计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：张羽丰

湖南大学 信息科学与工程学院

第 4 章

字符串

提纲

4.1 问题引入：模式匹配

查找功能：文本编辑工具中，给定一段文本，用户提供特定的关键词，找出这个关键词在文本中出现的位置，就是字符串匹配问题。

用户给定的关键词或字符串被称为**模式串**，段落文本成为**目标串**，因此字符串匹配又称为模式匹配。

4.2 字符串的定义与结构

字符串：一种在数据元素的组成上具有一定约束条件的线性表，即要求组成线性表的所有数据元素都是字符。

字符串一般记为s="s1s2...sn"(n>0)或∅，其中s为字符串名，双引号为字符串的定界符，双引号之间的内容是字符串的值，si（1≤i≤n）可以是字母、数字或其他字符，n为字符串长度。∅表示空串，空串的长度为0。

字符串中任意个连续的字符组成的子序列称为该字符串的**子串。**如长度为4的字符串s="abcd"，其子串包括"a"、"ab"、"abc"、"abcd"、"b"、"bc"、"bcd"、"c"、"cd"、"d"、""（空字符串）。

字符串的抽象数据类型定义

ADT String {

数据对象： $D=\{ s_i \mid s_i \in \text{CharacterSet}, i=1, 2, \dots, n, n>0 \}$或$\varnothing$，表示空字符串

数据关系： $R=\{ <s_{i-1}, s_i> \mid s_{i-1}, s_i \in D, i=2, \dots, n, n>0 \}$

基本操作:

InitStr(s)：初始化一个空的字符串s，字符串最大长度为kMaxSize。

StrCopy(s)：返回复制字符串s得到的字符串。

StrIsEmpty(s)：判断字符串s是否是空串。返回一个布尔值，若字符串s是空串，返回true；否则返回false。

StrInsert(s, pos, t)：在字符串s的pos位置处插入字符串t，并返回插入后的字符串s。

StrRemove(s, pos, len)：删除字符串s中从pos位置开始的长度为len的子串，返回删除后的字符串s。

SubString(s, pos, len)：返回字符串s从pos位置开始长度为len的子串。

StrLength(s)：返回字符串s的长度。

StrConcat(s,t)：返回字符串s和t联接而成的新串s。

StrCompare(s,t)：返回字符串s和t的大小关系。若s>t，返回+1；若s=t，返回0；若s<t，返回-1。

PatternMatch(s,t)：返回字符串s中首次出现字符串t的位置，若字符串s没有出现字符串t，则返回NIL。

Replace(s, sub_s, t)：将字符串s中的所有子串sub_s替换为字符串t。}

4.3 字符串的存储实现

字符串存储表示

- 顺序存储结构：将所有的数据元素存放在一段连续的存储空间中，数据元素的存储位置反应了它们之间的逻辑关系
- 链接存储结构：逻辑上相邻的数据元素不需要在物理位置上也相邻，数据元素的存储位置可以是任意的

字符串的顺序存储实现

字符串的顺序存储结构是用一组地址连续的存储单元来存储串中的字符序列，也是最为常见的字符串存储结构。

例：字符串"abcdef"的顺序存储形式

注意：一般在程序设计语言中，字符串会有结束符，如C语言和C++语言中字符串的结束符为'\0'，结束符不计入字符串长度，但要占存储空间。

根据是否预先确定串的存储空间大小，可将字符串的顺序存储结构分为**定长顺序存储**和**动态顺序存储**两种类型

字符串的顺序存储实现

1.字符串插入

在字符串s的pos位置插入字符串t。首先将字符串s从pos位置开始整体后移，为字符串t的插入提供空间，然后将字符串t中的字符逐一插入到字符串s从pos开始的空间中，得到插入后的字符串s，最后更新字符串s的length属性。

时间复杂度：最坏情况是将t插在s的头，时间复杂度是**O(n+m)**，其中n和m分别是两个字符串的长度。

算法4-1 字符串插入操作StrInsert(s, pos, t)

```
输入：字符串s ，要插入的位置pos≥1，需要插入的字符串t
输出：完成插入后的字符串s。若插入后的字符串长度大于kMaxSize，则直接退出
n ← s.length
m ← t.length
if n+m≤kMaxSize then
| for i←n-1 downto pos-1 do
| | s.data[i+m] ← s.data[i]  //将数组下标pos-1开始的子串后移，给t 留出空位
| end
| for i←0 to m-1 do
| | s.data[pos-1+i] ← t.data[i]  //将t 插入s
| end
| s.length ← n + m  //更新s的长度
else
| 长度超限，退出
end
```

字符串的顺序存储实现

2. 字符串删除

将字符串s从pos位置开始删除长度为len的子串。将字符串s中pos+len后面的部分逐位向前移动，最后更新字符串s的length属性。

时间复杂度：最坏情况是将s开头的len个字符删除，时间复杂度是**O(n)**，其中n是s的长度。

算法4-2 字符串删除操作StrRemove(s, pos, len)

```
输入：字符串s，要删除的位置pos≥1，删除的字符个数len
输出：删除子串后的字符串s
n ← s.length
if pos+len-1<n then //如果要删的部分没有到达串尾
| for i ← pos+len-1 to n-1 do
| | s.data[i-len] ← s.data[i]
| end
| s.length ← n – len
else //从数组下标pos-1开始的所有字符都删掉
| s.length ← pos-1
end
```

字符串的顺序存储实现

3. 字符串截取

返回字符串s从pos位置开始长度为len的子串。构造新串，将字符串s从pos之后的len个字符——赋值给新串。

时间复杂度：该操作仅与子串长度有关，为**O(len)**。

算法4-3 字符串截取子串操作SubString(s, pos, len)

```
输入：字符串s，开始截取的位置pos≥1，截取的字符个数len
输出：截取的子串
InitStr(sub_s) //初始化新串sub_s
n ← s.length
i ← 0
while pos-1+i<n 且 i<len do//若s从pos-1开始不到len个字符，就截取到s的末尾为止
| sub_s.data[i] ← s.data[pos-1+i] //将s串从pos-1之后的len个字符复制到sub_s
| sub_s.length ← sub_s.length + 1
| i ← i+1
end
return sub_s
```

字符串的顺序存储实现

4. 字符串连接

将字符串t连接在字符串s末尾。将字符串t中的字符一一赋值到s末尾，并更新s的长度。

时间复杂度：该操作仅与t的长度有关，为**O(m)**。

算法4-4 字符串连接操作StrConcat(s,t)

输入：字符串s，字符串t
输出：返回字符串s后面联接t而成的新串。若结果长度大于kMaxSize, 则退出。

```
n ← s.length
m ← t.length
if n+m ≤ kMaxSize then
|  for i ← 0 to m-1 do
|  |  s.data[n+i] ← t.data[i]
|  end
|  s.length ← n+m
else
|  长度超限，退出
end
```

字符串的顺序存储实现

5. 字符串比较

- 将这两个字符串从左到右逐个字符按照其ASCII码值进行比较。
- 如果两个字符串长度相等，且每一个相应位置上的字符都相同，则这两个字符串相等，如"abc"与"abc"相等。
- 如果两个字符串长度不相等，但较短字符串所有对应位置上的字符都与较长字符串相同，则字符串长度长的字符串大于长度短的字符串。如"abc"<"abcdef"。
- 如果两个字符串在某一相应位置上的字符不相同，则以第一个不相同的位置上的字符比较结果作为两个字符串的比较结果。如"abh">"abf"。

算法4-5对两个字符串做比较，若s>t返回+1；若s=t返回0；若s<t返回-1。

时间复杂度：仅与s和t的最小长度有关，为**O(min(n, m))**

算法4-5 字符串比较操作StrCompare(s,t)

```
输入：字符串s ，字符串t
输出：s>t输出+1；s=t输出0；s<t输出-1
len ← Min(s.length, t.length)
i ← 0
while i<len 且s.data[i] = t.data[i] do
   i ← i+1
end
if i=len then
| if s.length > len then
| | ret ← 1
| else if t.length > len then
| | ret ← -1
| else  //s=t
| | ret ← 0
| end
else if s.data[i] > t.data[i] then
| | ret ← 1
else
| | ret ← -1
end
return ret
```

字符串的链接存储实现

将每个结点存放一个字符的字符串链接存储方式称为**非紧缩链接存储结构**，一个结点存放多个字符的字符串链接存储方式称为**紧缩链接存储结构**，也称为块链存储结构。

例：字符串"abcdef"

非紧缩链接存储结构：

紧缩链接存储结构：

为容易理解，下面对非紧缩链接存储结构的字符串的基本操作进行介绍

字符串的链接存储实现

1. 字符串插入

相对于顺序存储结构，使用链接存储结构的字符串插入操作较为简单，不需要担心字符串长度超限的问题。但需要找到字符串s中位于pos位置的链表元素，并且需要找到插入字符串t的末尾

时间复杂度：该操作时间复杂度是**O(n+m)**，其中n和m分别是两个字符串的长度。

算法4-6 字符串插入操作StrInsert(s, pos, t)

```
输入：字符串s，要插入的位置pos≥1，需要插入的字符串t
输出：完成插入后的字符串s。若不存在位置pos，则s不变
1                flag ← NormalCode
2                if t.length>0 then //若t不是空串
3                | tail ← t.head
4                | while tail.next ≠ NIL do //找到t的最后一个元素
5                | | tail ← tail.next
6                | end
7                | if s.length>0 then //若s不是空串
8                | | p ← s.head
9                | | count ← 1
10               | | while p ≠ NIL 且 count<pos-1 do //找第pos个元素的前一个元素
11               | | | count ← count + 1
12               | | | p ← p.next
13               | | end
14               | | if count=(pos-1) then //将t插在p的后面
15               | | | tail.next ← p.next
16               | | | p.next ← t.head
17               | | else if pos = 1 then //t插在s的头
18               | | | tail.next ← s.head
19               | | | s.head ← t.head
20               | | else
21               | | | flag ← ErrorCode //输入错误：不存在位置pos
22               | | end
23               | else //若s是空串
24               | | s ← t
25               | end
26               end
27               if flag ≠ ErrorCode then  //正常完成插入
28               | s.length ← s.length + t.length
29               end
```

字符串的链接存储实现

2. 字符串删除

返回字符串s从pos位置开始删除长度为len的子串后的字符串。将pos-1位置上的链表元素的next指针指向原字符串s中的第pos+len个元素，并释放被删除的结点空间。

时间复杂度：这个操作的时间复杂度与s的长度有关，为**O(n)**

算法4-7 字符串删除操作StrRemove(s, pos, len)

输入：字符串s，要删除的位置pos≥1，删除的字符个数len
输出：删除子串后的字符串s。若删除位置不存在，则s不变。

```
1.  if s.length>0 then //若s不是空串
2.  | p ← s.head
3.  | count ← 1
4.  | while p ≠ NIL 且 count<pos-1 do //找第pos个元素的前一个元素
5.  | | count ← count + 1
6.  | | p ← p.next
7.  | end
8.  | if pos=1 或 (p ≠ NIL 且count=pos-1) then //将p后面的len个结点删除
9.  | | if pos=1 then
10. | | | deleted ← s.head
11. | | else
12. | | | deleted ← p.next
13. | | end
14. | | count ← 0
15. | | while deleted≠ NIL 且 count < len do //不足len个则一直删到末尾
16. | | | t ← deleted.next
17. | | | delete deleted
18. | | | count ← count + 1
19. | | | s.length ← s.length - 1
20. | | | deleted ← t
21. | | end
22. | | if pos = 1 then
23. | | | s.head ← deleted
24. | | else
25. | | | p.next ← deleted
26. | | end
27. | end
28. end
```

字符串的链接存储实现

3. 字符串截取

返回字符串s从pos位置开始的长度为len的子串。构造新串，将字符串s从pos之后len个字符一一赋值给新串。

时间复杂度：与顺序存储不同，这个操作必须首先找到截取的起始位置，时间复杂度就不仅与子串长度有关了，最坏时间复杂度为O(n)。

算法4-8 字符串截取子串操作SubString(s, pos, len)

```
输入：字符串s，开始截取的位置pos≥1，截取的字符个数len
输出：截取的子串
1.  InitStr(sub_s) //初始化新串sub_s
2.  if s.length>0 then //若s不是空串
3.  | p ← s.head
4.  | count ← 1
5.  | while p ≠ NIL 且 count<pos do //找第pos个元素
6.  | | count ← count + 1
7.  | | p ← p.next
8.  | end
9.  | if p ≠ NIL 且count=pos then //将s串从pos之后的len个字符复制到sub_s
10. | | count ← 0
11. | | sub_s.head ← new StringNode() //创建临时空头结点
12. | | tail ← sub_s.head
13. | | while p ≠ NIL 且 count<len do //若从pos开始不到len个字符，就截取到s的末尾
14. | | | t ← new StringNode(p.data, NIL) //复制一个新结点
15. | | | tail.next ← t //将新结点接到sub_s的末尾
16. | | | tail ← tail.next
17. | | | sub_s.length ← sub_s.length + 1 //sub_s 长度加1
18. | | | p ← p.next
19. | | | count ← count + 1
20. | | end
21. | end
22. end
23. tail ← sub_s.head
24. sub_s.head ← tail.next
25. delete tail //删除临时空头结点
26. return sub_s
```

字符串的链接存储实现

4. 字符串连接

将字符串t连接在字符串s末尾。与顺序存储不同，这里不需要将字符串t中的字符复制到字符串s末尾，只需要找到字符串s末尾并将字符串t接在后面即可。

时间复杂度：这个操作的时间复杂度与字符串t的长度无关，仅与字符串s的长度成正比，为**O(n)**。

算法4-9 字符串连接操作StrConcat(s,t)

```
输入：字符串s ， 字符串t
输出：返回后面联接t而成的字符串s
1.          if s.length > 0 then //若s非空串
2.          | p ← s.head
3.          | while p.next ≠ NIL do  //找到s的最后一个结点
4.          | | p ← p.next
5.          | end
6.          | p.next ← t.head
7.          else  //若s是空串
8.          | s.head ← t.head
9.          end
10.         s.length ← s.length + t.length
```

字符串的链接存储实现

5. 字符串比较

返回字符串s和字符串t的大小关系。若s>t返回+1；若s=t返回0；若s<t返回-1。与顺序存储相似，

时间复杂度：这个操作的时间复杂度也是仅与s和t的最小长度有关，为**O(min(n, m))**

算法4-10 字符串比较操作StrCompare(s,t)

```
输入：字符串s，字符串t
输出：s>t输出+1；s=t输出0；s<t输出-1
1.      sp ← s.head
2.      tp ← t.head
3.      while sp≠NIL 且 tp≠NIL且 sp.data = tp.data do
4.      | sp ← sp.next
5.      | p ← tp.next
6.      end
7.      if sp≠NIL且tp=NILthen
8.      | ret ← 1
9.      else if sp=NIL且tp≠NIL then
10.     | ret ← -1
11.     else if sp=NIL且tp=NIL then //s=t
12.     | ret ← 0
13.     else if sp.data > tp.data then
14.     | | ret ← 1
15.     else //sp.data < tp.data
16.     | | ret ← -1
17.     end
18.     return ret
```

4.4 字符串的模式匹配

概念：在字符串s中找出与字符串t相等的子串的操作称为字符串的模式匹配，又称为子串的定位操作。其中字符串s称为主串或目标串，字符串t称为模式串。

模式匹配算法：朴素字符串匹配算法（BF算法）、KMP算法、BM算法、KR算法、Sunday算法等。

形式化描述：假设目标串s使用一个长度为n的字符数组s[0,1,...,n-1]表示，模式串t使用一个长度为m (m≤ n)的数组t[0,1,...,m-1]表示，如果存在p (0 ≤ p ≤ n - m)，使得s[p+0, p+1, ... , p+m-1] = t[0, 1, ..., m-1]，则p被称为一个有效位移。字符串匹配就是从字符串s中找出存在的有效位移p。

朴素模式匹配算法

朴素模式匹配算法是字符串模式匹配算法中最简单的暴力解法，又称为BF（Brute Force）算法。

实现：枚举每个目标串s与模式串t等长的子串，判断是否匹配

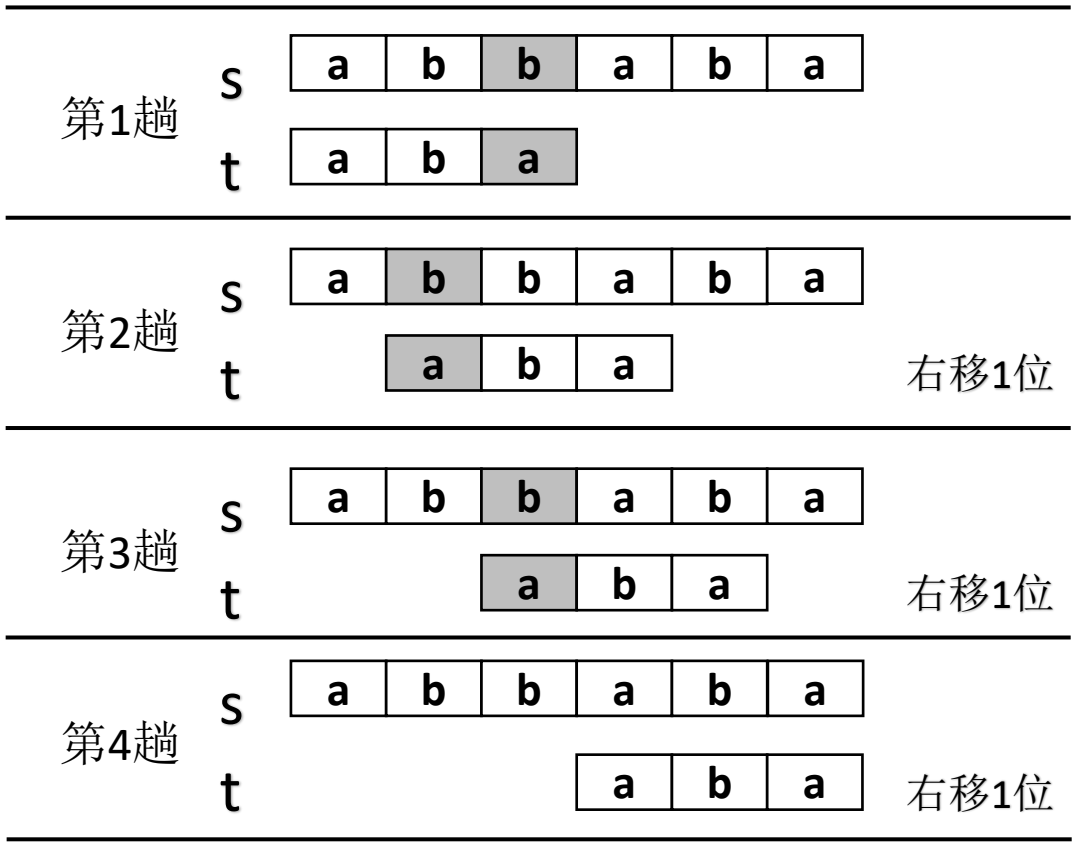
即：将模式串t的第0位字符与目标串s的第0位字符对齐，然后依次比对每个字符，若都相等，则匹配成功；若s和t对应位置上的字符不相等，则将t整体后移1位重新从模式串t的第0位开始依次比对。

算法4-11 朴素字符串匹配算法PatternMatchBF(s, t)

```
输入：目标串s与模式串t
输出：返回首个有效位移p，匹配失败则返回NIL
n ← s.length
m ←t.length
for p←0 to n-m do
| for i← 0 to m-1 do
| | if s.data[p+i]≠t.data[i] then
| | | break
| | end
| end
| if i=m then
| | break
| end
end
if p>n-m then
| p ← NIL
end
return p
```

朴素模式匹配算法

例：目标串s="abbaba"，模式串t="aba"



直观上比较简单

实现层面上：什么是模式串右移?

朴素模式匹配算法

朴素模式匹配算法时间复杂度分析：

最好情况下，仅需匹配m次，此种情况时间复杂度为O(m)

最坏情况下，需要移动n-m+1次，每次匹配m次，时间复杂度为$O(nm-m^2)$，即O(nm)

在实际运行过程中字符串匹配情况复杂多变，朴素模式匹配算法的执行时间通常取上界**O(nm)**。

朴素模式匹配算法

字符串匹配算法是其它部分字符串操作的基础，例如字符串替换操作需要先使用匹配操作找到位置，接着进行字符串的删除和插入替换，算法如下：

算法 4-12 字符串替换算法Replace(s, sub_s, t)

```
输入：字符串s，被替换的子串sub_s，替换目标字符串t
输出：替换字符串s中的所有子串sub_s为字符串t后的字符串s
1    len ← sub_s.length
2    m ← t.length
3    pos ← 0
4    while pos ≠ NIL do
5    |  pos ← PatternMatchBF(s, sub_s)  //从s中找到第一次出现的sub_s
6    |  if pos ≠ NIL then
7    |  |  StrRemove(s, pos+1, len) //删除sub_s
8    |  |  StrInsert(s, pos+1, t)  //插入t
9    |  |  s.length ← s.length - len + m
10   |  end
11   end
```

KMP算法

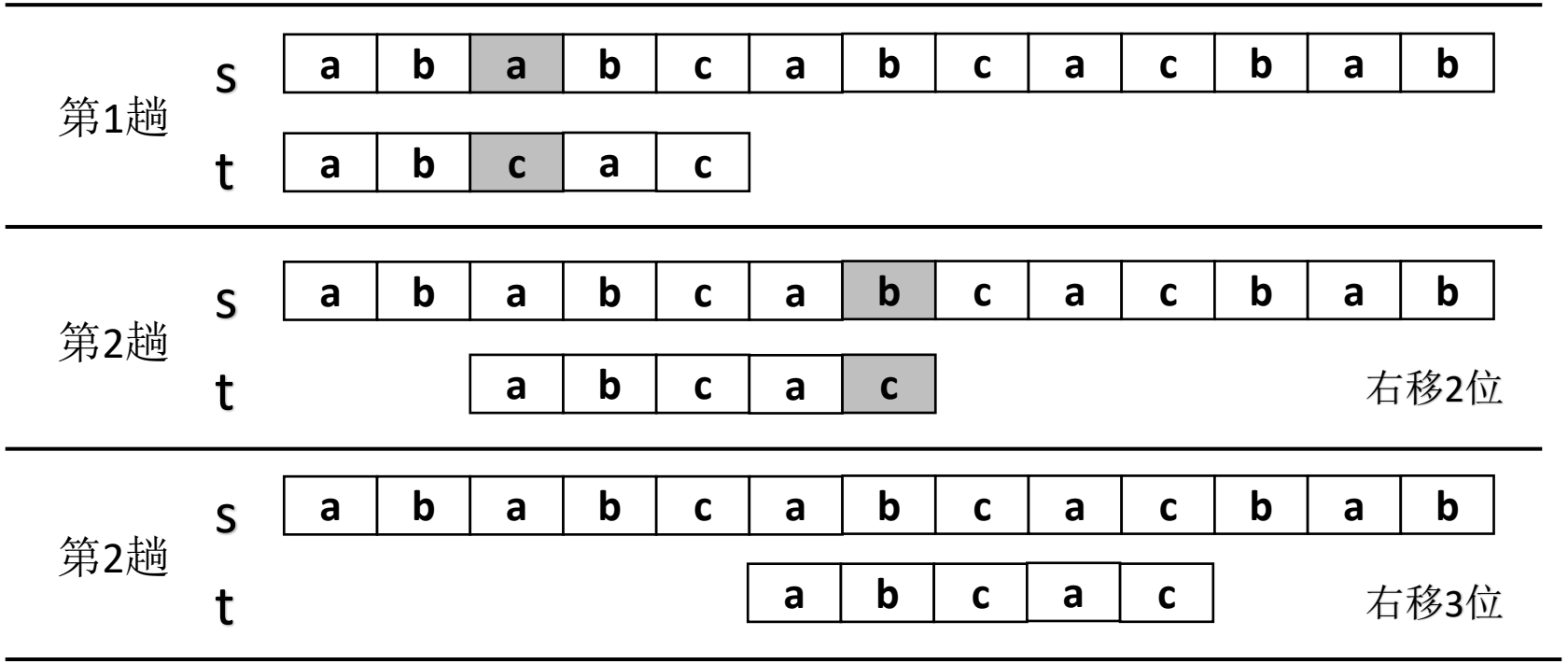
BF算法缺点：当模式串失配时将其整体向后移动一位，然后从头开始匹配，将目标串中每个位置作为起点与模式串中的字符进行逐个比较会耗费较长时间。

提出：KMP算法由D.E.Knuth，J.H.Morris和V.R.Pratt提出的，因此也称为克努特—莫里斯—普拉特算法。

主要思想：假设BF算法在失配时已经匹配到了字符串的第 j 位，则说明目标串与模式串的前 j-1 位是匹配成功的，利用已匹配的信息可对BF算法进行优化。

KMP算法

例：目标串s="ababcabcacbab"，模式串t="abcac “



可以看出在比较的时候，模式串向后移动了更多的位数，因此能够更快地完成模式匹配

KMP算法

思考：当出现失配情况时，如何将模式串向右移动正确的位数?

字符串相关概念：

前缀：从长度为n的字符串第0位开始，第i位（0≤i<n-1）结束的任意子串。对字符串s，其前缀可表示为s[0...i](0≤**i<n-1**)。字符串的所有前缀构成的集合称为前缀集合。

后缀：从长度为n的字符串第i位（0<i ≤n-1）开始，最后一位结束的任意子串。对字符串s，其后缀可表示为s[i...n-1](**0<i**≤n-1)。字符串的所有后缀构成的集合称为后缀集合。

公共前后缀：字符串的前缀集合与后缀集合中相同的子串。

最长公共前后缀：字符串的前缀集合与后缀集合中相同的长度最长的子串。

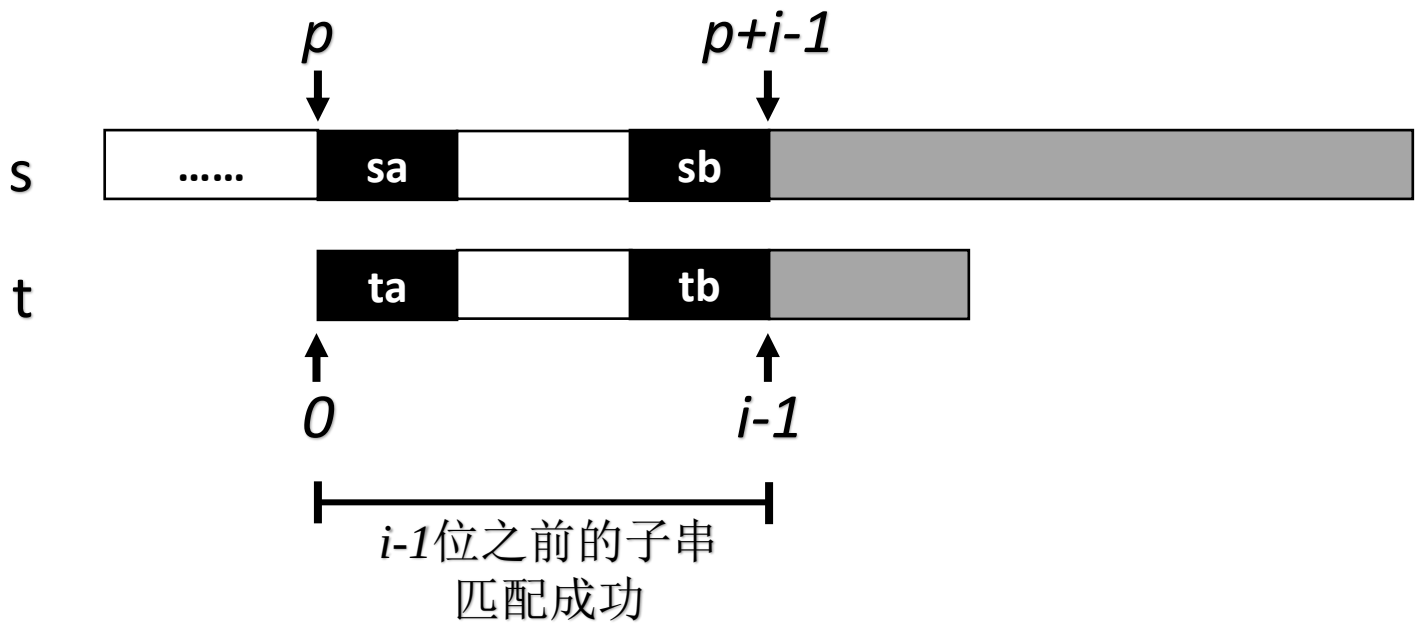
***约定**：前缀、后缀集合不包含自身

例：字符串"aabaa“

前缀集合为{"a","aa","aab","aaba"}，后缀集合为{"a","aa","baa","abaa"}，其公共前后缀包括"a"和"aa"两个子串，最长公共前缀为"aa"这个子串。

KMP算法

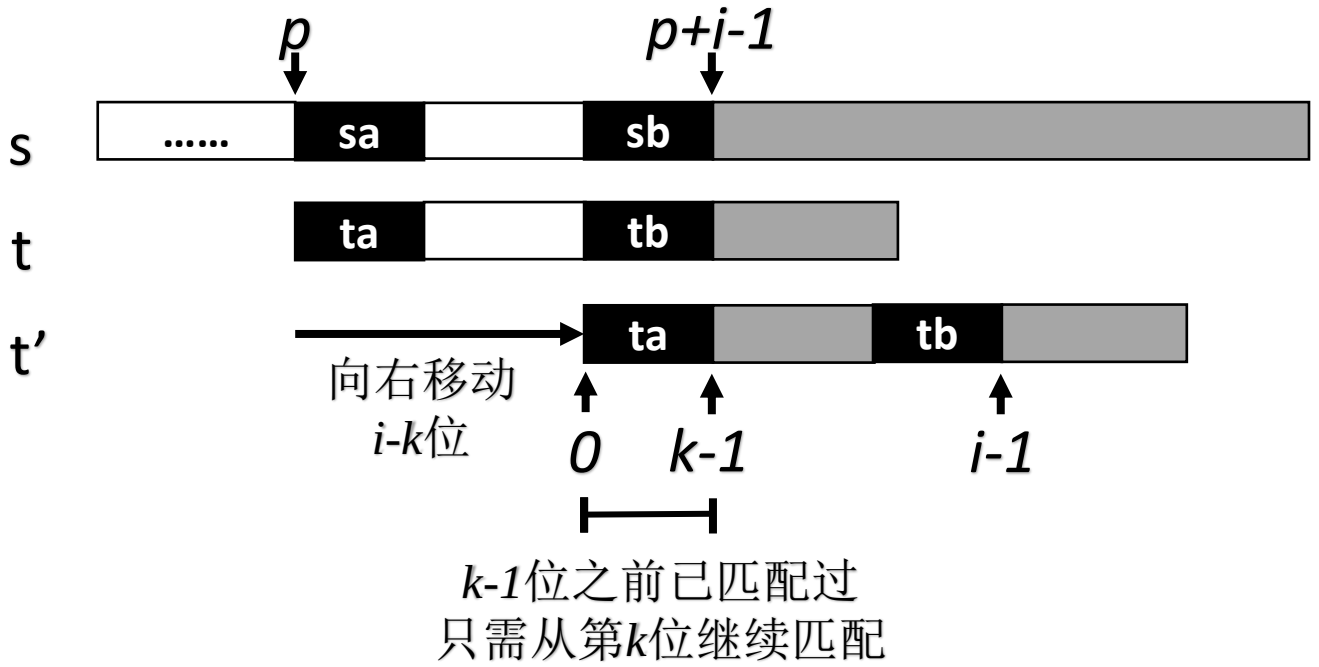
假设当前模式串t已经移动到目标串s的某一位置p，即字符串t的第0位字符与s的第p位字符对齐，正在对模式串t的第i位进行匹配，则说明模式串t的i-1位之前的子串已经完成匹配，即s[p...p+i-1]和t[0...i-1]相同。



由于s[p...p+i-1]和t[0...i-1]已成功匹配，所以显然有sa=sb=ta=tb成立，记**最长公共前后缀**长度为k。

KMP算法

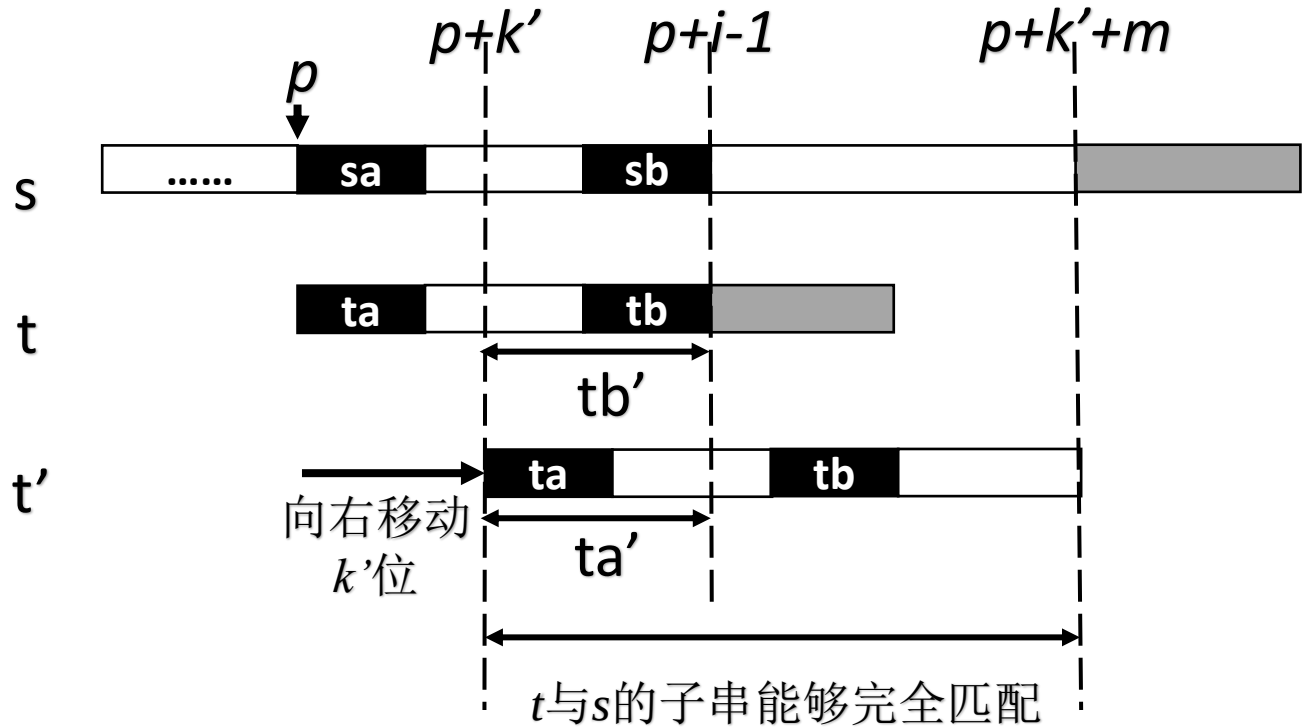
当模式串t的第i位与目标串s的第p+i位出现失配情况时，如果把字符串t向右移动i-k位，使sb和ta对齐，如图4-7所示。那么，由于sb=ta，所以只需要从ta的下一位即字符串t的第k位与字符串s的第p+i位再开始进行匹配即可，就不需要再从字符串t的第0位开始进行匹配。



这种方法虽然能够大大提升匹配的效率，但多位移动中会不会漏掉一些匹配的情况?

KMP算法

记模式串t的长度为m，假设出现了这样的情况，即s的子串s[p+k ‘...p+k’ +m]与字符串t完全相同，那么显然子串s[p+k ‘...p+i-1]与模式串t的前缀ta’ 相同，又由于前i-1位已经成功匹配，故子串s[p+k ‘...p+i-1]与模式串t的后缀tb’ 也相同，也就是说模式串t的前缀ta ‘与后缀tb’ 也相同，是t的公共前后缀。但由于ta ‘长度大于ta，与ta是t的最长公共前后缀矛盾，所以KMP算法在多位移动中不会漏掉可能出现的匹配情况。



KMP算法

KMP算法中向右移动的位数由已匹配部分字符串的最长公共前后缀的长度决定。
如何计算这个位数?

字符串特征向量：长度为m的字符串t的特征向量是一个m维向量，通常记作next，且用数组形式进行存储，所以特征向量也可非正式地称为**next数组**。next[i]表示字符串特征向量的第i位分量（0≤i<m），其形式化定义为：

$$next[i]=\begin{cases}\text{满足}t[0\ldots k]=t[i-k\ldots i]\text{的最大}k(\text{存在}k,\ \text{且}k<i)\\ -1,\qquad\qquad\qquad\qquad\text{如果这样的}k\text{不存在}\end{cases}$$

next[i]表示的真实含义是字符串t的子串t[0...i]的**最长公共前后缀中的前缀最末尾的字符的位置。**
由于字符串从0位开始，所以t[0...i]的**最长公共前后缀长度为next[i]+1**。如果t[0...i]的最长公共前后缀不存在，那么将next[i]置为-1。

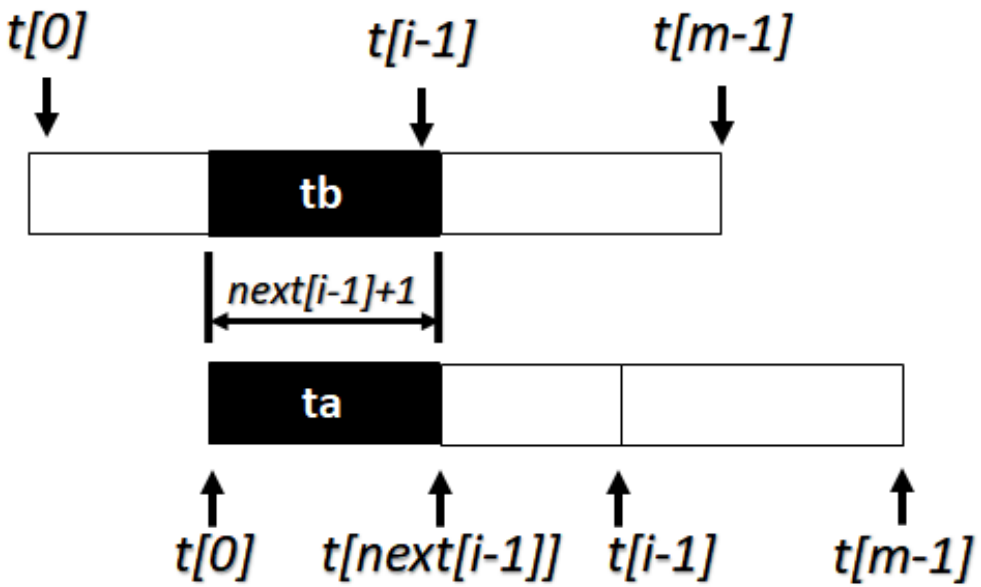
KMP算法

例：字符串"abcac"的特征向量为[-1,-1,-1,0,-1]

i	子串 *t*[*0...i*]	前缀集合	后缀集合	最长 公共前后缀	特征向量 第*i*位分量
i=*0*	a	空	空	空	-1
i=1	ab	a	b	空	-1
i=2	abc	a,ab	c,bc	空	-1
i=3	abca	a,ab,abc	a,ca,bca	a	0
i=4	abcac	a,ab,abc,abca	c,ac,cac,bcac	空	-1

KMP算法

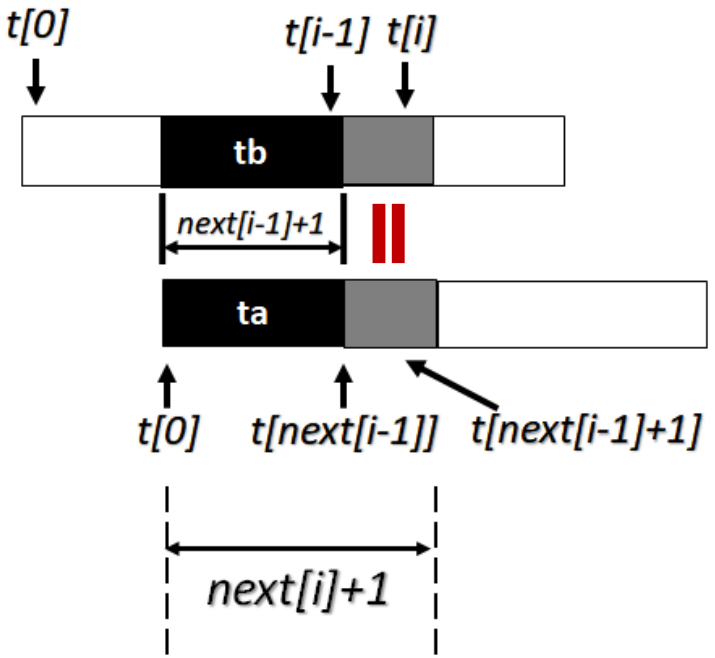
- 若朴素计算next数组，时间复杂度为$O(m^3)$，不能满足进行快速模式匹配的要求
- 一种时间复杂度为 **O(m)**的字符串特征向量算法：
  - 图示为在字符串t中next[i-1]的含义，黑色背景部分为子串t[0...i-1]的最长公共前后缀，前缀用ta表示，后缀用tb表示，有ta=tb成立，next[i-1]表示的是ta最末位字符的下标，next[i-1]+1表示的是最长公共前后缀的长度。



KMP算法

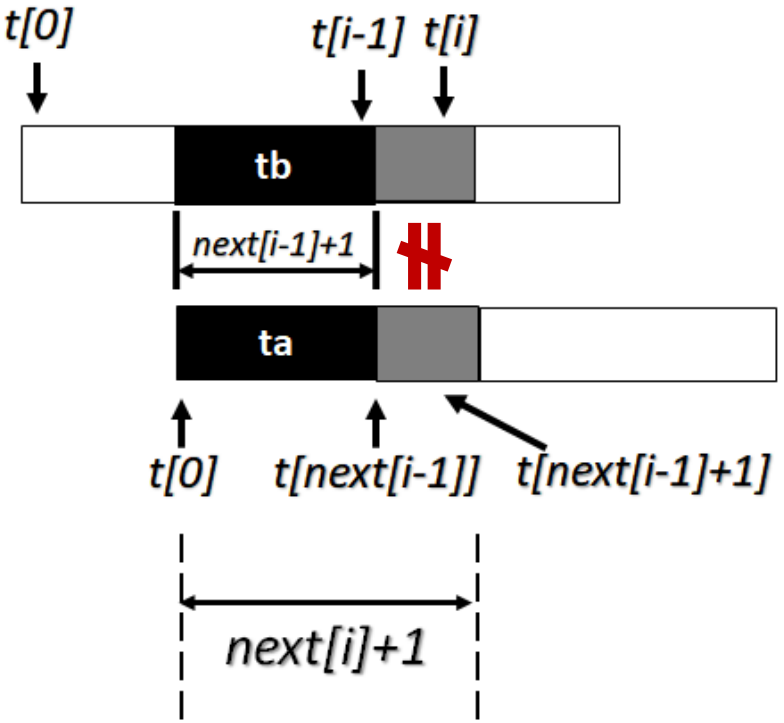
考察tb后面一个字符，即t串第i位置上的字符t[i]与ta后面一个字符，即字符串t第next[i-1]+1位字符t[next[i-1]+1]进行比较，则可对next[i]的值进行推导。

- **情况1**：如果t[i]与t[next[i-1]+1]相等，则tb加上字符t[i]构成的子串与ta加上字符t[next[i-1]+1]构成的子串能够完全匹配，这两个子串即为t[0...i]的最长公共前后缀，故可推出next[i]=next[i-1]+1



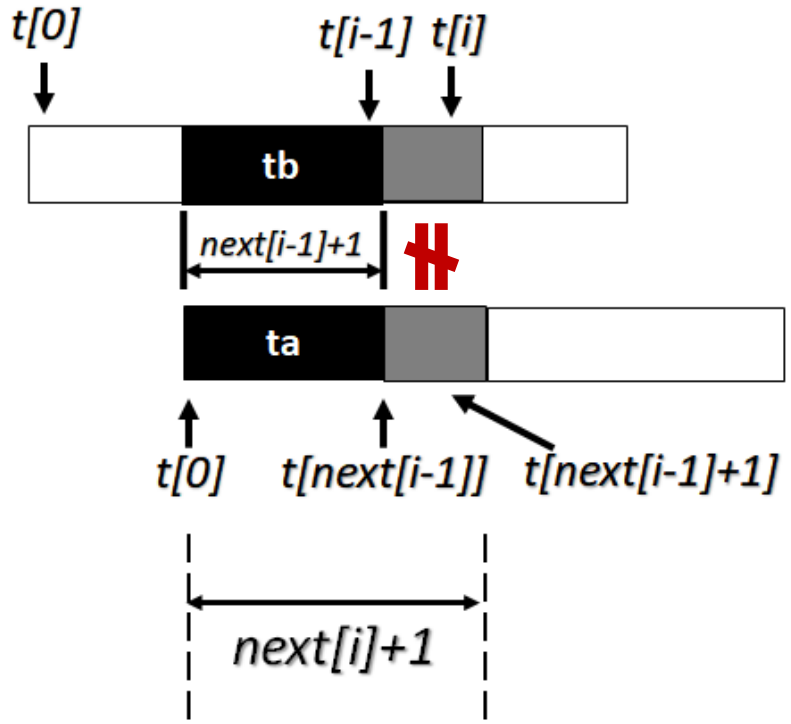
KMP算法

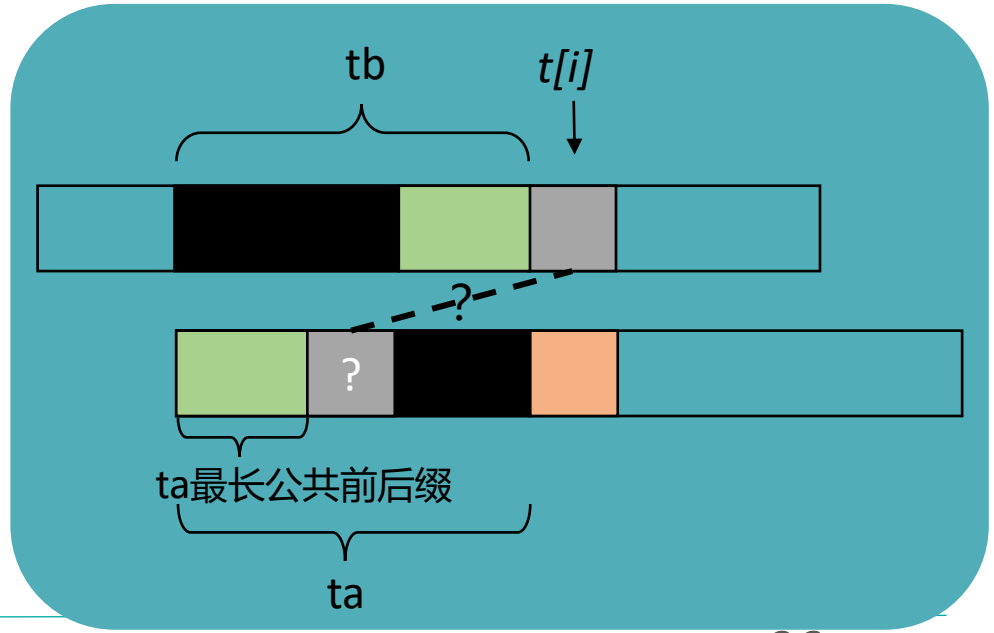
- **情况2**：如果当前t[i]与t[next[i-1]+1]不同，则说明tb与其后一个字符形成的子串t[0...i]的后缀，与tb与其后一个字符形成的子串t[0...i]的前缀并不匹配。
  - 所以，在求解子串t[0...i]的最长公共前后缀时需要进一步缩小搜索范围，在子串t[0...next[i-1]]中去寻找可能的最长公共前后缀，即为t[0...i]的最长公共前后缀，而子串t[0...next[i-1]]的最长公共前后缀的前缀的末位下标即为next[next[i-1]+1]。在此种情况下，有next[i]=next[next[i-1]+1]成立。



KMP算法

- 如果当前t[i]与t[next[i-1]+1]不同，则说明tb与其后一个字符形成的子串t[0...i]的后缀，与tb与其后一个字符形成的子串t[0...i]的前缀并不匹配。
  - 此时，t[0...i]的最长公共前后缀的长度一定比len(tb)+1=next[i-1]+1更小。我们需要找到**仅次于next[i-1]+1的一个长度j**，使得t[0...j-1]=t[i-1-j, ..., i-1]，并且t[j]=t[i]
  - 注意到，t[0...next[i-1]]=ta=tb。第一次缩小搜索范围，尝试ta的最长公共前后缀，也即是t[0....next[next[i-1]]]，看该前缀后面的一个字符t[next[next[i-1]]+1]是否与t[i]一样。如果一样，就定位到了t[0...i]的最长公共子前后缀。否则进一步缩小搜索范围，需要找ta的最长公共前后缀内部更短的前缀，是否与其后一个字符组合起来等于t[0...i]的某个后缀。该过程迭代进行，直到找到或者不能再往前





KMP算法

- 设j=next[i-1]，则问题变为求子串t[0...j]的最长公共前后缀长度next[j+1]，而next[j+1]可由同样的方法求得，即如果t[j]与t[next[j-1]+1]相等，则next[j]=next[j-1]+1，否则next[j]=next[next[j-1]+1]，设j'为next[j-1]，继续按同样的方法求next[j']，直到t[j']与t[next[j'-1]+1]相等或j'不合法即小于0为止。

next数组的求解算法：

```
算法4-13 求解字符串t的next数组GetNext (t, next)
输入：字符串t
输出：字符串t的next数组
m ← t.length
next[0] ← -1
for i←1 to m-1 do    //求出next[1]~next[m-1]
| j ← next[i-1]
| while j ≥ 0 且 t.data[i] ≠ t.data[j+1] do
| | j ← next[j]
| end
| if t.data[i]=t.data[j+1] then
| | next[i] ← j+1
| else
| | next[i] ← -1
| end
end
```

KMP算法

KMP算法流程：在next数组的基础上，在目标串与模式串在模式串的第i位失配时，将模式串右移，从模式串的第next[i-1]+1位开始进行匹配，并不断执行以上步骤直到字符串末尾。

时间复杂度：算法执行过程中目标串只向右移动，比较的复杂度为O(n)。计算next数组的复杂度为O(m)，其中m为模式串的长度，故KMP算法的时间复杂度为**O(n+m)**

算法4-14 字符串匹配的KMP算法PatternMatchKMP(s, t)

```
输入：目标串s，模式串t
输出：返回首个有效匹配位置p，匹配失败则返回NIL
n ← s.length
m ← t.length
p ← NIL
if n ≥ m then
| GetNext (t, next)
| i ← 0
| j ← 0
| while j<n且i<m do
| | if s.data[j]=t.data[i] then //匹配一个字符
| | | i ← i+1
| | | j ← j+1
| | else if i>0 then //找到回退的位置
| | | i ← next[i-1]+1
| | else //找不到匹配，在目标串中后移
| | | j ← j+1
| | end
| end
| if i=m then
| | p ← j-m
| end
end
return p
```

带有通配符的字符串匹配

概念：通配符是一种特殊语句，主要有星号(*)和问号(?)，用来模糊搜索。带有通配符的字符串是一种带有星号和问号的字符串，主要用于字符串的模糊匹配。其中，'*'可以匹配0个或多个连续的任意字符，'?'可以匹配1个任意字符。

例：模式串t为"a*b?c"，字符串"axyzbdc"、"abvc"均能够匹配模式串t。

注意：带有通配符的字符串匹配定义与4.4节中提到的字符串模式匹配不同，其定义为判断目标串s和模式串t是否可以完全匹配，而非子串匹配。

例如，模式串t可以和目标串s的子串匹配，但不能与目标串s完全匹配，则认为模式串t与目标串s失配。

带有通配符的字符串匹配

一种递归算法：

1. s或者t其中一个已经到末尾了，那么如果t的剩余字符都是'*'，返回匹配，否则返回不匹配；

2. 如果s的当前字符和t的当前字符相等，继续向后移动；

3. 当s的当前字符和t的当前字符不相等，分为三种情况：

(1) t的当前字符不是'*'或'?'，返回不匹配；

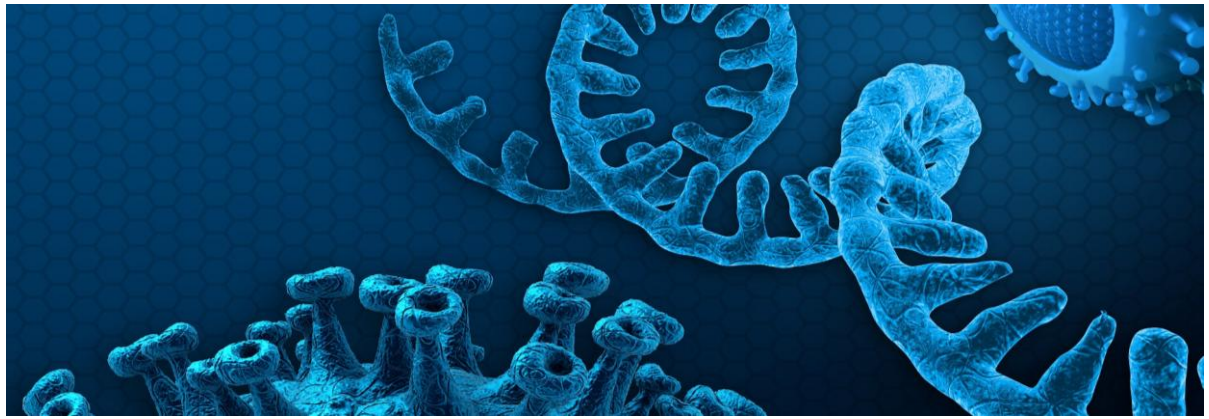
(2) 若t的当前字符是'?'，继续向后移动；

(3) t的当前字符是'*'，那么可跳过s的0到多个字符，再递归判断是否匹配

时间复杂度：由于对每次递归判断过程，目标串s和模式串t的指针i和j都是单调递增的，所以复杂度为O(n+m)。由于最多递归O(n)次，所以该算法的时间复杂度为**O(n(n+m))**。

4.6 应用场景：基因测序

示例场景：生物信息学中已知一串复杂的病毒RNA序列，其规模可以达到百万级别。致病性的RNA序列段规模达到万级，此时科学家需要判断这串病毒的RNA序列中是否包含致病序列段。



解决方法：将复杂的病毒RNA序列认为是目标串s，致病性的RNA序列段认为是模式串t，使用KMP算法解决模式匹配问题，判断模式串t是否为目标串s的子串，如果是，则可以认定该病毒具有致病性，反之则认为该病毒没有致病性。

4.7 小结

字符串和字符类型的线性表的异同点：

字符串：主要是对字符串的整体操作　　**线性表**：是对表中元素的操作。

字符串的主要操作：

字符串插入、字符串删除、字符串截取、字符串连接、字符串比较

字符串的两种实现方法：

顺序存储结构实现，链接存储结构实现

介绍了字符串匹配中的几个重要算法：

朴素算法、KMP算法、BM算法等。

4.7 小结

各种匹配算法的比较：

- 朴素算法通过逐位比较的方式进行匹配，思路最容易理解，算法时间复杂度为O(nm)；
- KMP算法利用利用字符串特征向量加速匹配过程，在任何场景下算法复杂度能够达到稳定的O(n+m),在小字符集模式应用场景下表现尤其突出；
- BM算法基于坏字符与好后缀规则来优化匹配算法，最坏情况复杂度为O(n+m)，更适用于大字符集模式、字符较为随机的应用场景；
- KR算法和Sunday算法分别利用散列表技术和类似于坏字符规则的方法来提高匹配效率，虽然最坏情况下时间复杂度均为O(nm)，但在字符随机出现的情况下表现良好，并且由于其简单明了、实现简单的优势，在工程实践中得到广泛关注。

谢谢观看