第4章 串

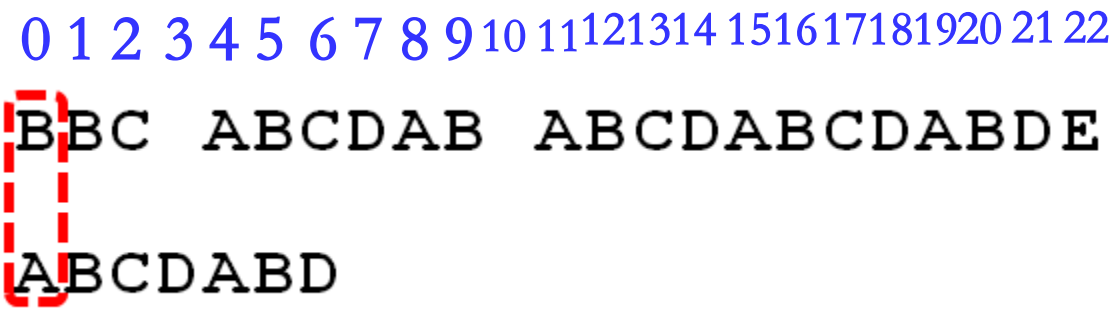
4.1 串的基本概念

4.2 串的存储结构

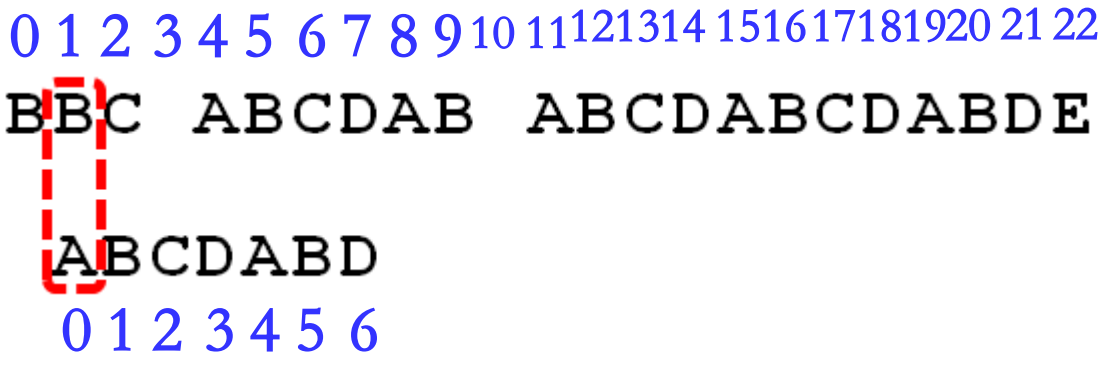
4.3 串的模式匹配

串的模式匹配

- Brute-Force算法回顾



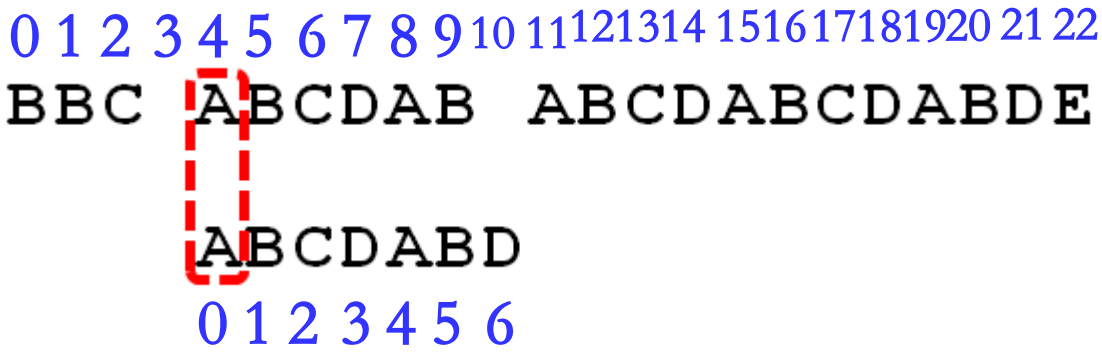
S[0]为B，T[0]为A，不匹配→ 文本串右移1位



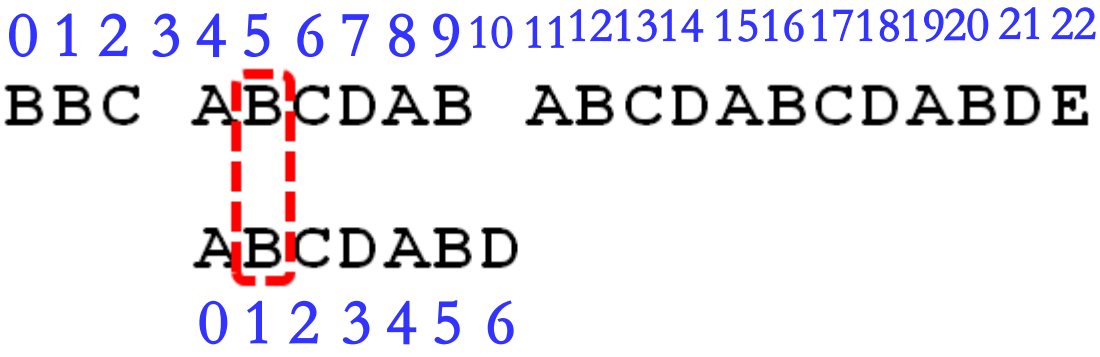
S[1]为B，T[0]为A，不匹配→ 文本串右移1位

串的模式匹配

- Brute-Force算法回顾



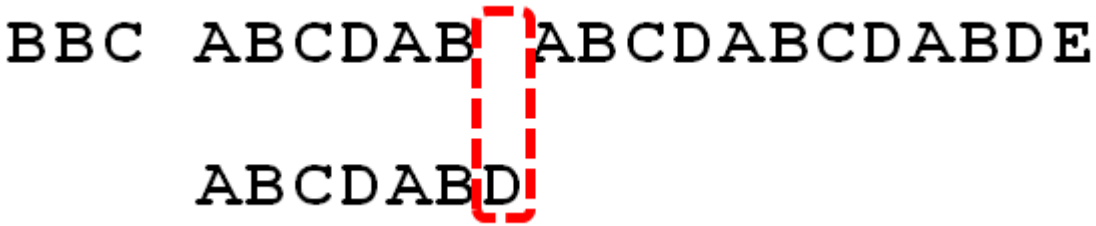
S[4]为A，T[0]为A，匹配→ 文本串、模式串均右移1位



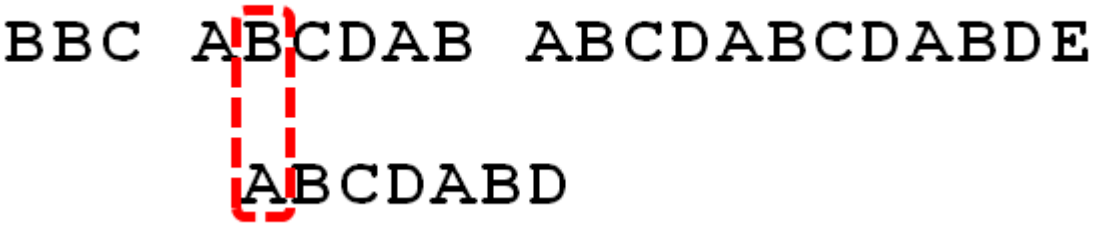
文本串、模式串继续右移1位

串的模式匹配

- Brute-Force算法回顾



S[10]为” ”，T[6]为D，不匹配，文本串→S[5]，模式串→T[0]



S[5]与T[0]必然失配。之前匹配信息可知S[5]=T[1];T串自身可知T[0]!=T[1] → S[5]!=T[0]，失配

串的模式匹配

- Brute-Force算法回顾
  - 主串回溯、模式串回溯
- 改进方案
  - 如何利用已有的匹配结果、模式串的自身前缀、后缀之间的关系，减少不必要的指针回溯？
    - 主串指针不回溯
    - 模式串少回溯

串的模式匹配

- KMP算法

Vol. 6, No. 2, June 1977

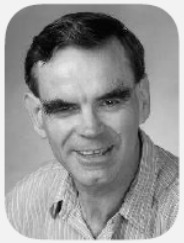
FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Don Knuth

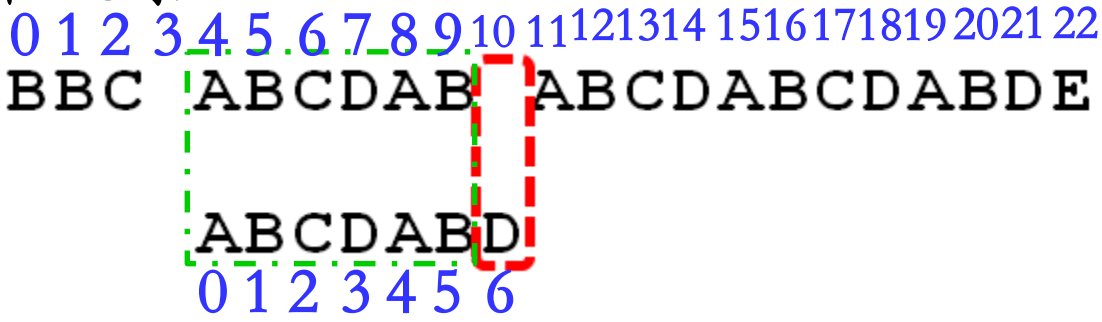
Jim Morris



Vaughan Pratt

KMP算法

- 基本观察

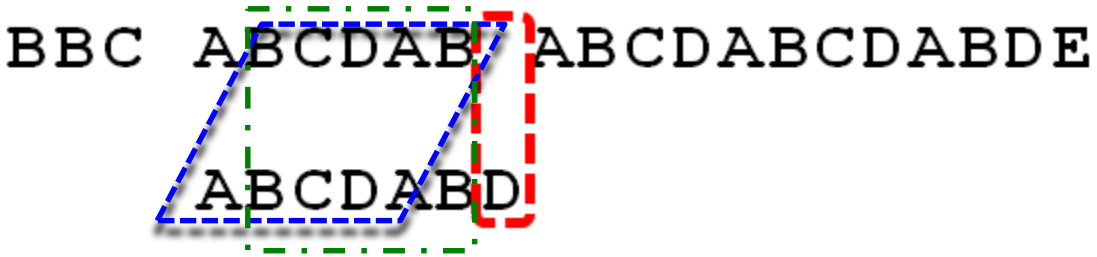


S[10]与T[6]失配（注意，模式串失配位置为6），意味着:
S[4]S[5]S[6]S[7]S[8]S[9]==T[0]T[1]T[2]T[3]T[4]T[5]

主串回溯到S[5]如果能匹配，则:（反之亦反）
S[5]S[6]S[7]S[8]S[9]==T[1]T[2]T[3]T[4]T[5]
==T[0]T[1]T[2]T[3]T[4]
且
S[10]==T[5]

KMP算法

- 基本观察

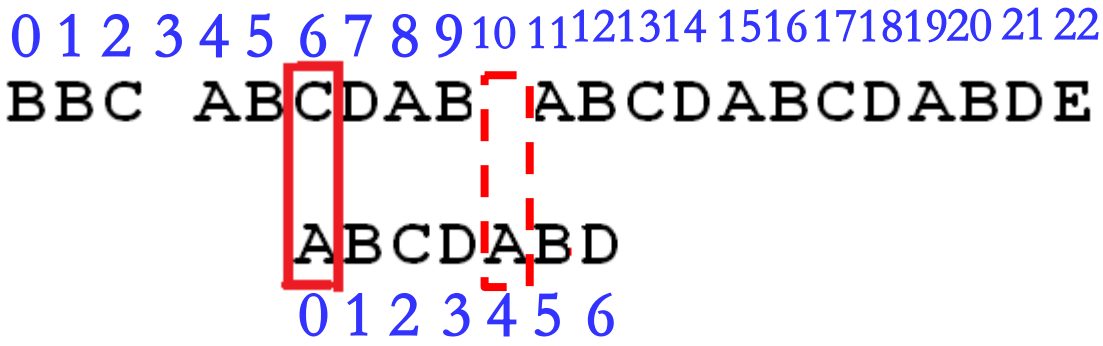


S[10]与T[6]失配（注意，模式串失配位置为6），意味着:
S[4]S[5]S[6]S[7]S[8]S[9]==T[0]T[1]T[2]T[3]T[4]T[5]

主串回溯到S[5]如果能匹配，则:（反之亦反）
S[5]S[6]S[7]S[8]S[9]==T[1]T[2]T[3]T[4]T[5]
==T[0]T[1]T[2]T[3]T[4]
且
S[10]==T[5]

KMP算法

- 基本观察



主串回溯到S[6]如果能匹配，则:（反之亦反）

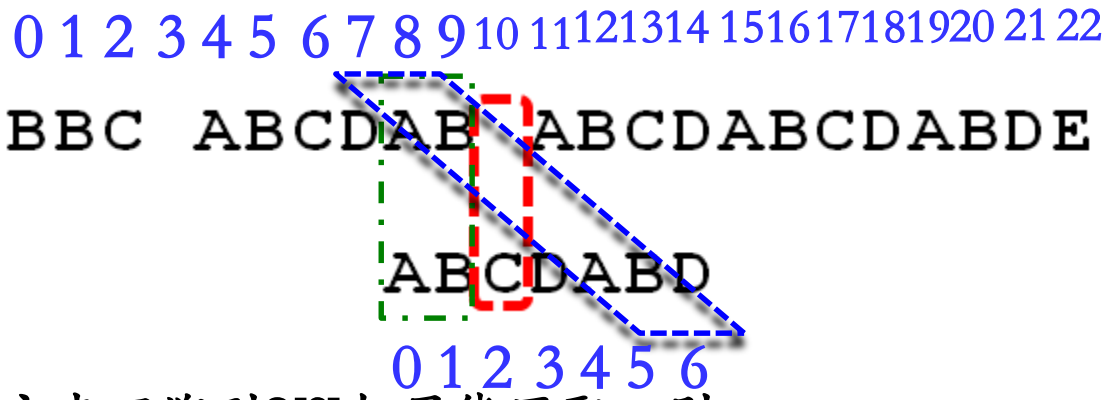
S[6]S[7]S[8]S[9]==T[2]T[3]T[4]T[5]

==T[0]T[1]T[2]T[3]

且

S[10]==T[4]

KMP算法

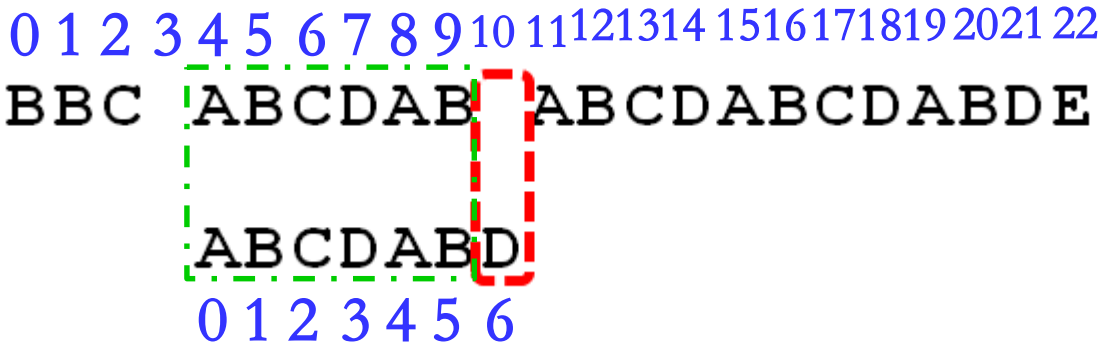


主串回溯到S[8]如果能匹配，则:

S[8]S[9]==T[4]T[5]==T[0]T[1] 且 S[10]==T[2]

- Intuition: 主串不需要回溯，只需要模式串回溯
- Question: 模式串回溯到哪里?
  - 仿佛与模式串失配位置有关
  - 回溯位置与失配之前模式串字串的前缀、后缀有关

KMP算法



- Question: 模式串回溯到哪里(失配位置为6)？
  - 由于T[1]T[2]T[3]T[4]T[5] != T[0]T[1]T[2]T[3]T[4]，模式串向右滑动一位，即模式串指针回退到T[5]，必然匹配失败
  - 由于T[2]T[3]T[4]T[5] != T[0]T[1]T[2]T[3]，模式串向右滑动两位，即模式串指针回退到T[4]必然失败
  - ……
  - 若T[4]T[5]==T[0]T[1], 模式串向右滑动四位，即模式串指针回退到T[2]，必然有S[8]S[9]与T[0]T[1]匹配
  - 模式串回退步长取决于其自身匹配和失配位置:
    - 若T[0]……T[k-1]==T[j-k]……T[j-1] 则后退j-k位
    - 若后退到S[0]都失配，主串指针应该+1

KMP算法用next数组保存部分匹配信息的演示

next[j]是指t[j]字符前有多少个字符与t开头的字符相同。

模式串t存在某个k（0<k<j），使得以下成立：

$$\text{“}t_0t_1\cdots t_{k-1}\text{”} = \text{“}t_{j-k}t_{j-k+1}\cdots t_{j-1}\text{”}$$

开头的k个字符　　t[j]前面的k个字符

0 1 2 3 4

例如，t= “a b a b c” 考虑t[4]='c'

有$t_0t_1 = t_2t_3$ = "ab" ⇨ k=2

所以next[4] = k = 2。

归纳起来，定义next[j]数组如下：

开头的k个字符　　后面的k个字符

$$next[j]=\begin{cases} MAX\{\ k \mid 0<k<j，且“t_0t_1\cdots t_{k-1}”=“t_{j-k}t_{j-k+1}\cdots t_{j-1}”\ \} & 当此集合非空时 \\ -1 & 当j=0时 \\ 0 & 其他情况 \end{cases}$$

t=“aaab”对应的next数组如下：

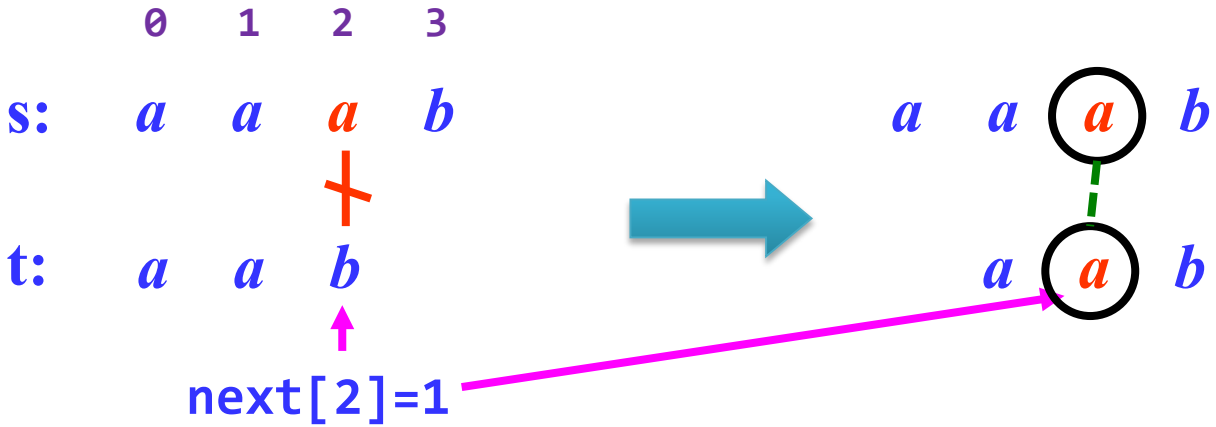
j	0	1	2	3
t[j]	a	a	a	b
next[j]	-1	0	1	2

$t_0=t_1=$"a"　$t_0t_1=t_1t_2=$"aa"

next[j]的含义

（1）next[j]=k表示什么信息？

说明模式串t[j]之前有k个字符已成功匹配，下一趟应从t[k]开始匹配。



（2）next[j]=-1表示什么信息？

说明模式串t[j]之前没有任何用于加速匹配的信息，下一趟应从t的开头即j++ ⇨ j=0开始匹配。

如t=“abcd”，next[0]=-1，next[1]=next[2]=next[3]=0。

KMP算法

- 建立NEXT数组 → 记录模式串的前缀后缀匹配

模式串t=“*ABCDABD*”，next数组的“部分匹配”信息

j	0	1	2	3	4	5	6
t[*j*]	*A*	*B*	*C*	*D*	*A*	*B*	*D*
next[*j*]	-1	0	?	?	?	?	?

KMP算法

- 建立NEXT数组 → 记录模式串的前缀后缀匹配

模式串t=“*ABCDABD*”，next数组的“部分匹配”信息

j	0	1	2	3	4	5	6
t[*j*]	*A*	*B*	*C*	*D*	*A*	*B*	*D*
next[*j*]	-1	0	0	0	0	1	2

模式串t=“*abab*”，next数组的“部分匹配”信息

j	0	1	2	3
t[*j*]	*a*	*b*	*a*	*b*
next[*j*]	-1	0	0	1

KMP算法

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

```
BBC  ABCDAB  ABCDABCDABDE

ABCDABD
```

S[0]与T[0]失配, NEXT[0]=-1, 主串与子串指针右移1位
继续比较S[1]与T[0]

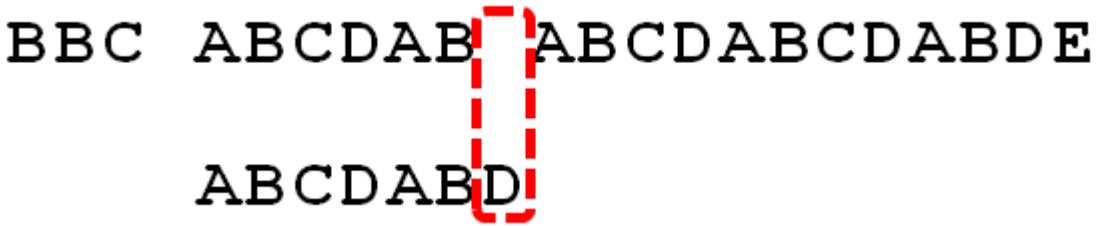
```
BBC  ABCDAB  ABCDABCDABDE

     ABCDABD
```

主串、模式串字符匹配，两个指针均右移

建立NEXT数组 → 记录模式串的前缀后缀匹配

KMP算法

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

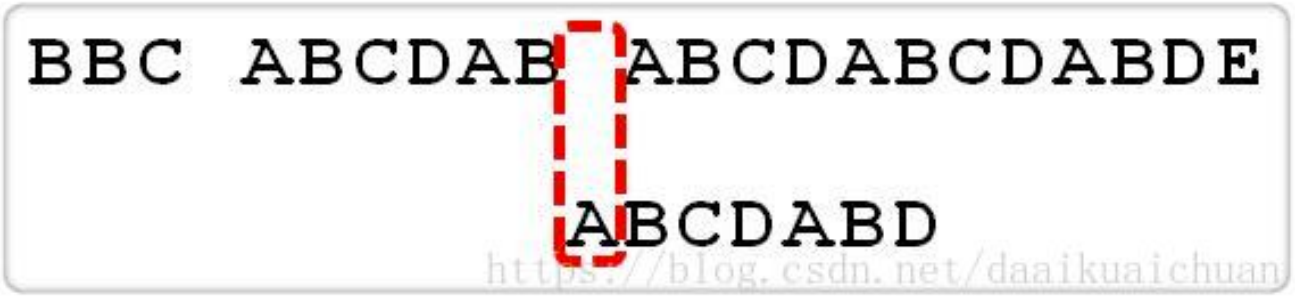


S[10]!=T[6], 因NEXT[6]==2, 指针回退4位, 比较S[10]与T[2]

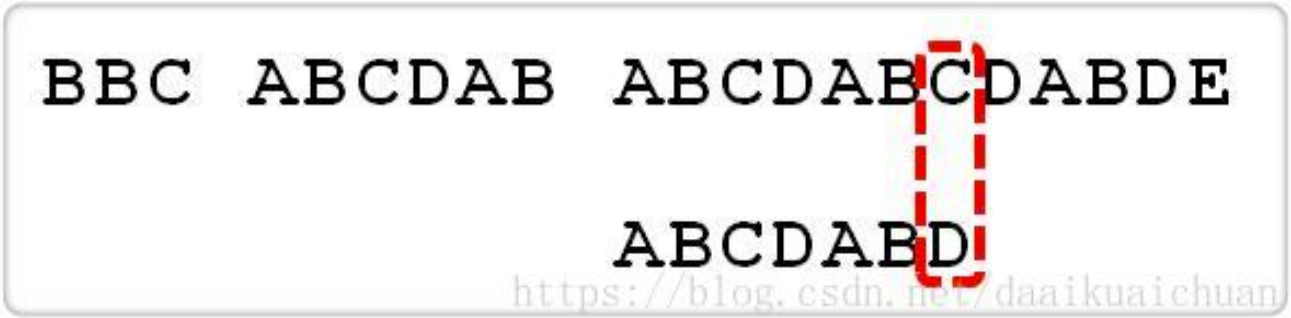
S[10]!=T[2], 因NEXT[2]==0, 指针回退2位, NEXT[0]=-1, 主串与模式串指针均右移

KMP算法

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2



空格与A不匹配，此处next值为-1，表示模式串的第一个字符就不匹配，那么直接往后移一位。



C与D不匹配，下一步从下标为2的地方开始匹配

KMP算法

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

匹配成功

如何代码实现模式串t的next数组求解？

模式串t存在某个k（$0<k<j$），使得以下成立：

$$\text{“}t_0t_1\cdots t_{k-1}\text{”} = \text{“}t_{j-k}t_{j-k+1}\cdots t_{j-1}\text{”}$$

开头的k个字符　　t[j]前面的k个字符

时间复杂度？$O(m^2)$？

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;
      }
      else  k=next[k];
   }
}
```

分支1

分支2

Q1: 代码如何运行的？

Q2: 时间复杂度？ $O(m^2)$？

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

$j = 0 \rightarrow j = 1, k = 0, next[1] = 0$ (分支1)

$j = 1 \rightarrow k = next[0] = -1$ (分支2)

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

$j = 1 \rightarrow j = 2, k = 0, next[2] = 0$ (分支1)

$j = 2 \rightarrow k = next[0] = -1$ (分支2)

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

$j = 2 \rightarrow \quad j = 3, k = 0, next[3] = 0$ **(分支1)**

$j = 3 \rightarrow \quad k = next[0] = -1$ **(分支2)**

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

$j = 3 \rightarrow \quad j = 4, k = 0, next[4] = 0$ **(分支1)**

$j = 4 \rightarrow \quad j = 5, k = 1, next[5] = 1$ **(分支1)**

由模式串t求next值的算法：

```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

$j = 5 \rightarrow j = 6, k = 2, next[6] = 2$ (分支1)

由模式串t求next值的算法：

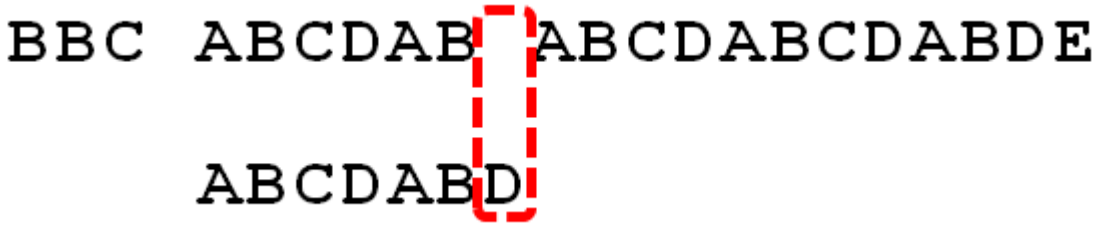
```
void GetNext(SqString t, int next[])
{  int j, k;
   j=0;  k=-1;  next[0]=-1;

   while (j<t.length-1)
   {  if (k==-1 || t.data[j]==t.data[k])
      {  j++; k++;
         next[j]=k;          分支1
      }
      else  k=next[k];       分支2
   }
}
```

else分支k回退的次数不大于if分支语句k增加的次数，而k增加次数等于j增加次数→最坏的可能性是if分支执行m次、else执行m次 → O(m)复杂度

KMP算法：

```
int KMPIndex(SqString s, SqString t)
{  int next[MaxSize],  i=0,  j=0;
   GetNext(t, next);
   while (i<s.length && j<t.length)
   {
       if (j==-1 || s.data[i]==t.data[j])
       {   i++;
           j++;                  //i、j各增1
       }
       else  j=next[j];          //i不变，j后退
    }
    if (j>=t.length)
        return(i-t.length);      //返回匹配模式串的首字符
    else
        return(-1);              //返回不匹配标志
}
```

```
while (i<s.length && j<t.length)
   {
      if (j==-1 || s.data[i]==t.data[j])
(分支1) {   i++;
            j++;                  //i、j各增1
         }
(分支2) else  j=next[j];          //i不变，j后退
     }
}
```



i = 10, j = 6 (分支2)

→ i = 10, j = 2

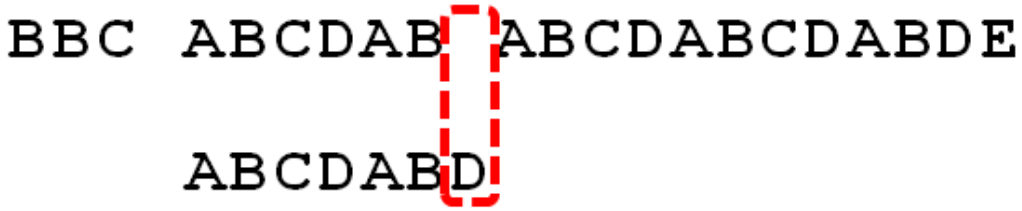
i = 10, j = 2 (分支2)

→ i = 10, j = -1

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

KMP算法

- 算法复杂度分析
  - 主串、模式串匹配的时间复杂度简单分析



- i和j同步增加，或i保持不变，j回溯
- j回溯至-1后，必定开始增加
- 如果某次失配发生在S[i+p]与T[p]处，
  - 模式串回溯次数<=主串增加次数p
  - 最坏情况：主串指针增加p，模式串回溯p次（每次回溯1步），需作n次比较
  - 最好情况：主串指针增加p，模式串回溯1次（回溯p步），需作一次比较
- 至多2n次基本操作

KMP算法

- 算法复杂度分析（串s的长度为n，串t长度为m）
  - 模式串自身匹配的时间复杂度
    - O(m)
  - 模式串自身匹配的空间复杂度
    - O(m)
  - 主串、模式串匹配的时间复杂度
    - O(n)
  - 总复杂度O(m+n)
  - 匹配过程中的比较次数之多为2n-1

【例】 已知字符串S为“*abaabaabacacaabaabcc*”，模式串t为“*abaabc*”，采用KMP算法进行匹配，第一次出现“失配”(s[*i*] != t[*j*])时，*i*=*j*=5，则下次开始匹配时，*i*和*j*的值分别是（ ）。

A.*i*=1，*j*=0　　B.*i*=5，*j*=0　　C.*i*=5，*j*=2　　D.*i*=6，*j*=2

j	0	1	2	3	4	5
t[*j*]	*a*	*b*	*a*	*a*	*b*	*c*
next[*j*]	-1	0	0	1	1	2

选C

思考题

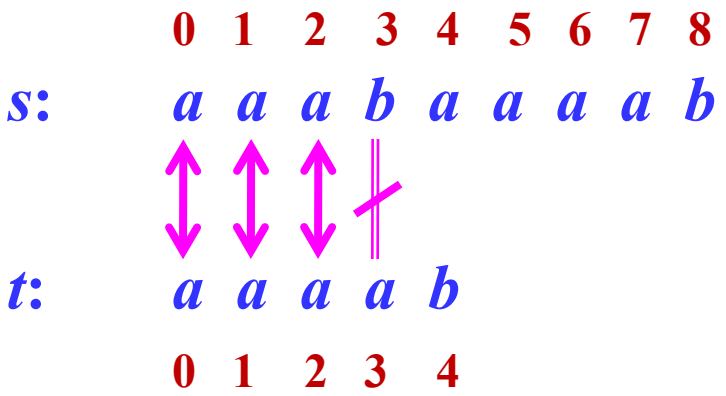
上述KMP算法仍然存在什么缺陷？

设目标串s="*aaabaaaab*"，模式串t="*aaaab*"。KMP模式匹配。

求t的next:

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3

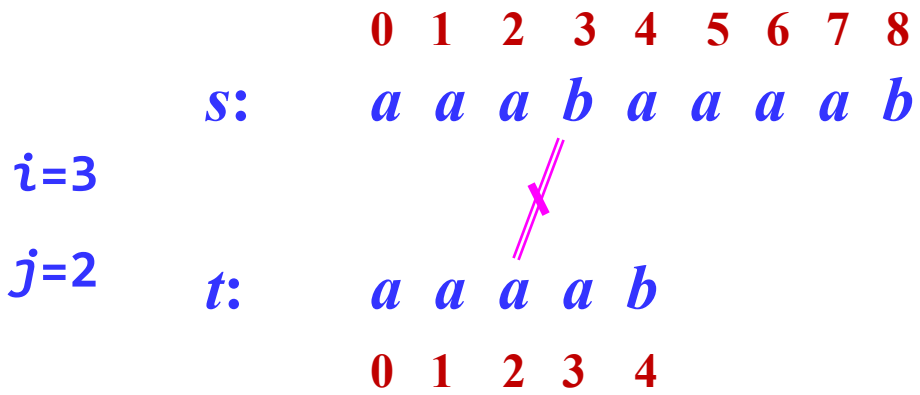


失败：

i=3

j=3, j=next[3]=2

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3

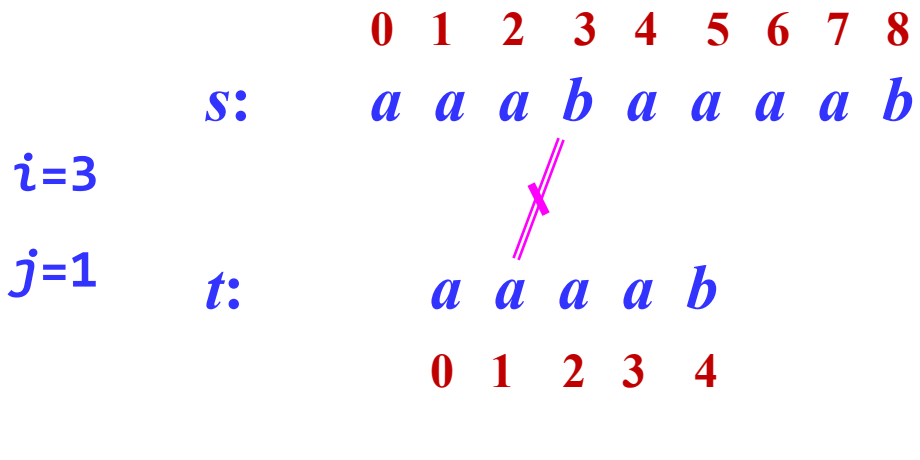


失败：

i=3

j=2，j=next[2]=1

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3

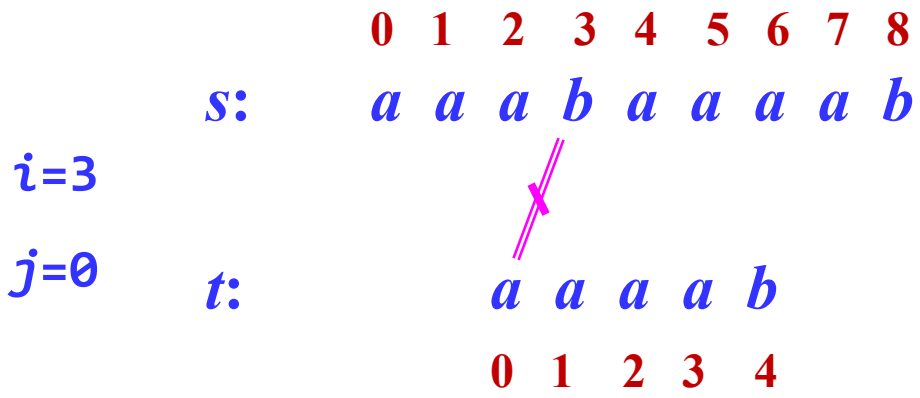


失败：

i=3

j=1, j=next[1]=0

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3



失败：

i=3

j=0,

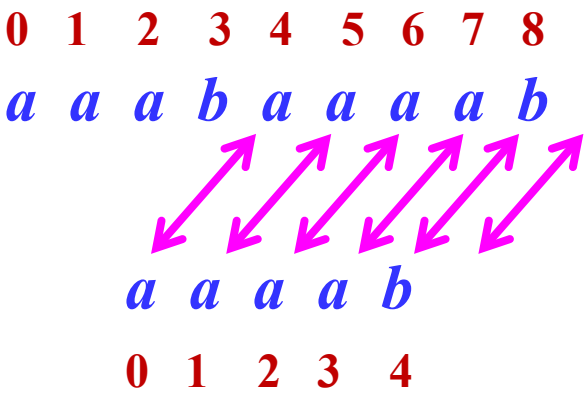
j=next[0]=-1

j	0	1	2	3	4
t[j]	a	a	a	a	b
next[j]	-1	0	1	2	3

因为j=-1:

i++;

j++;

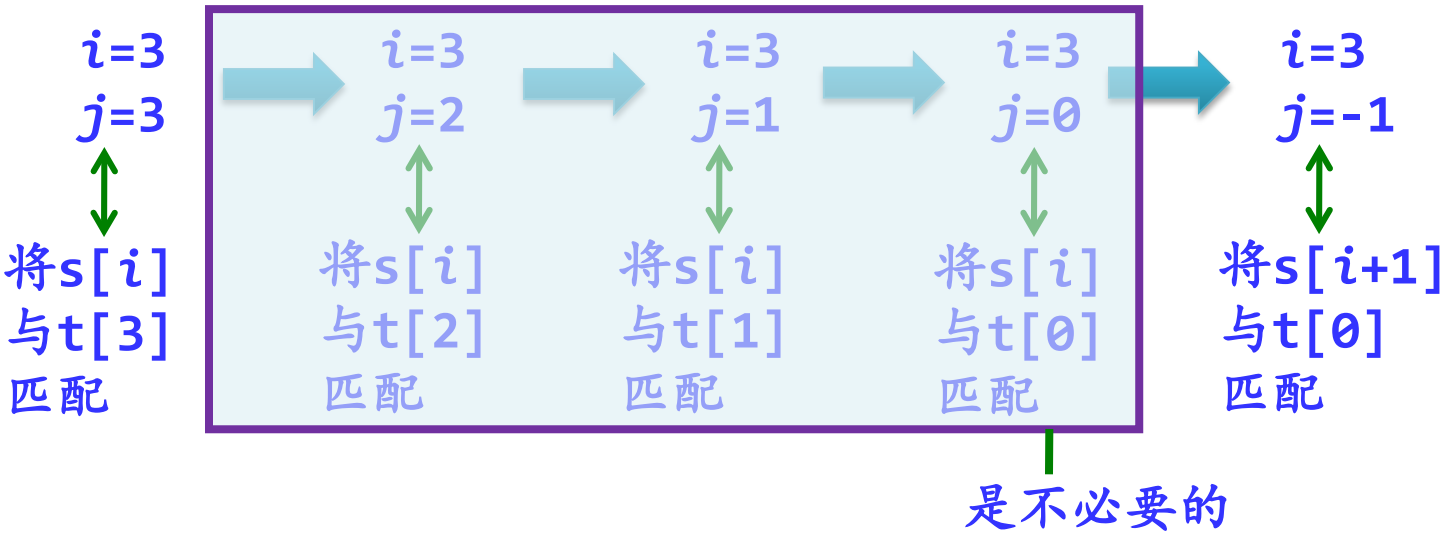


成功:

返回4

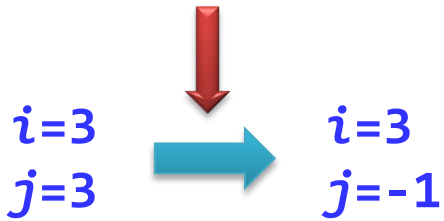
j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3

前面的匹配过程：



因为

t[3]=t[2]=t[1]=t[0]='a'



将next改为nextval:

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

next[1]=0
t[1]=t[next[1]]=t[0]='*a*'
∴ nextval[1]=nextval[0]=-1

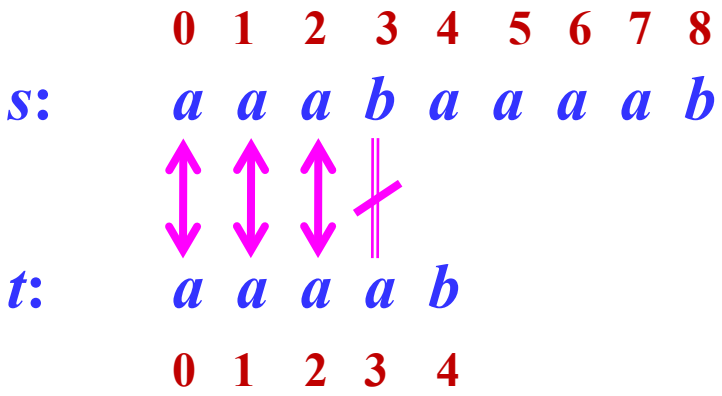
t[4]='*b*' ≠ t[next[4]]='*a*'
∴ nextval[4]=next[4]

- nextval[0]=-1
- 当t[*j*]=t[next[*j*]]时: nextval[*j*]=nextval[next[*j*]]
- 否则: nextval[*j*]=next[*j*]

用nextval取代next，得到改进的KMP算法。

使用改进后的KMP算法示例：

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
nextval[j]	-1	-1	-1	-1	3



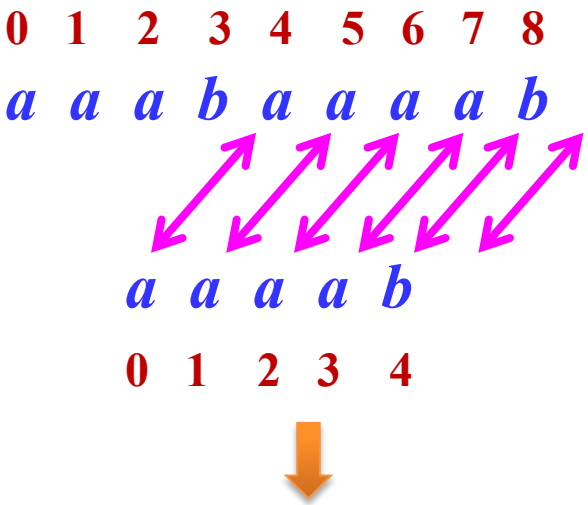
失败：

i=3

j=3, j=nextval[3]=-1

j	0	1	2	3	4
t[j]	*a*	*a*	*a*	*a*	*b*
nextval[j]	-1	-1	-1	-1	3

因为j=-1:
i++;
j++;



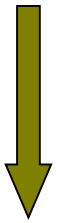
成功:

返回4

改进后的KMP算法进一步提高模式匹配的效率。

数据结构经典算法的启示

BF算法



利用模式串中部分匹配信息

KMP算法

第4章 串

4.4 有限状态机

String Matching with Finite Automata

有限自动机简称 FA（Finite Automata）。FA 是一个有限状态的集合，还有一些从一个状态通向另一个状态的边，每条边上有一个符号，期中一个状态是初态，某些状态是终态，是一种状态转移图。形式上，FA 是一个五元组（S、Σ、δ(s, c)、S(0)、S(A)），其中各个分量表示如下：

- S：是 FA 中的有限状态集合，包含错误状态 S(E)，通常 S(E) = S(0)。
- Σ：是 FA 中的使用的字母表，通常 Σ 是转移图种边的标签集合。
- δ(s, c)：是 FA 的状态转移函数。它将每个状态 s 和每个字符 c的组合（s, c）映射到下一个状态。在状态 S(i) 遇到输入字符 c，FA 将进行状态转移。
- S(0)：是指定的起始状态，属于 S 的子集。
- S(A)：是接受状态集合，属于 S 的子集。

String Matching with Finite Automata

Example:

$Q = \{0,1,2,3,4,5\}$,

$q_0 = 0$,

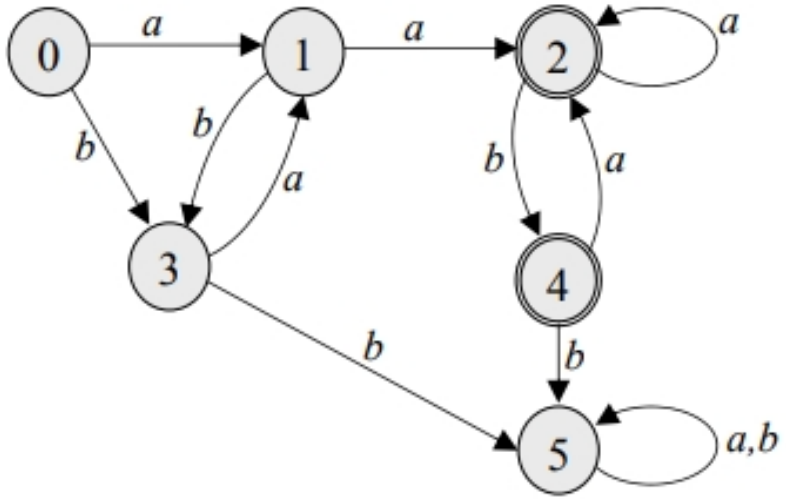
$A = \{2,4\}$,

$\Sigma = \{a,b\}$,

$\delta(q,\sigma)$

q \ σ	a	b
0	1	3
1	2	3
2	2	4
3	1	5
4	2	5
5	5	5

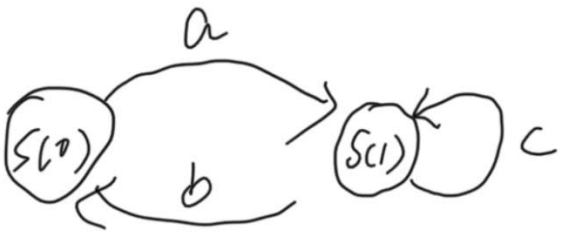
We can represent our FA graphically like this:



State 0 = starting state.

Double line boundary = accepting state

String Matching with Finite Automata



$$\delta = \{ S(0), S(1), S(E) \}$$

$$\Sigma = \{ a, b, c \}$$

$$\delta(S,C) = \{ S(0) \xrightarrow{a} S(1) \quad S(1) \xrightarrow{c} S(1) \quad S(1) \xrightarrow{b} S(0) \}$$

$$S(0) = S(0)$$

$$S(A) = \{ S(1) \}$$

字符串匹配过程

有限自动机开始于状态 S(0)，每次读入字符串中的一个字符 c，状态就会通过 δ(s, c) 进行一次状态转移，每当当前达到的状态 s 属于 S(A) 时，自动机 FA 接受了迄今为止所读入的字符串，同时达到 S(A) 时主串（Text）的下标即匹配完成的结束位置。如果没有被接受的输入则匹配失败。有限自动机字符串匹配主要分为两步：构造字符串匹配自动机、进行主串匹配。

1. 字符串匹配自动机

字符串匹配自动机是针对模式串 P 来构建的，类似于 KMP 算法构造 Next 数组。

假设模式串 P 的长度为 m，则有限自动机对应的分量分别设定为：

状态集合 S = {0, 1, 2, …, m-1, m}
字母表 Σ 为 P 包含的所有字符集合
起始状态 S(0) = 0
接受状态 S(A) = {m}
状态转移函数 δ(s, c) 下面会详细介绍

字符串匹为了定义状态转移函数 δ(s, c)，我们先引入一个后缀函数 φ(t)，φ(t) 是一个从输入文本（Σ* ）到状态集合（S）的映射，φ(t) 表示 t 的后缀同时是 P 前缀的情况下，t 后缀的最长长度。并且认为空字符串 ϵ 是任意字符串的后缀，所以后缀函数是良定义（well-defined）的。

例如：设定模式串 P = ab，则

φ(ϵ) = 0,

φ(a) = 1，t = a 的后缀在模式串 P = ab 的最长前缀是 a，长度为 1

φ(ca) = 1，t = ca 的后缀在模式串 P = ab 的最长前缀是 a，长度为 1

φ(cc) = 0，t = ca 的后缀在模式串 P = ab 的最长前缀是 ϵ，长度为 0

φ(cab) = 2，t = cab 的后缀在模式串 P = ab 的最长前缀是 ab，长度为 2

φ(cabc) = 0，t = cabc 的后缀在模式串 P = ab 的最长前缀是 ϵ，长度为 0

在引入后缀函数 φ(t) 的基础下，对于任意的状态 s 和字符 c，状态转移函数 δ(s, c) 的定义为

$$\delta(s, c) = \phi(P(s)c)$$

- P(s) 表示模式串 P 在状态 s 时的前缀
- P(s)c 表示在 P(s) 后拼接字符 c。

2. 字符串匹配

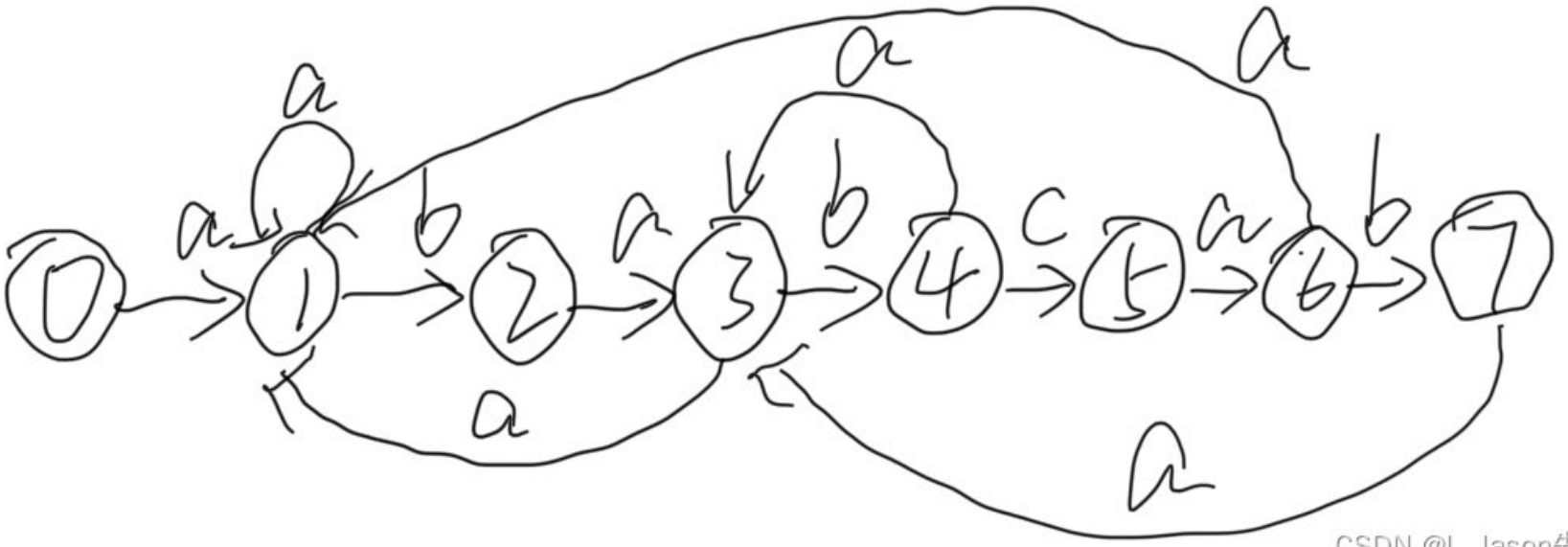
我们通过一个例子来说明对应的状态转移图是什么样子，状态是怎么转移的，以及如何与主串进行匹配：

例：

模式串 P = ababcab

构建状态转移图如下：

- 它可以接受所有以 P 结尾的字符串，初始状态 S(0) = 0，接受状态 S(A) = {7}；
- 向右的边表示匹配模式串 P 与主串输入字符匹配成功；
- 向左的边表示匹配失败后对于模式串的回溯；
- 直接匹配失败回溯到初始状态 S(0) 的边并没有画出来



下表表示转移函数 δ(s, c) 和模式串 P 的对应关系

在「状态」s 的基础上输入 a、b、c 会到达对应表格种的状态，以及匹配成功后对应模式串中的字符：ababcab

状态 >	输入 a	输入 b	输入c	> P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	3	0	5	c
5	6	0	0	a
6	1	7	0	b
7	3	0	0	NA

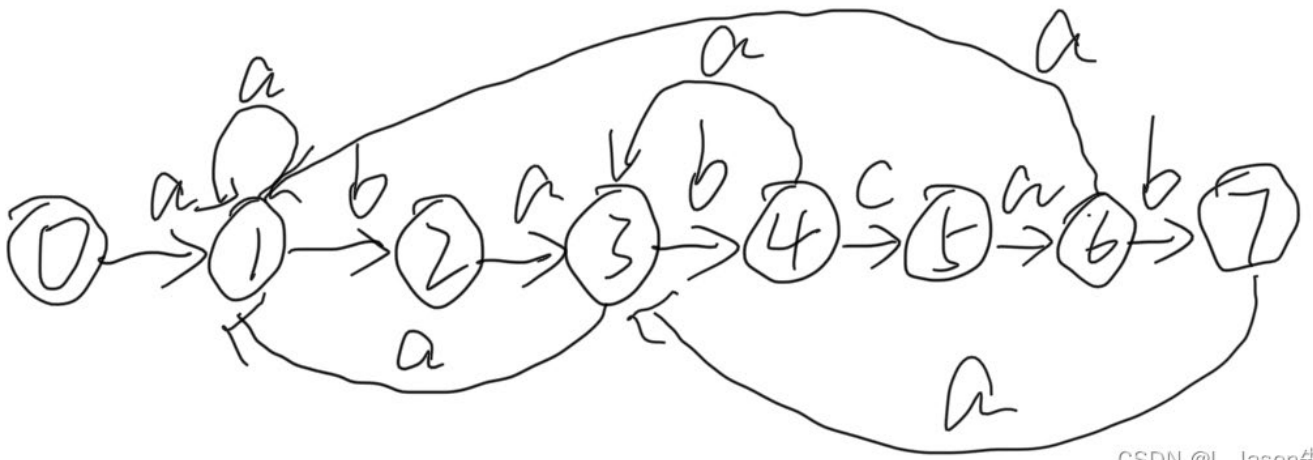
匹配过程

设定 主串 T = abababcabac

主串 T 字符下标从 1 开始，经过 ϕ(P(s)c) 在主串的位置 9 找到一个模式串 P 的完全匹配

结果：成功匹配，匹配位置以 9 结尾

index	-	1	2	3	4	5	6	7	8	9	10	11
T	-	a	b	a	b	a	b	c	a	b	a	c
ϕ(P(s)c)	0	1	2	3	4	3	4	5	6	7	3	0



DFA 有限状态机的构造过程.

1) 物理意义 $\delta(s, c)$

s表示当前状态下已经有s个字符和模式串P的前s个字符是匹配的；下一个待比对的字符是'c'

① 例：

字符集中的字母	P =	0 A	1 B	2 A	3 B	4 A	5 C
	A	1		3		5	
	B		2		4		
	C						6

→ State s

→ 每个表项表示已经有s个字符匹配时，新增加一个字符再与模式串匹配，将会有多少个字符的匹配上.

$\delta(2, A)$：字符串 ***** AB 与模式串P的前缀AB匹配，故状态为2，现在多考虑一个字符'A'，即寻找 ***** ABA 和 P的前缀 ABA 的最长匹配

(ABA：原来的后缀(即状态)；ABA：原来的前缀)

易于理解

对比新遇到的字符与P[2]是否相等，若相等，则有3个字符可以匹配，进入状态3

$\delta(2, A) = 3$.

同理：$\delta(3, B) = 4$，$\delta(4, A) = 5$，$\delta(5, C) = 6$

※若新考察字符与P[s]不同该怎么办？

② 例：在状态5遇到A或B，即如何计算 $\delta(5, A)$ 和 $\delta(5, B)$？

物理意义：P: A B A B A C

Q: * * * * * A B A B A A

下一步如何去考察匹配情况？

P: A B A B A C

Q': * * * * * A B A B A A

该字母必然会被考虑

→ 需要检查红框中的后缀与P的前缀的匹配程度。

→ idea：可否利用已经计算出来的不完整的状态机来判断匹配程度？

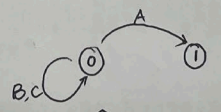
2) 计算和高效计算状态转移矩阵

一步一步地填充 δ 矩阵：

①

	0 A	1 B	2 A	3 B	4 A	5 C
A	1					
B	0					
C	0					

	0 A	1 B	2 A	3 B	4 A	5 C
A	1	1				
B	0	2				
C	0	0				



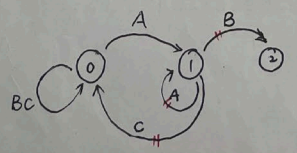
i) 图1

('A'在图1运行结果为1，'C'在图1运行结果为0)

右缀 前缀

A 和 ABABAC 匹配 ⇒ 1

B或C 和 ABABAC 匹配 ⇒ 0



iii) 图2

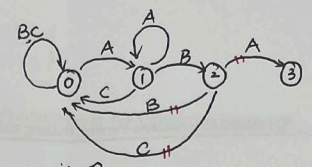
('BB'在图2中运行结果为0，'BC'也是0)

遇到B，状态转移至2

遇到A，由于A≠B=P[1]，让'AA'在图1的状态机运行一趟去掉第一个字母

~~(2) Memory-Efficient Fetching~~

	0	1	2	3	4	5
	A	B	A	B	A	C
A	1	1	3			
B	0	2	0			
C	0	0	0			



iii) 图3

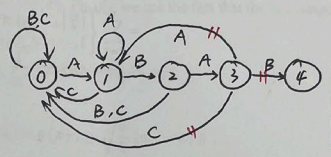
(运行'BAA'得到1，运行'BAC'得到0)

遇到A，状态转移至3

遇到B或C，由于≠P[3]，则扣掉第一个字母外剩余的后缀 ~~'A'~~'BB'，在图2中运行

删掉

	0	1	2	3	4	5
	A	B	A	B	A	C
A	1	1	3	1		
B	0	2	0	4		
C	0	0	0	0		



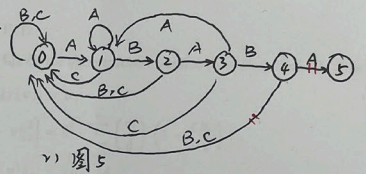
iv) 图4

(运行'BABB'得到0，运行'BABC'也得到0)

遇到B，状态转移至4

遇到A或C，由于不等于P[3]，则扣掉第一个字母，在图3上运行'BAA'和'BAC'

	0	1	2	3	4	5
	A	B	A	B	A	C
A	1	1	3	1	5	
B	0	2	0	4	0	
C	0	0	0	0	0	



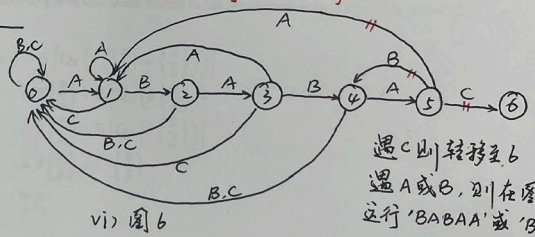
v) 图5

(运行'BABAA'得1，运行'BABAB'得4)

遇到A，状态转移至5

遇到B或C，由于≠P[4]，则在图4上运行'BABB'或'BABC'

	0	1	2	3	4	5
	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



vi) 图6

遇C则转移至6

遇A或B，则在图5上运行'BABAA'或'BABAB'

② 降低复杂度

上述方法复杂度高的根源在于多次在DFA上运行，寻找部分匹配。

例：最后一次运行'BABAA'和'BABAB'，其实不需要重复运行。若知道'BABA'会到达状态3，只需要检查下一个字符在状态3下是A还是B，即为δ(3,A)和δ(3,B)。

倒数第二次运行'BABB'和'BABC'，若知道'BAB'到达状态2，只需要检查δ(2,B)和δ(2,C)

即记录B，BA，BAB，BABA在DFA上运行的结果，可以在构造DFA的过程中构造数组

TEMP = [0 1 2 3]

串的算法案例

- 例1： Implement strStr().

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Input: haystack = "hello", needle = "ll"

Output: 2

Input: haystack = "aaaaa", needle = "bba"

Output: -1

串的算法案例

- 字符串中的第一个唯一字符
  - s = "leetcode" 返回 0.
  - s = "loveleetcode", 返回 2.

    //假定该字符串只包含小写字母。
- 有效的字母异位词
  - 给定两个字符串 *s* 和 *t*，编写一个函数来判断 *t* 是否是 *s* 的一个字母异位词。
  - s = "anagram"，t = "nagaram"，返回 true
  - s = "rat"，t = "car"，返回 false

  //注意：假定字符串只包含小写字母。

串的算法案例

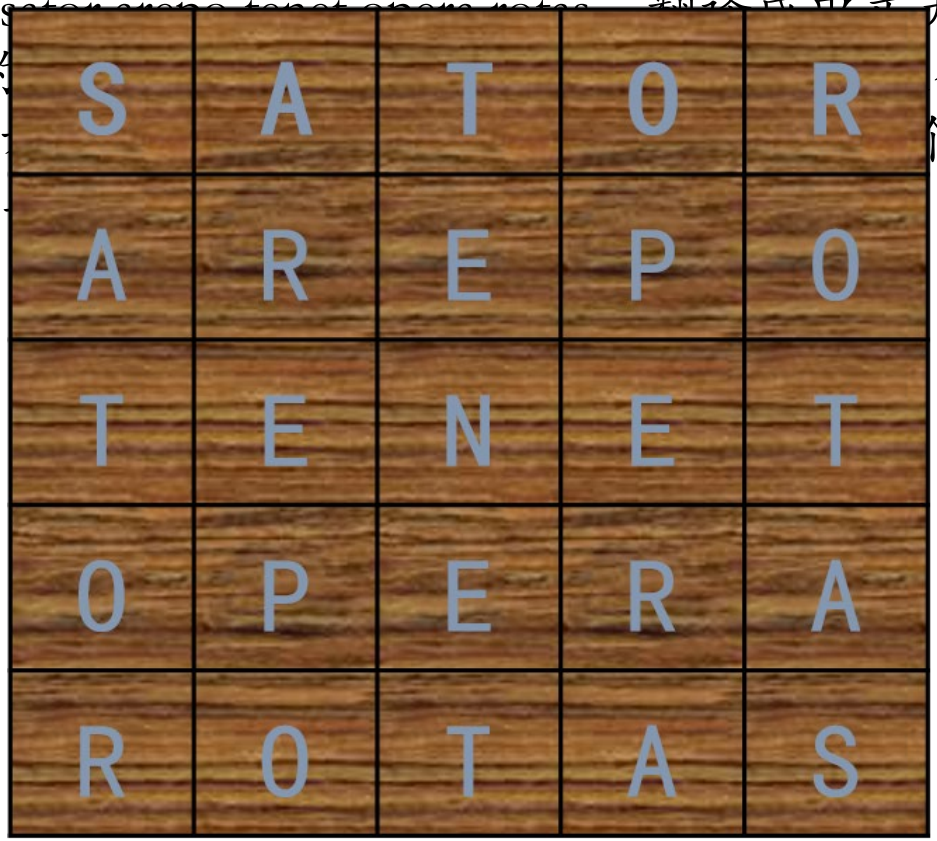
- 验证回文字符串
  - 给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。
  - 输入: "A man, a plan, a canal: Panama" 输出: true
  - 输入: "race a car" 输出: false

串的算法案例

- 最长回文子串
  - 正读和反读都相同的字符序列为“回文”，如“abba”、“abccba”是“回文”，“abcde”和“ababab”则不是“回文”。
  - 字符串的最长回文子串，是指一个字符串中包含的最长的回文子串。例如“1212134”的最长回文子串是“12121”。

串的算法案例

- 人们在一座名为赫库兰尼姆的古城遗迹中，找到了一个好玩的拉丁语回文串：[illegible]翻译成中文大概就是`一个叫做Arepo的[illegible]个串的每个单词首字母刚[illegible]个字母刚好组成了第二个[illegible]



串的算法案例

- 无重复字符的最长子串
  - 给定"abcabcbb"，没有重复字符的最长子串是"abc"，那么长度就是3。
  - 给定"bbbbb"，最长的子串就是"b"，长度是1。
  - 给定"pwwkew"，最长子串是"wke"，长度是3。

小结

- 串的存储结构：顺序串、链串
- 三种不同复杂度的模式匹配算法
  - Brute-force 算法
  - Rabin-Karp 算法
  - KMP 算法

——本章完——