中国科学技术大学
University of Science and Technology of China

数据结构

串（5 学时）

October 19, 2020

目录

串的概念

主要概念与术语

- 串 (字符串)：是零个或多个字符组成的有限序列。记作：$S =$ "$a_1\, a_2\, a_3 \ldots$"，其中 S 是串名，$a_i (1 \leq i \leq n)$ 是单个字母、数字或其它字符
- 串值：双引号括起来的字符序列是串值
- 串长：串中所包含的字符个数称为该串的长度
- 空串 (空的字符串)：长度为零的串称为空串，它不包含任何字符
- 空格串 (空白串)：构成串的所有字符都是空格的串称为空白串
- 串和空白串的不同，例如 " " 和 "" 分别表示长度为 1 的空白串和长度为 0 的空串。

子串/substring

- 串中任意个连续字符组成的子序列称为该串的子串，包含子串的串相应地称为主串
- 空串是任意串的子串，任意串是其自身的子串

串的概念

子串的位置

- 将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）
- 例如，设有串 A 和 B 分别是：A=“这是字符串是”，B=“是”则 B 是 A 的子串，A 为主串。B 在 A 中出现了两次，其中首次出现所对应的主串位置是 3。因此，称 B 在 A 中的序号为 3 汉字占两个字符

串相等

- 如果两个串的串值相等 (相同)，称这两个串相等
- 换言之，只有两个串的长度相等，且各个对应位置的字符都相同时才相等

串变量和串常量

- 串常量和整常数、实常数一样，在程序中只能被引用但不能改变其值，即只能读不能写
- 通常串常量是由直接量来表示的，例如语句错误（“溢出”）中“溢出”是直接量
- 串变量和其它类型的变量一样，其值是可以改变

串的 ADT

```
1  ADT String{
2    数据对象: D = {a_i | a_i ∈ CharacterSet, i = 1, 2, ..., n, n ≥ 0}
3    数据关系: R = {< a_{i-1}, a_i > | a_{i-1}, a_i ∈ D, i = 2, 3, ..., n}
4    基本操作:
5        StrAssign(&t,chars) //生成一个值为chars的串t
6        StrConcat(&s,t) // 将串t联结到串s后形成新串存放到s中
7        StrLength(t) //返回串t中的元素个数，称为串长
8        SubString(s,pos,len,&sub) //用sub返回串s的子串
9        Strcopy(&s,t)
10       Strcmp(s,t)
11       Replace(&s,t,v)
12       ......
13 } ADT String
```

串的存储实现

串的主要存储方式

- 定长顺序存储方式：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变
- 堆分配存储方式：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的
- 块链存储方式：是一种链式存储结构表示

评述

- 串是一种特殊的线性表，其存储表示类似线性表，但又不完全相同
- 串的存储方式取决于将要对串所进行的操作

串的定长顺序存储方式

方法

- 用一组连续的存储单元来存放串中的字符序列
- 所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定

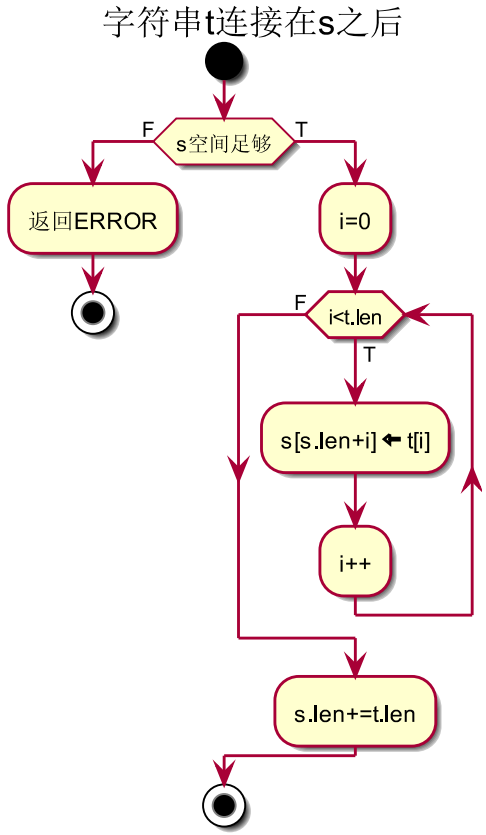
定长顺序存储结构的定义

```
1  #define MAX_STRLEN 256
2  typedef struct{
3    char str[MAX_STRLEN];
4    int length;
5  }StringType;
```

串的连接操作实现

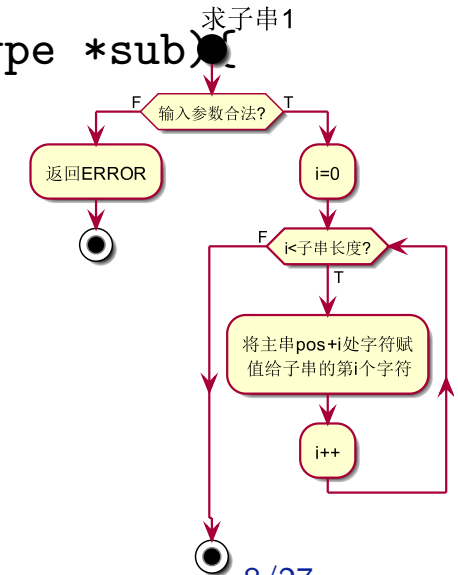
将一个串拼接在另一个串后

```
1  Status StrConcat(StringType &s,StringType t){
2    int i;//将串t联结到串s之后，结果仍然保存在s中
3    if ((s.length+t.length)>MAX_STRLEN)
4      Return ERROR; //联结后长度超出范围
5    for (i=0;i<t.length;i++)
6      s.str[s.length+i]=t.str[i];//串t联结到串s之后
7    s.length=s.length+t.length;//修改联结后的串长度
8    return OK;
9  }
```



求主串给定起点和长度的子串

```
1  Status SubString(StringType s,int pos,int len,StringType *sub){
2    int k,j;
3    if (pos<0||pos>s.length-1||len<0||len>s.length-pos)
4      return ERROR;//参数非法
5    sub->length=len; //求得子串长度
6    for (j=0,k=pos;j<len;k++, j++)
7      sub->str[j]=s.str[k];//逐个字符复制求得子串
8    return OK;
9  }//sub指向了一个已经存在的结构体
```



串的堆分配存储方式

方法

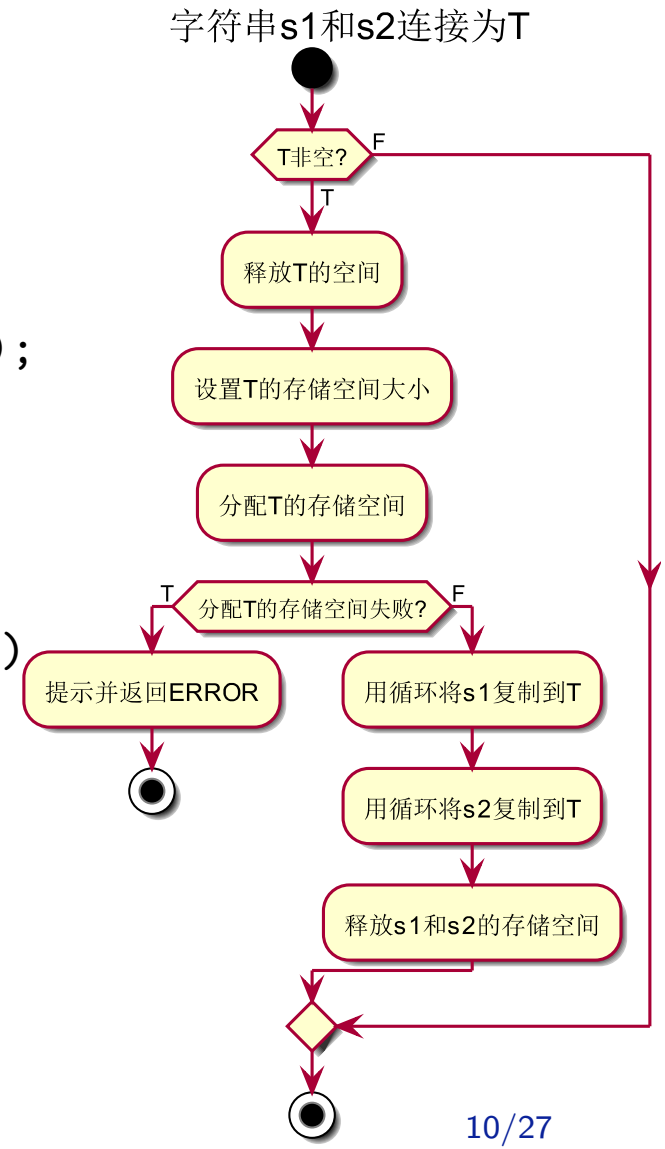
- 系统提供一个空间足够大且地址连续的存储空间 (称为“堆”) 供串使用
- 可使用 C 语言的动态存储分配函数 malloc() 和 free() 来管理
- 仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的

串的堆式存储结构的类型定义

```
1  typedef struct{
2    char *ch;//若非空，按长度分配，否则为NULL
3    int length; //串的长度
4  }HString;
```

串的连接操作实现

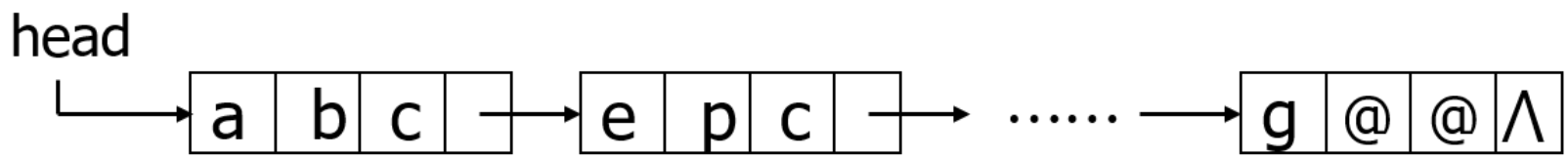
```
1  Status StrConcat(HString *T,HString *s1,HString *s2){
2  //用T返回由s1和s2联结而成的串
3    int k,j,t_len;
4    if (T->ch) free(T->ch); //释放旧空间
5    t_len=s1->length+s2->length;
6    if ((T->ch=(char *)malloc(
7              sizeof((char)*t_len))==NULL){
8      printf(``系统空间不够, 申请空间失败 ! \n'');
9      return ERROR;
10   }
11   for (j=0;j<s1->length; j++)
12     T->ch[j]=s1->ch[j];//将串s复制到串T中
13   for (k=s1->length,j=0;j<s2->length;k++, j++)
14     T->ch[k]=s2->ch[j]; //将串s2复制到串T中
15   free(s1->ch);
16   free(s2->ch);
17   return OK;
18 } //思考一下为什么T能带回返回值?
```



串的块链存储方式

方法

- 一种链式存储方式，块链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串
- 结点的构成包括 data 域以存放字符，data 域可存放的字符个数称为结点的大小；next 域：存放指向下一结点的指针
- 若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构
- 例子，如图所示块大小为 3 的块链



串的块链存储方式的实现

```
1  #define BLOCK_SIZE 4
2  //块结点的类型定义
3  typedef struct Blstrtype{
4    char data[BLOCK_SIZE];
5    struct Blstrtype *next;
6  }BNODE;
7
8  //块链串的类型定义
9  typedef struct{
10   BNODE head;//头指针
11   int Strlen; //当前长度
12 }Blstring;
```

进一步说明

- 在这种存储结构下，结点的分配总是以完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结
- 当一个块 (结点) 内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符

串的基本操作的实现

类似线性表

- 略

串模式匹配的基本概念

模式匹配：找子串在主串中的位置

- 子串 T 在主串 S 中的定位称为模式匹配或串匹配 (字符串匹配)
- 模式匹配成功是指在主串 S 中能够找到模式串 T，否则，称模式串 T 在主串 S 中不存在

模式匹配的应用场景

- 模式匹配的应用在非常广泛
- 例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置，显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能

模式匹配算法

- 模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。介绍两种主要的模式匹配算法
  - Brute-Force/暴力模式匹配算法
  - KMP 算法

暴力模式匹配

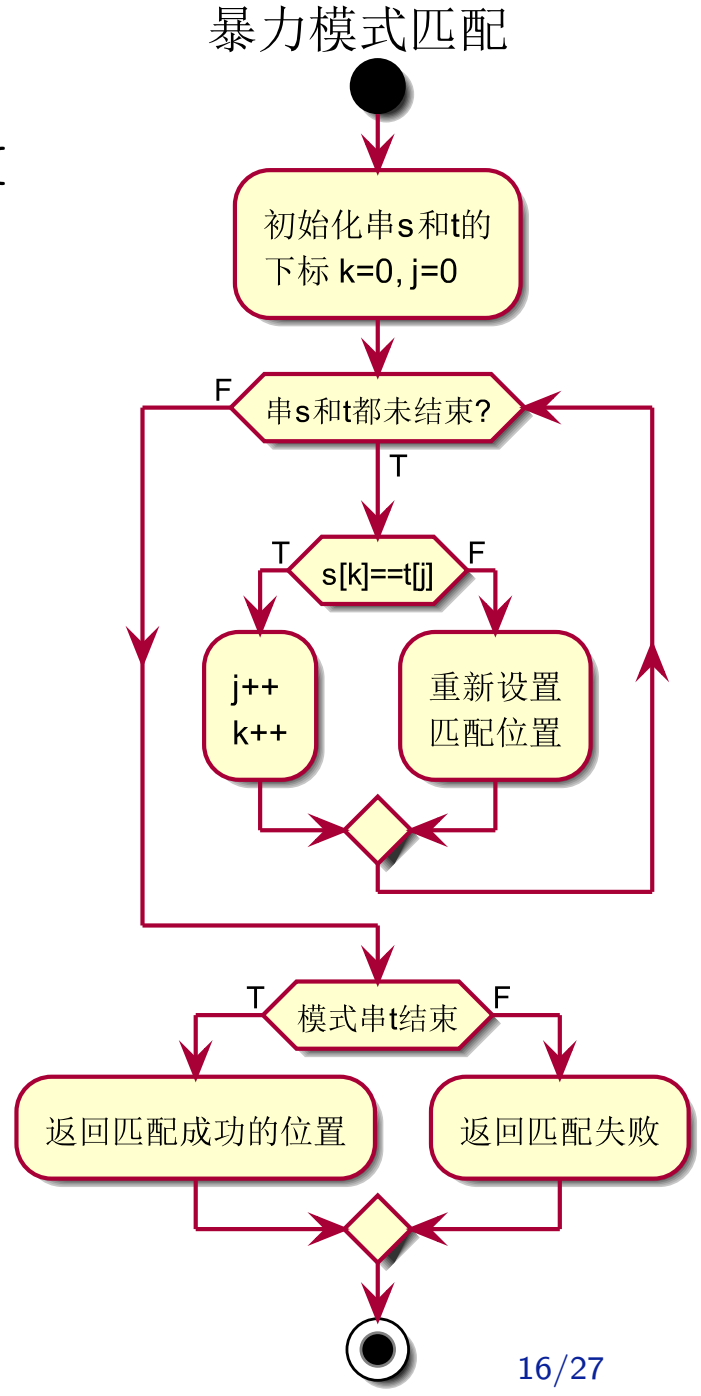
设 S 为目标串，T 为模式串

- 不妨设：$S = "s_0 s_1 \dots s_{n-1}"$, $T = "t_0 t_1 \dots t_{m-1}"$
- 串的匹配实际上是对位置 $0 \le i \le n - m$ 依次将目标串中的子串 $s[i \dots i + m - 1]$ 和模式串 $t[0 \dots m - 1]$ 进行比较
  - 若 $s[i \dots i + m - 1] = t[0 \dots m - 1]$：则称从位置 i 开始的匹配成功，亦称模式 t 在目标 s 中出现；位置 i 称为有效位移
  - 若 $s[i \dots i + m - 1] \neq t[0 \dots m - 1]$：从 i 开始的匹配失败。位置 i 称为无效位移

故，模式匹配就是找出某给定模式 T 在给定目标串 S 中首次出现的有效位移

暴力模式匹配算法的实现

```
1  int IndexString(StringType s,StringType t){
2  /*采用字符数组存储主串s和模式t*/
3  /*返回位置，否则返回-1*/
4    int k,j; //i指向主串，j指向模式串
5    k=0;j=0; //初始匹配位置设置
6    while (k<s.length)&&(j<t.length){
7      if (s[k]==t[j]){
8        k++;
9        j++;
10     } else {//重新设置匹配位置
11       k=k-j+1;
12       j=0;
13     }
14   }
15   if (j==t.length)
16     return(k-t.length);//匹配，返回位置
17   else return(-1);//不匹配，返回-1
18 }
```



暴力模式匹配算法分析

算法的关键点

- 当第一次 $s_k \neq t_j$ 时：主串要退回到 $k-j+1$ 的位置，而模式串也要退回到第一个字符（即 $j=0$ 的位置）
- 比较出现 $s_k \neq t_j$ 时：则应该有 $s_{k-1}=t_{j-1},\ldots,s_{k-j+1}=t_1,s_{k-j}=t_0$，即模式串的 $j-1$ 个字符已经在主串中匹配上了

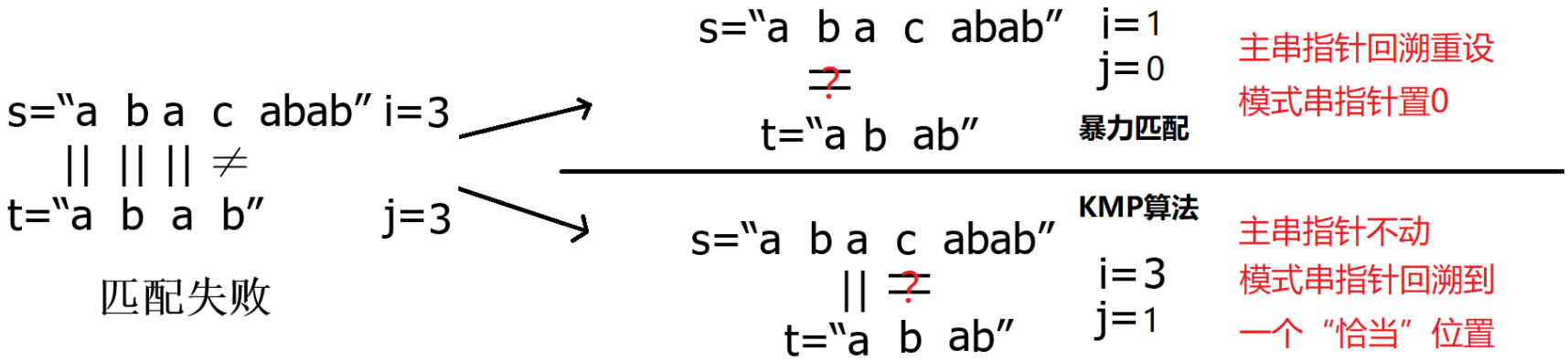
算法特点

- 简单，易于理解。在一些场合的应用，如文字处理中的文本编辑，其效率较高

时间复杂度

- 该算法的时间复杂度为 $O(nm)$ ，其中 n,m 分别是主串和模式串的长度
- 通常情况下，实际运行过程中，该算法的执行时间近似于 $O(n+m)$
- 思考：主串每次匹配失败，只前进一个字符，而模式串完全重来，从位置 0 开始；能不能主串下标不重置，保持不动；模式串下标也不回到 0，回到某个“恰当”的位置？

模式匹配算法的改进：引入 KMP 算法



由 D.E.Knuth，J.H.Morris 和 V.R.Pratt 提出来的，简称为 KMP 算法

- 例：设有串 $s = "abacabab"$，$t = "abab"$，则第一次匹配过程如上图所示
- 在 $i = 3$ 和 $j = 3$ 时，匹配失败；但重新开始第二次匹配时，不必从 $i = 1, j = 0$ 开始。因为 $s_1 = t_1, t_0 \neq t_1$，必有 $s_1 \neq t_0$，又因为 $t_0 = t_2, s_2 = t_2$，所以必有 $s_2 = t_0$。由此可知，第二次匹配可以直接从 $i = 3, j = 1$ 开始
- 总之，在匹配过程中，一旦出现 $s_i \neq t_j$，主串 s 的指针不必回溯，直接与模式串的 $t_k, 0 \leq k < j$ 进行比较，看上去就是将模式串向右“滑动”一段距离后，到一个“恰当”的位置 k(令 $j = k$) 对齐主串的位置 i，开始继续进行比较
- “恰当”位置 k 如何计算?

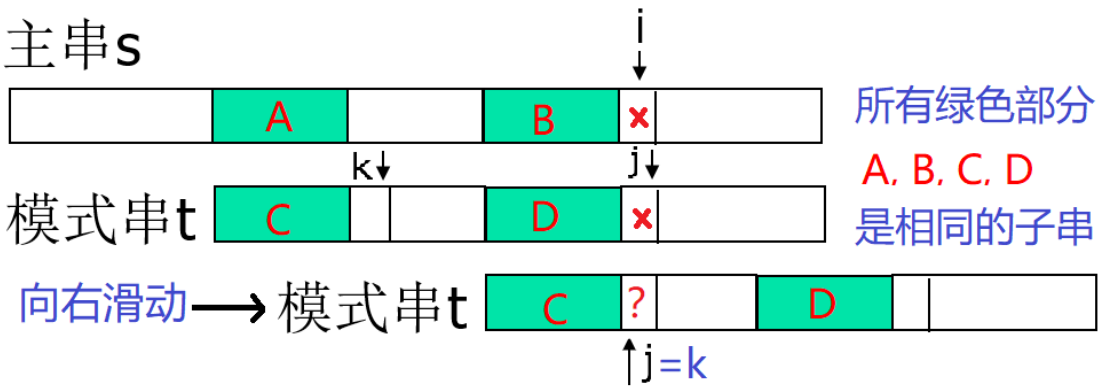
KMP 算法思想溯源

不失一般性，设主串 $S = "s_0 s_1 \ldots s_{n-1}"$，模式串 $T = "t_0 t_1 \ldots t_{m-1}"$

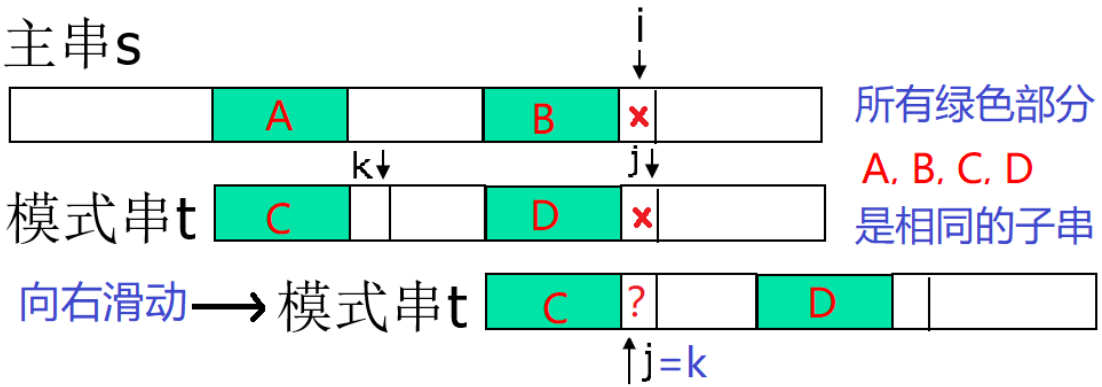
- 当 $s_i \neq t_j, 1 \leq i \leq n-m, 1 \leq j < m, m < n$ 时，主串 s 的指针 i 不必回溯，而模式串 t 的指针 j 回溯到第 $k(k < j)$ 个字符继续比较，换而言之，模式串滑动完后要满足 $B = C$，即满足 (1) 式，而且不可能存在 $k' > k$ 满足 (1) 式

  $t_0 t_1 \ldots t_{k-1} = s_{i-k} s_{i-(k-1)} \ldots s_{i-2} s_{i-1}$ (1)
- 已经得到的“部分匹配”的结果 $B = D$，即

  $t_{j-k} t_{j-(k-1)} \ldots t_{j-2} t_{j-1} = s_{i-k} s_{i-(k-1)} \ldots s_{i-2} s_{i-1}$ (2)
- 由 (1) 和 (2)，可以得到 $C = D$，即 $t_0 t_1 \ldots t_{k-1} = t_{j-k} t_{j-(k-1)} \ldots t_{j-2} t_{j-1}$
- 结论：查找模式串 $t_0 t_1 \ldots t_{j-1}$ 中首尾最长的重复子串 ($C = D$)，该重复子串的长度即为所求的 k (=next(j))，j 回溯的目标位置
- 注意到的事实：k 的取值与主串 s 无关，只与模式串 t 本身的构成有关



KMP 算法的 $next()$ 函数



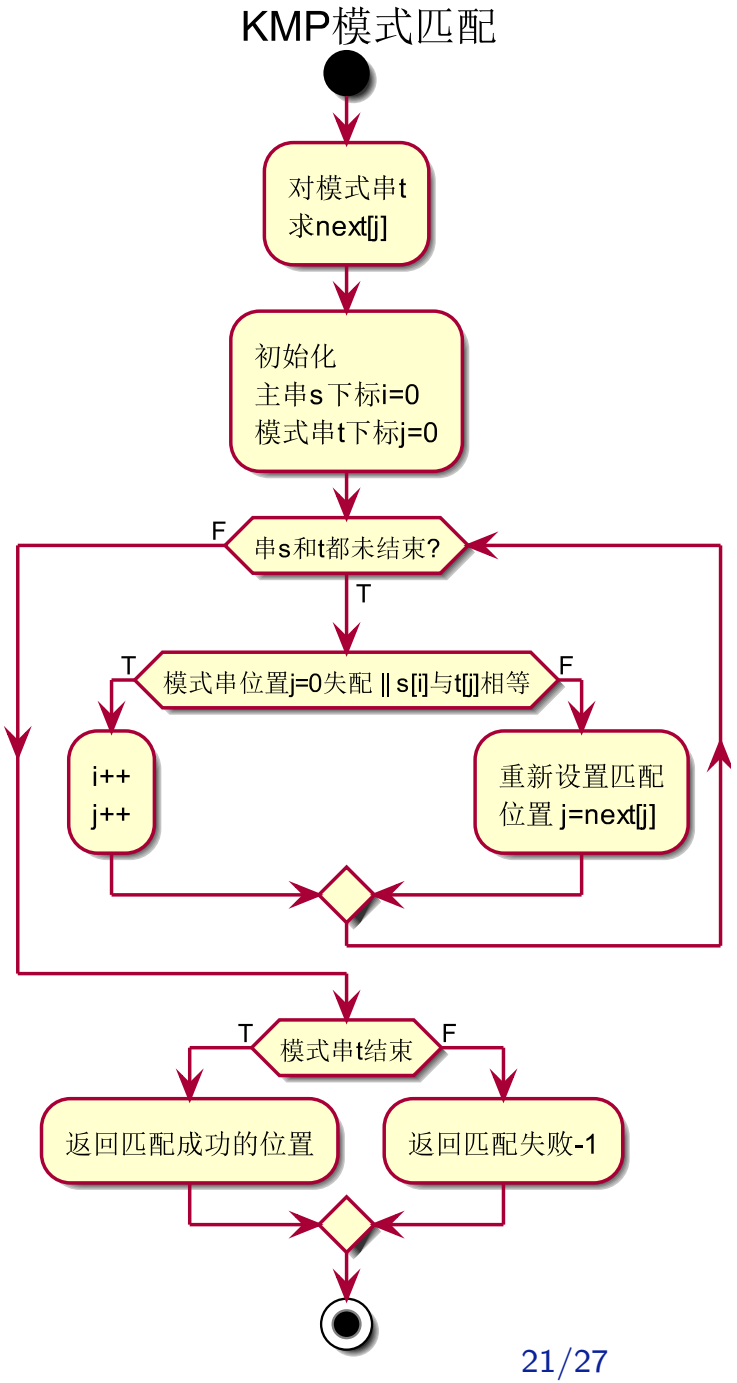
求子串当前失配位置 j 回溯的目标位置 $k = next(j)$ or $k = next[j]$

- 通常对模式串进行一次预处理，对 $\forall j$ 求出 $next(j)$，保存在数组 $next[j]$ 中
- 在 $t_0 t_1 \dots t_{j-1}$ 中找首尾重复子串 $(C = D)$，要求子串长度 k 最大且 $k < j$
- $next[j] = Max\{\{k | 1 \le k < j \wedge t_0 \dots t_{k-1} = t_{j-k} \dots t_{j-1}\} \cup \{0\}\}$
- KMP 执行过程中，若 $next[j] = 1$，就是图中青色部分长度为 1，若 $next[j] = 0$，就是图中青色部分长度为 0
- 小技巧：令 $next[0] = -1$，也就是说在模式串位置 0 失配时，做特殊处理

KMP 算法的实现

算法框架

```
1  #define Max_Strlen 1024
2  int next[Max_Strlen]; //已经算好
3  int KMP_index(char *s, char *t){
4  //s和t分别表示主串和模式串
5    int i=0,j=0;//初始下标
6    while (i<s.len)&&(j<t.len){
7      if (((j==-1))
8                  || (s[i]==t[j])){
9        i++;
10       j++;
11     }
12     else j=next[j];
13   }
14   if (j>=t.len)
15     return(i-t.len);
16   else return(-1);
17 } //行7处条件写成 j==-1 因为next[0]=-1
```



KMP 算法的实现：计算 $next[j]$

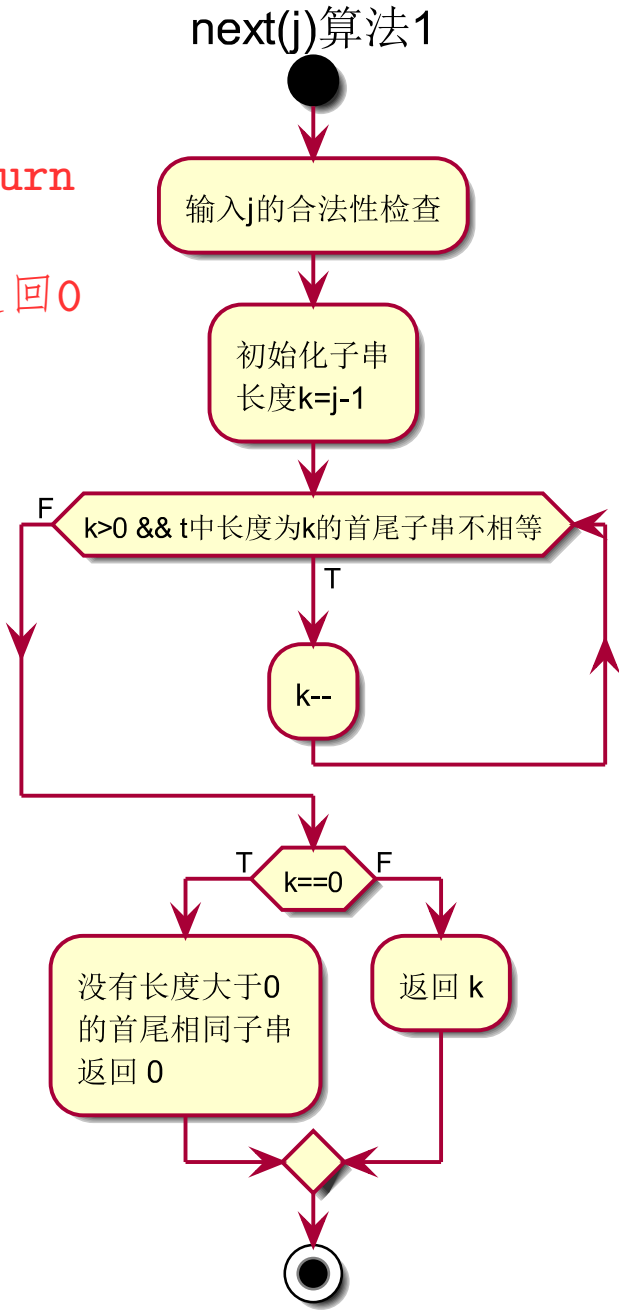
$$next[j] = Max\{\{k|1 \leq k < j \wedge t_0 \ldots t_{k-1} = t_{j-k} \ldots t_{j-1}\} \cup \{0\}\}$$

- $next[j]$ 仅取决于模式串自身特点，有三种计算思路：
  1. $next_1(j)$: 穷举法 — 比较 $t_0 t_1 \ldots t_{j-1}$ 中所有长度为 $k = 1, 2, \ldots, j-1$ 的首尾子串，找出最长相同首尾子串 O(m^3)
  2. $next_2(j)$: 改进穷举法 — 找到形如以 t_0 开始，以 t_{j-1} 结束、长度 $k < j$ 所有子串，然后检查其是否在 $t_0 t_1 \ldots t_{j-1}$ 的首尾都出现
  3. $next_3(j)$: 递推算法 — 假设 $next[1], next[2], \ldots next[j-1]$ 已知，求 $next[j]$

KMP 算法的实现: 计算 $next[j]$

$next_1(j)$: 穷举法

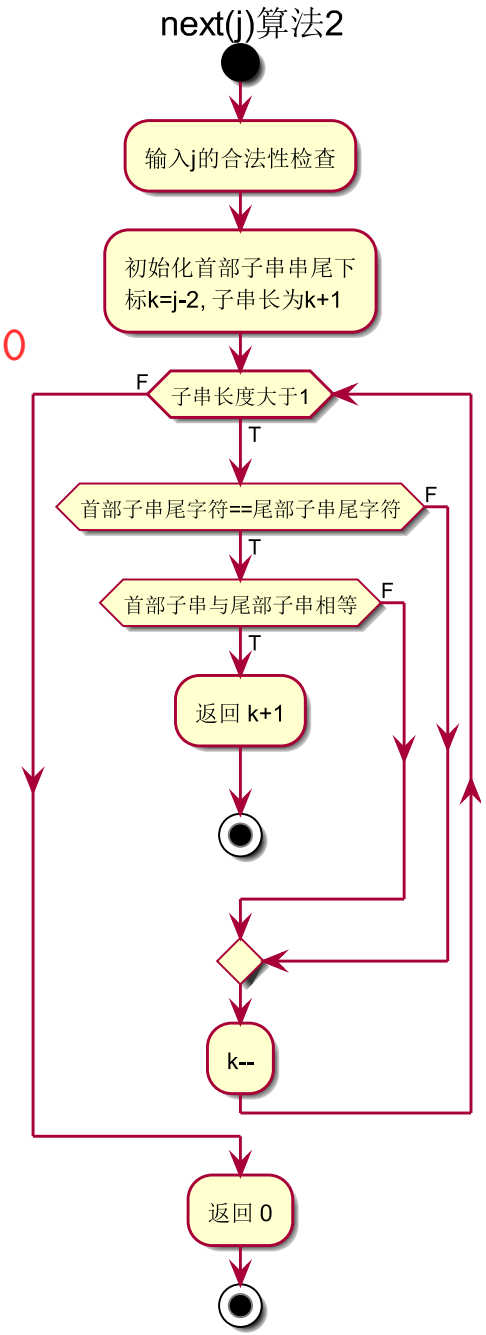
```
1  //借助改进的基本操作subString(), 返回值改用return
2  //char *subString(s,startpos,len)
3  //strncmp(t1,t2,n) 比较t1,t2的前n个字符,相等返回0
4  //求next[]数组, 需要对不同的j, 调用next1(j)
5  int next1(char *t, int j){
6    //0<j<t.len, 这里要做j的合法性检查
7    char *str1, *str2;
8    int k=j;
9    do {
10      k--;
11      str1 = subString(t,0,k);
12      str2 = subString(t,j-k,k);
13   while (k && strncmp(str1,str2,k))
14   if (k==0)
15     return 0;
16   else
17     return k;
18   }
19 }
```



KMP 算法的实现: 计算 $next[j]$

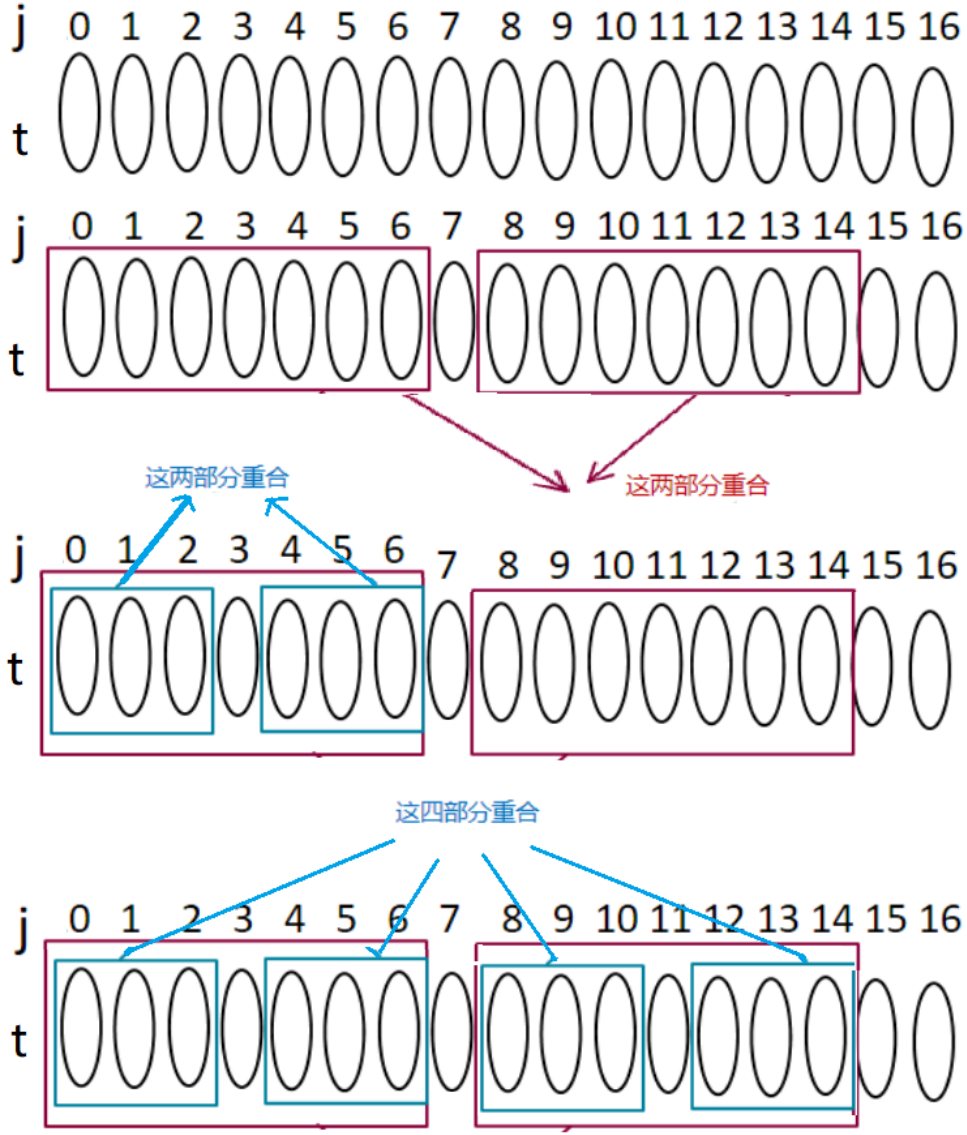
$next_2(j)$: 改进穷举法

```
1  //strncmp(t1,t2,n) 比较t1,t2的前n个字符,相等返回0
2  int next2(char *t, int j){
3    //0<j<t.len, 这里要做j的合法性检查
4    for (int k=j-2;k>=0;--k){
5      if (t[k]==t[j-1])
6        if (strncmp(t, &t[j-k-1],k+1)==0)
7          return k+1;
8    }
9    return 0;
10 }
```



KMP 算法的实现: 计算 $next[j]$

$next_3(j)$: 递推算法的例子



计算 next(j), j=16

j-1=15 next[j-1]=7

若t[7]=t[15]

next(16)=next(15)+1=8

若t[7]!=t[15], 可令

k=next[j-1] = 7

假设 next[k]=3

若t[3]=t[15]

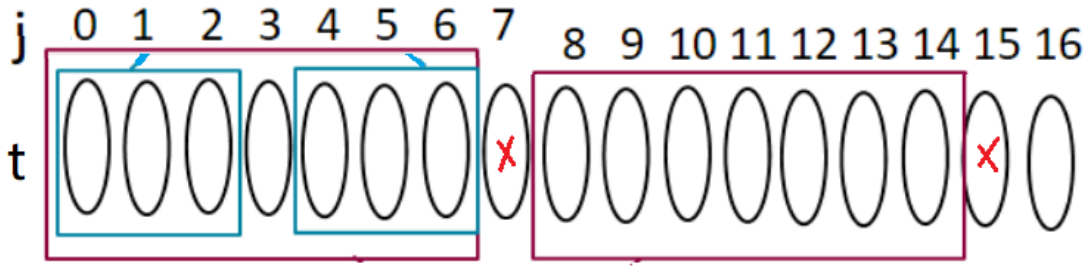
next[16]=next[k]+1=4

类似继续讨论:

t[3]!=t[j-1]时

KMP 算法的实现: 计算 $next[j]$

$next_3(j)$: 递推算法的例子

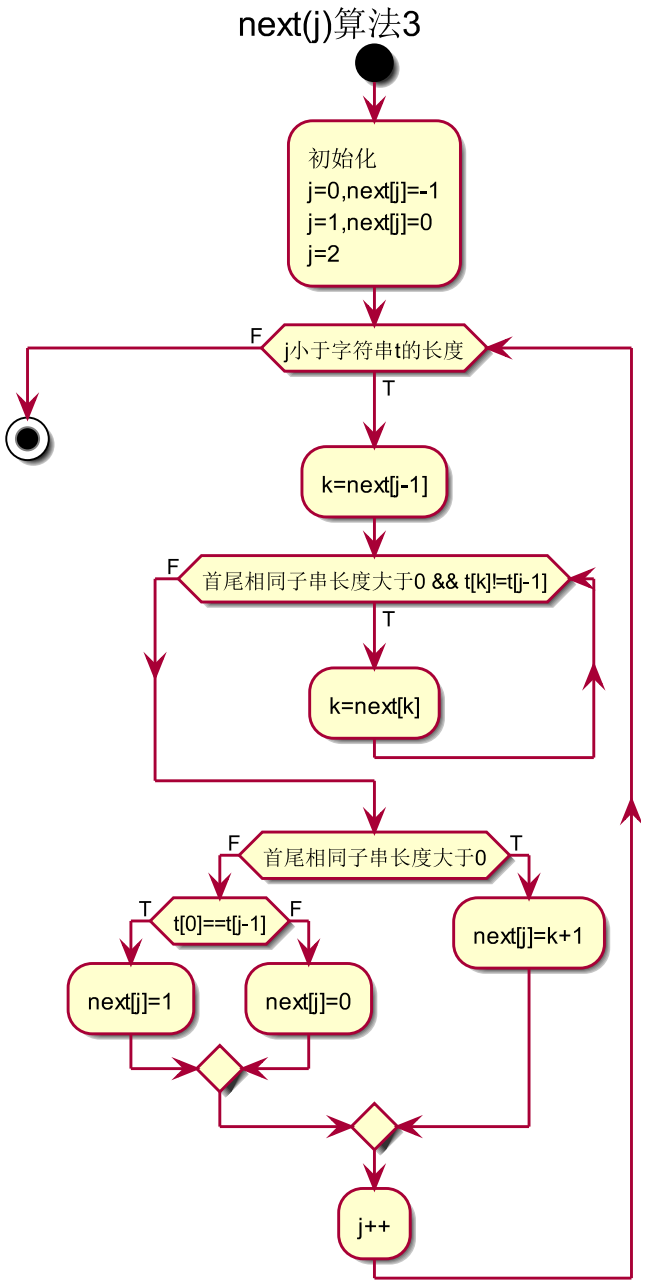


- $k = next[j-1] = next[15] = 7$，当 $t[15]! = t[k]$ 时，算法中令 $k \longleftarrow k' = next[k]$，即 $k = next[next[j-1]]$
- 为什么不考虑 $k > next[j-1]$ 的位置，即图中 $next[8], next[9], \ldots, next[14]$ 的情况，直接考虑 next[7]?
  - 事实 1：此时有 $k_j = next[j] <= next[j-1] = k$，可用反证法证明
  - 故，$next[j]$ 指示的长度为 k_j 尾部子串 $str1$ 起始位置在右边紫色方框内 (或就是 $j-1$，此时 $t_0 = t_{j-1}$ 是最长相同首尾子串)
  - 存在与 $str1$ 完全一样的子串完全落在左侧的紫色框内

KMP 算法的实现

$next[j]$ 的实现代码

```
1  void next3(char *t,int next[]){
2    //求模式t的next数组
3    int k,j=2;
4    next[0]=-1;
5    next[1]=0;
6    while (j<t.length){
7      k=next[j-1];
8      while (k>0 && t[k]!=t[j-1])
9        k=next[k];
10     if (k<=0)
11       if (t[0]==t[j-1])
12         next[j]=1;
13       else
14         next[j]=0;
15     else
16       next[j]=k+1;
17     j++;
18   }
19 }//相比于穷举算法，时间开销降低两个数量级
```



模式匹配练习题

题目说明

- 试分析模式匹配算法的时间复杂度 暴力方法O（mn）平均O（m+n）
- void StrReplace(char *T, char *P, char *S)，将 T 中第一次出现的与 P 相等的子串替换为 S，串 S 和 P 的长度不一定相等，并分析时间复杂度
- 试证明或说明 KMP 算法不会遗漏模式串。(提示：反证法，假设遗漏了某个可匹配的子串，分析这种子串的可能性和特点)
- 给定字符串"abaaababc"，求 next[] 值。答案为：{-1, 0, 0, 1, 1, 1, 2, 3, 2}，利用定义直接看出来，利用三个算法手工执行出来，熟悉和掌握算法