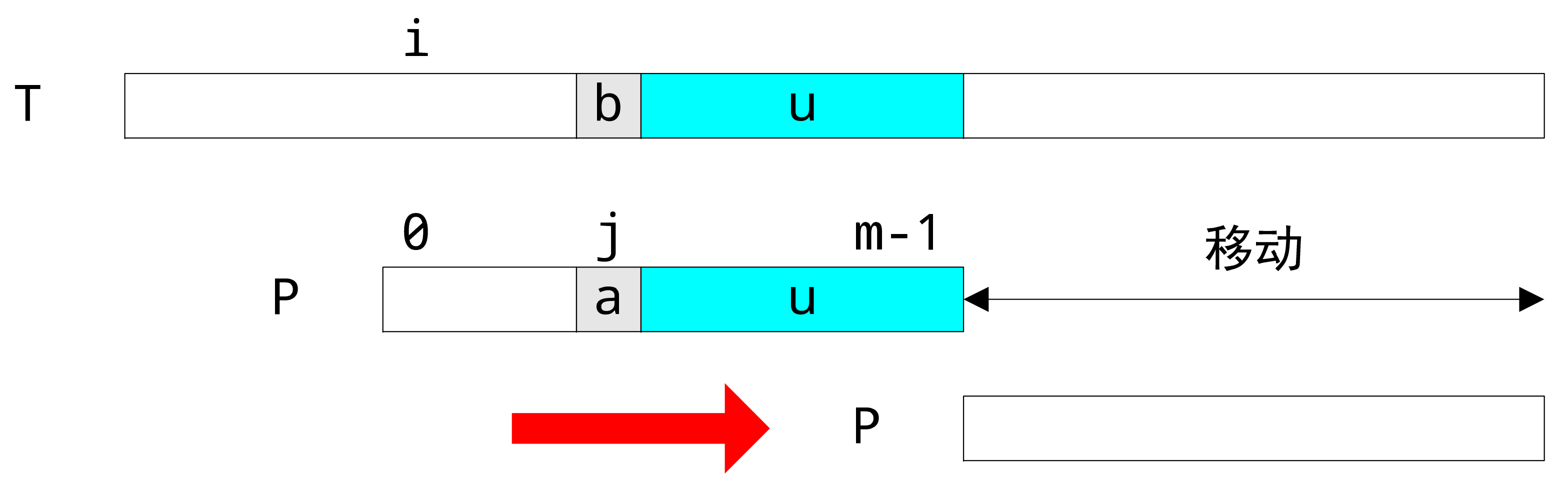


- 好后缀规则
- 2）模式串P中没有子串匹配上好后缀u，此时需要寻找P的一个最长前缀，让该前缀等于好后缀u的部分后缀v（v是u最右边的一部分），寻找到该前缀后，让该前缀和后缀v对齐。



- 1)和2)可以看成P还含有好后缀（好后缀子串也是好后缀）。

- 好后缀规则
- 3）模式串P中没有子串匹配上好后缀，并且在P中也找不到最长前缀等于好后缀的后缀。此时，直接移动P到好后缀u的下一个字符。



- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24
P	E	X	A	M	P	L	E	\0																		n=7
j	0	1	2	3	4	5	6	7																		

1）开始时，T与P头部对齐，从P尾部开始比较。

BM的确是一个很聪明的想法，因为如果尾部字符不匹配，那么只要一次比较，就可以知道前7个字符肯定不是要找的结果。

- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24
P	E	X	A	M	P	L	E	\0																		n=7
j	0	1	2	3	4	5	6	7																		

2）接下来，i=0，j=6。

P[6]与T[6]失配，T[6]=S是坏字符，且S不包含在P中。P直接移到T[6]的后一位。

- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24

P	E	X	A	M	P	L	E	\0	n=7
j	0	1	2	3	4	5	6	7	

3）接下来，i=7，j=6。

P[6]与T[13]失配，T[13]=P是坏字符，但"P"包含在模式串P中。所以，将P后移2位，使坏字符对齐。

- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24
P										E	X	A	M	P	L	E	\0									n=7
j										0	1	2	3	4	5	6	7									

4）接下来，i=9，j=6。

比较前一位，发现T[11]与P[2]失配，所以I是坏字符。此时P应该后移 2 –（-1）= 3 位。

- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24

P	E	X	A	M	P	L	E	\0	
j	0	1	2	3	4	5	6	7	n=7

4）接下来，i=9，j=6。

好后缀"MPLE"匹配（"MPLE"、"PLE"、"LE"、"E"都是好后缀）。所有好后缀中，只有"E"在P中出现两次，所以 后移位数 = 好后缀位置 – P中上一次出现位置=6 – 0 = 6位。

- （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24

P	E	X	A	M	P	L	E	\0	n=7
j	0	1	2	3	4	5	6	7	

4）接下来，i=9，j=6。

可以看到，坏字符规则只能移3位，好后缀规则可以移6位。所以，BM算法取两个规则之中的较大值。

▶ （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24

P	E	X	A	M	P	L	E	\0	
j	0	1	2	3	4	5	6	7	n=7

5）接下来，i=15，j=6。

P[6]与T[21]失配，T[21]=P是坏字符，根据坏字符规则，后移 6 – 4 = 2位。

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

▶ （2）求解分析：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E	\0	m=24

P	E	X	A	M	P	L	E	\0	n=7
j	0	1	2	3	4	5	6	7	

6）接下来，i=17，j=6。

从尾部开始逐位比较，发现全部匹配，于是搜索结束。匹配位置=i=17。

CP 程序设计