


课前签到

数据结构与算法2023秋季



 微信扫一扫，使用小程序

1. 微信扫码+**实名**

2. 点击今日签到

签到时间：

9:45~10:15

签到地方：

珠海校区-教学
大楼-C407

人脸识别；智能定位

上课纪律



上课期间手机静音：

- 1. 关闭手机**
- 2. 飞行模式**



数据结构与算法 串

余建兴

中山大学人工智能学院

提纲

- 1 串与串的查找
- 2 串的存储结构
- 3 串的模式匹配
- 4 KMP算法实现

8.1 串与串的查找

什么是串

- 线性存储的一组数据（默认是字符）
- 特殊操作集
 - 求串的长度
 - 比较两串是否相等
 - 两串相接
 - 求子串
 - 插入子串
 - 匹配子串
 - 删除子串

什么是串

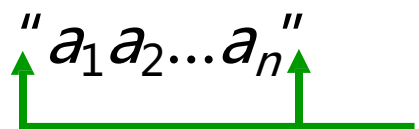
串（或字符串）是由零个或多个字符组成的有限序列。

串 \subseteq 线性表

串中所含字符的个数称为该串的长度（或串长），含零个字符的串称为空串，用 Φ 表示。

串的逻辑表示， a_i ($1 \leq i \leq n$) 代表一个字符：

$"a_1 a_2 \dots a_n"$



双引号不是串的内容，起标识作用

一般记作 $S: "c_0 c_1 c_2 \dots c_{n-1}"$

- N 是串长（串的长度）：一个字符串所包含的字符个数
 - 空串：长度为零的串，它不包含任何字符内容（注意与空格串 " " 的区别）

什么是串



字符串是一种特殊的线性结构

- 数据对象
 - 无特殊限制
 - 串的数据对象为字符集
- 基本操作
 - 线性表的大多以“单个元素”为操作对象
 - 串通常以“串的整体”作为操作对象
- 线性表的存储方法同样适用于字符串
 - 应根据不同情况选择合适的存储表示

串的字符/符号

- **字符** (char) : 组成字符串的基本单位
- 取值依赖于字符集 Σ (同线性表, 结点的有限集合)
 - 二进制字符集 : $\Sigma = \{0,1\}$
 - 生物信息中 DNA 字符集 : $\Sigma = \{A,C,G,T\}$
 - 英语语言 : $\Sigma = \{26\text{个字符}, \text{标点符号}\}$
 -

字符编码



- 单字节 (8 bits)
 - 采用 ASCII 码对 128 个符号进行编码
 - 在 C 和 C++ 中均采用
- 其他编码方式
 - GB
 - CJK
 - UNICODE

字符编码顺序

- 为了字符串间比较和运算的便利，字符编码表一般遵循约定俗成的 “**偏序编码规则**”
- **字符偏序**：根据字符的自然含义，某些字符间 两两可以比较次序
 - 其实大多数情况下就是**字典序**
 - 中文字符串有些特例，例如 “笔划” 序

字符串的数据类型

- 因语言而不同
 - 简单类型
 - 复合类型
- 字符串常数和变量
 - 字符串常数 (string literal)
 - 例如： “\n” , “a” , “student” ...
 - 字符串变量

子串

串相等：当且仅当两个串的长度相等并且各个对应位置上的字符都相同时，这两个串才是相等的。如：

"abcd" ≠ "abc"

"abcd" ≠ "abcde"

所有空串是相等的。

子串：一个串中任意个连续字符组成的子序列（含空串）称为该串的子串。

- 例如，*"abcde"* 的子串有：
- *""*、*"a"*、*"ab"*、*"abc"*、*"abcd"* 和 *"abcde"* 等
- 空串是任意串的子串
- 任意串 *S* 都是 *S* 本身的子串
- 真子串是指不包含自身的所有子串。

串的基本运算

C 标准函数库需要 `#include <string.h>`

- 求串长 `int strlen(char *s);`
- 串复制 `char *strcpy(char *s1, char *s2);`
- 串拼接 `char *strcat(char *s1, char *s2);`
- 串比较 (注意)
 - `int strcmp(char *s1, char *s2);`
 - 看 ASCII 码, $s1 > s2$, 返回值 > 0 ; 两串相等, 返回 0
- 定位 `char *strchr(char *s, char c);`
- 右定位 `char *strrchr(char *s, char c);`
- 求子串 `char *strstr(const char *str1, const char *str2);`

定位函数示例

- 字符串 s :

| | | | | | | | | | | | |
|---|---|---|---|---------------|---|---|----------------|---|---|----|----|
| H | e | l | l | o | | w | o | r | l | d | \0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | | | | ↑ | | | ↑ | | | | |
| | | | | strchr(s,'o') | | | strrchr(s,'o') | | | | |

- 寻找字符 o, strchr(s,'o') 结果返回 4
- 反方向寻找 r, strrchr(s,'o') 结果返回 7

串抽象数据类型

C++ 标准字符串类库

```
#include <string>  
using namespace std;
```

- 字符串类 (class String)
 - 适应字符串长度动态变化的复杂性
 - 不再以字符数组 char S[M] 的形式出现，而采用一种动态变长的存储结构

串抽象数据类型

| 操作类别 | 方法 | 描述 |
|--------|-----------------------|------------------------|
| 子串 | substr () | 返回一个串的子串 |
| 拷贝/ 交换 | swap () | 交换两个串的内容 |
| | copy () | 将一个串拷贝到另一个串中 |
| 赋值 | assign () | 把一个串、一个字符、一个子串赋值给另一个串中 |
| | = | 把一个串或一个字符赋值给另一个串中 |
| 插入/ 追加 | insert() | 在给定位置插入一个字符、多个字符或串 |
| | append () / += | 将一个或多个字符、或串追加在另一个串后 |
| 拼接 | + | 通过将一个串放置在另一个串后面来构建新串 |
| 查询 | find () | 找到并返回一个子序列的开始位置 |
| 替换/ 清除 | replace () | 替换一个指定字符或一个串的字串 |
| | clear () | 清除串中的所有字符 |
| 统计 | size () / length() | 返回串中字符的数目 |
| | max_size () | 返回串允许的最大长度 |

得到字符串中的字符

- 重载下标运算符[]

```
char& string::operator [] (int n);
```

- 按字符定位下标

```
int string::find(char c, int start=0);
```

- 反向寻找，定位尾部出现的字符

```
int string::rfind(char c, int pos=0);
```

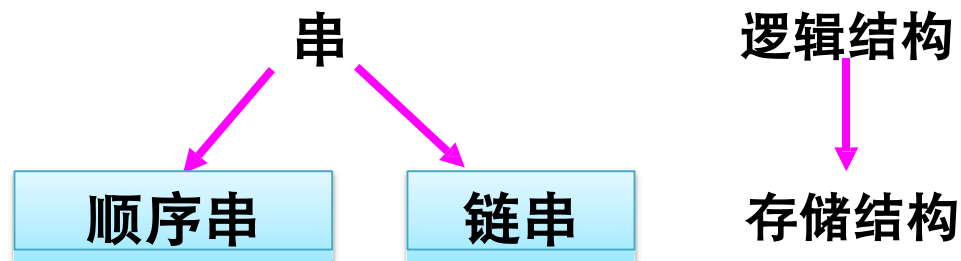
8.2 串的存储结构

串的顺序存储

- 对串长变化不大的字符串，有三种处理方案
 1. 用 `S[0]` 作为记录串长的存储单元 (Pascal)
 - 缺点：限制了串的最大长度不能超过256
 2. 为存储串的长度，另辟一个存储的地方
 - 缺点：串的最大长度一般是静态给定的，不是动态申请数组空间
 3. 用一个特殊的末尾标记 `'\0'` (C/C++)
 - 例如：C/C++ 语言的 string 函数库 (`#include <string.h>`) 采用这一存储结构
 - `'\0'` 的 ASCII 字符表中编号为 0，等价于常量 `NULL`、数字 `0`、常量 `false`

串的存储结构

串中元素逻辑关系与线性表的相同，串可以采用与线性表相同的存储结构。



串的顺序存储

串的顺序存储（顺序串）有两种方法：

- 每个单元（如4个字节）只存一个字符，称为非紧缩格式（其存储密度小）。
- 每个单元存放多个字符，称为紧缩格式（其存储密度大）。

| | | | | |
|------|---|--|--|--|
| 1001 | A | | | |
| 1002 | B | | | |
| 1003 | C | | | |
| 1004 | D | | | |
| 1005 | E | | | |
| 1006 | F | | | |
| 1007 | G | | | |
| 1008 | H | | | |
| 1009 | I | | | |
| 100a | J | | | |
| 100b | K | | | |
| 100c | L | | | |
| 100d | M | | | |
| 100e | N | | | |

非紧缩格式示例

| | | | | |
|------|---|---|---|---|
| 1001 | A | B | C | D |
| 1002 | E | F | G | H |
| 1003 | I | J | K | L |
| 1004 | M | N | | |

紧缩格式示例

一个单元



串的顺序存储

对于非紧缩格式的顺序串，其类型定义如下：

```
#define MaxSize 100
```

```
typedef struct
```

```
{   char data[MaxSize];
```

```
    int length;
```

```
} SqString;
```

用来存储字符串

用来存储字符串长度

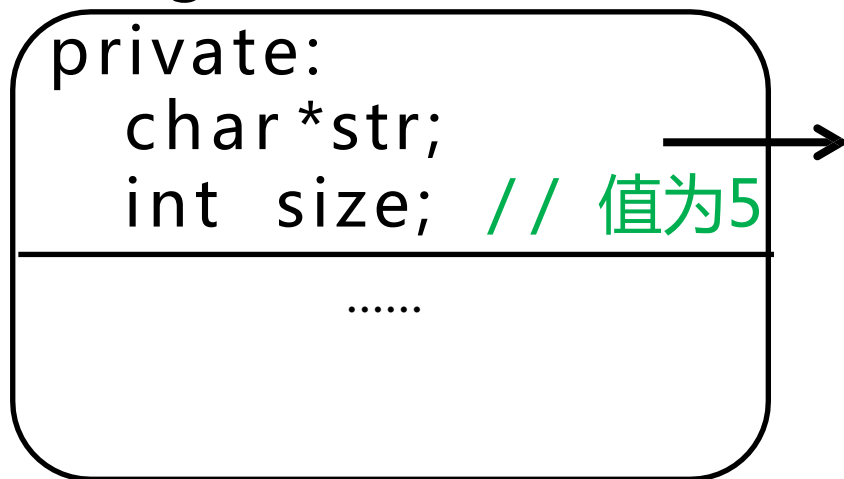
顺序串中实现串的基本运算与顺序表的基本运算类似。

串的顺序存储

```
private: // 具体实现的字符串存储结构
char *str; // 字符串的数据表示
int size; // 串的当前长度
```

例如，

```
String s1 = "Hello";
```



| | | | | | |
|---|---|---|---|---|----|
| H | e | l | l | o | \0 |
| 0 | 1 | 2 | 3 | 4 | 5 |

串的顺序存储演示

静态存储分配的顺序串

bobo

本演示将字符串以'\0'为串值终结保存在一个定长的数组中。

注意：

【定长的数组大小】输入数组的大小（建议输入小于20的整数）

【要保存的字符串】输入字符串（建议串长小于定长数组大小）

已完成。请
点击【返回】

```
#define MaxStrSize  ——请输入定长数组的大小  
typedef char SeqString[MaxString];
```

请输入要保存的字符串(S)

▶ 确定

▶ 重新开始

◀ 返回

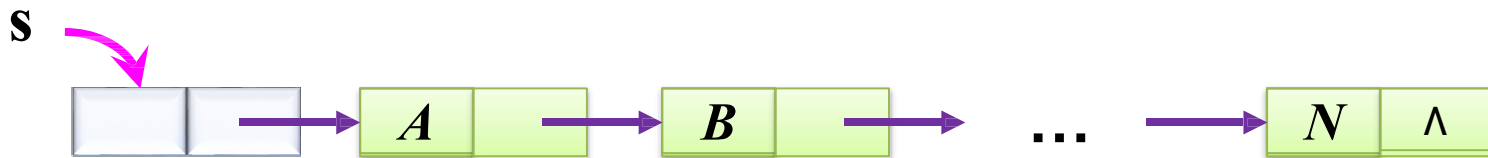
串的链式存储

链串的组织形式与一般的链表类似。

链串中的一个节点可以存储多个字符。通常将链串中每个节点所存储的字符个数称为节点大小。



节点大小为4的链串

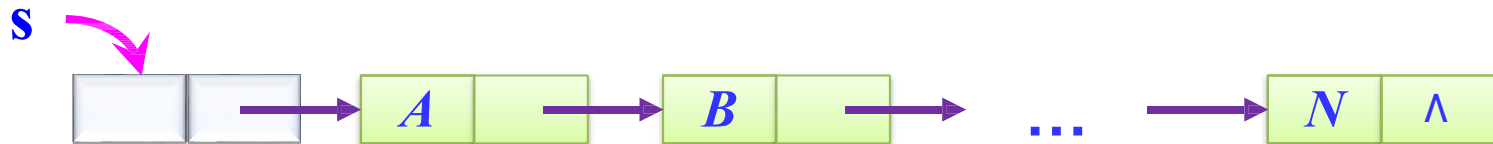


节点大小为1的链串

串的链式存储

链串节点大小1时，链串的节点类型定义如下：

```
typedef struct snode
{
    char data;
    struct snode *next;
} LiString;
```



串的链式存储

链串中实现串的基本运算与单链表的基本运算类似。

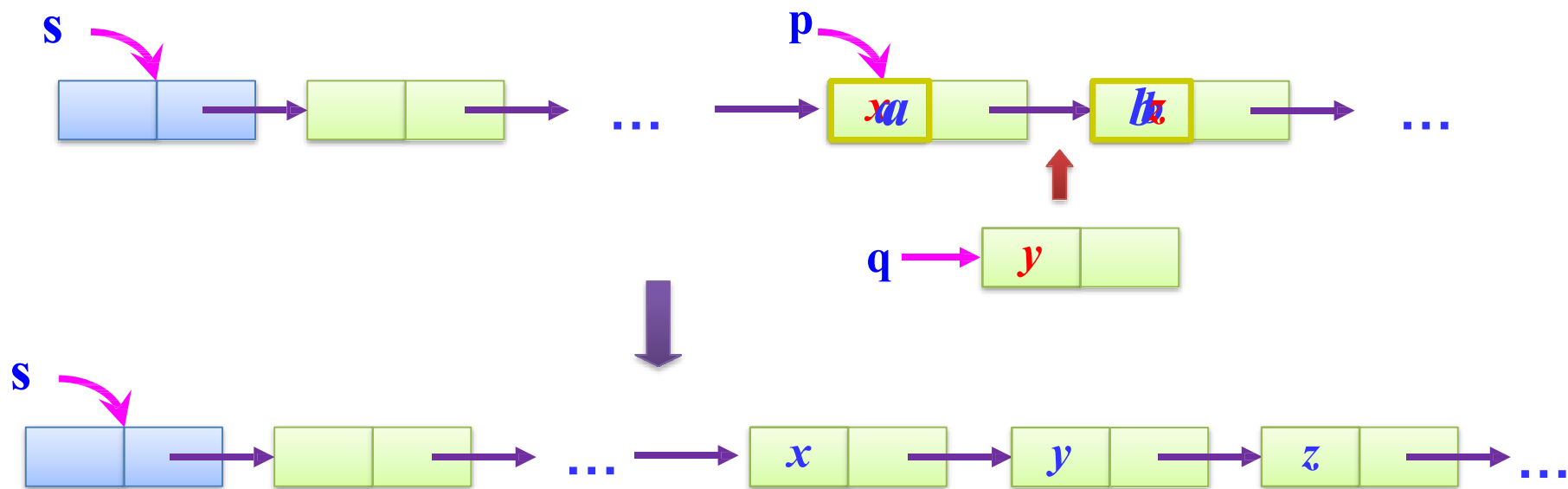
【例4-2】 在链串中，设计一个算法把最先出现的子串 “*ab*”
改为 “*xyz*”。

❶ 查找： $p \rightarrow \text{data} = 'a' \ \&\& \ p \rightarrow \text{next} \rightarrow \text{data} = 'b'$



串的链式存储

② 替换



串的链式存储

```
void Repl(LiString *&s)
{
    LiString *p=s->next,*q;
    int find=0;
    while (p->next!=NULL && find==0)           //查找ab子串
    {
        if (p->data==' a' && p->next->data=='b')
        {
            p->data='x'; p->next->data='z';
            q=(LiString *)malloc(sizeof(LiString));
            q->data='y'; q->next=p->next; p->next=q;
            find=1;
        }
        else p=p->next;
    }
}
```

替换为xyz

算法的时间复杂度为 $O(n)$ 。

串的运算的实现

- 串长函数
 - `int strlen(char *s);`
- 串复制
 - `char *strcpy(char *s1, char*s2);`
- 串拼接
 - `char *strcat(char *s1, char *s2);`
- 串比较
 - `int strcmp(char *s1, char *s2);`

串运算的实现

// 求字符串的长度

```
int strlen(char d[ ]) {  
    int i = 0;  
    while (d[i] != '\0')  
        i++;  
    return i;  
}
```

// 字符串的复制

```
char *strcpy(char *d, char *s) {  
    int i = 0;  
    while (s[i] != '\0') {  
        d[i] = s[i];    i++;  
    }  
    d[i] = '\0';  
    return d;  
}
```


串的比较

【例4-1】设计顺序串上实现串比较运算Strcmp(s,t)的算法。

例如：

"ab" < "abcd"

"abcd" < "abd"

解： 算法思路如下：

(1) 比较s和t两个串共同长度范围内的对应字符：

- ① 若s的字符 > t的字符，返回1；
- ② 若s的字符 < t的字符，返回-1；
- ③ 若s的字符 = t的字符，按上述规则继续比较。

(2) 当(1)中对应字符均相同时，比较s和t的长度：

- ① 两者相等时，返回0；
- ② s的长度 > t的长度，返回1；
- ③ s的长度 < t的长度，返回-1。

串的比较

// 字符串的比较

```
int strcmp(const char *s1, const char *s2) {  
    int i = 0;  
    while (s2[i] != '\0' && s1[i] != '\0' ) {  
        if (s1[i] > s2[i])  
            return 1;  
        else if (s1[i] < s2[i])  
            return -1;  
        i++;  
    }  
    if (s1[i] == '\0' && s2[i] != '\0')  
        return -1;  
    else if s2[i] == '\0' && s1[i] != '\0')  
        return 1;  
    return 0;  
}
```

串的比较

```
int strcmp_1(char *s1, char *s2) {  
    int i;  
    for (i = 0; s1[i] == s2[i]; ++i) {  
        if(s1[i] == '\0' && s2[i] == '\0')  
            return 0;           // 两个字符串相等  
    }  
    // 不等, 比较第一个不同的字符  
    return (s1[i]-s2[i]) / abs(s1[i]-s2[i]);  
}
```

其他函数

```
// 构造函数(constructor)
String::String(char *s) {
    // 先要确定新创字符串实际需要的存储空间，s的类型为(char *)，
    // 作为新创字符串的初值。确定s的长度，用标准字符串函数
    // strlen(s)计算长度
    size = strlen(s);

    // 然后，在动态存储区域开辟一块空间，用于存储初值s，把结束
    // 字符也包括进来
    str = new char [size + 1];
    // 开辟空间不成功时，运行异常，退出
    assert(str != NULL);

    // 用标准字符串函数strcpy，将s完全复制到指针str所指的存储空间
    strcpy(str, s);
}
```

其他函数

// 析构函数

```
String::~String() { // 必须释放动态存储空间  
    delete [] str;
```

// 赋值函数

```
String String::operator= (String& s) {  
    // 参数 s 将被赋值到本串。  
    // 若本串的串长和s的串长不同，则应该释放本串的  
    // str存储空间，并开辟新的空间  
    if (size != s.size) {  
        delete [] str ;           // 释放原存储空间  
        str = new char [s.size+1];  
        // 若开辟动态存储空间失败，则退出正常运行  
        assert(str != NULL) ;  
        size = s.size;  
    }  
    strcpy(str , s.str );  
    // 返回本实例，作为String类的一个实例  
    return *this;  
}
```

习题

- 设 $S1, S2$ 为串，请给出使 $S1+S2 == S2+S1$ 成立的所有可能的条件（其中 $+$ 为连接运算）？
- 设计一个算法来实现字符串逆序存储，要求不另设串存储空间？

8.3 串的模式匹配

串匹配

给定一段文本，从中找出某个指定的关键字。

例如从一本 Thomas Love Peacock 写于十九世纪的小说
《 Headlong Hall 》中找到那个最长的单词

osseocarnisanguineoviscericartilaginonervomedullary

或者从古希腊喜剧 《 Assemblywomen 》 中找到一道菜 的名字

*Lopadotemachoselachogaleokraniroleipsanodrimhypotrim
matosilphioparaomelitokatakechymenokichlepikeossyphoph
attoperisteralaektryonoptekephalliokigklopeleiolagoiosiraio
baphetraganopterygon*

串匹配

- 模式匹配 (pattern matching)
 - 一个目标对象 T (字符串)
 - (pattern) P (字符串)

在目标 T 中寻找一个给定的模式P的过程
- 应用
 - 文本编辑时的特定词、句的查找
 - UNIX/Linux: sed, awk, grep
 - DNA 信息的提取
 - 确认是否具有某种结构
 - ...
- 模式集合

串匹配

- 用给定的模式 P ，在目标字符串 T 中搜索与模式 P 全 同的一个子串，并求出 T 中第一个和 P 全同匹配的子 串（简称为“**配串**”），返回其首字符位置

$$\begin{array}{ccccccccccc} \textcolor{red}{T} & t_0 & t_1 & \dots & t_i & t_{i+1} & t_{i+2} & \dots & t_{i+m-2} & t_{i+m-1} & \dots & t_{n-1} \\ & & & & \parallel & \parallel & \parallel & & \parallel & \parallel & & \\ \textcolor{red}{P} & & & & p_0 & p_1 & p_2 & \dots & p_{m-2} & p_{m-1} & & \end{array}$$

为使模式 P 与目标 T 匹配，必须满足

$$p_0 \ p_1 \ p_2 \ \dots \ p_{m-1} \quad = \quad t_i \ t_{i+1} \ t_{i+2} \ \dots \ t_{i+m-1}$$

串匹配

给定一段文本： $\text{string} = s_0s_1 \dots s_{n-1}$ 给

定一个模式： $\text{pattern} = p_0p_1 \dots p_{m-1}$

求 pattern 在 string 中出现的位置

$\text{Position PatternMatch}(\text{char} * \text{string}, \text{char} * \text{pattern})$

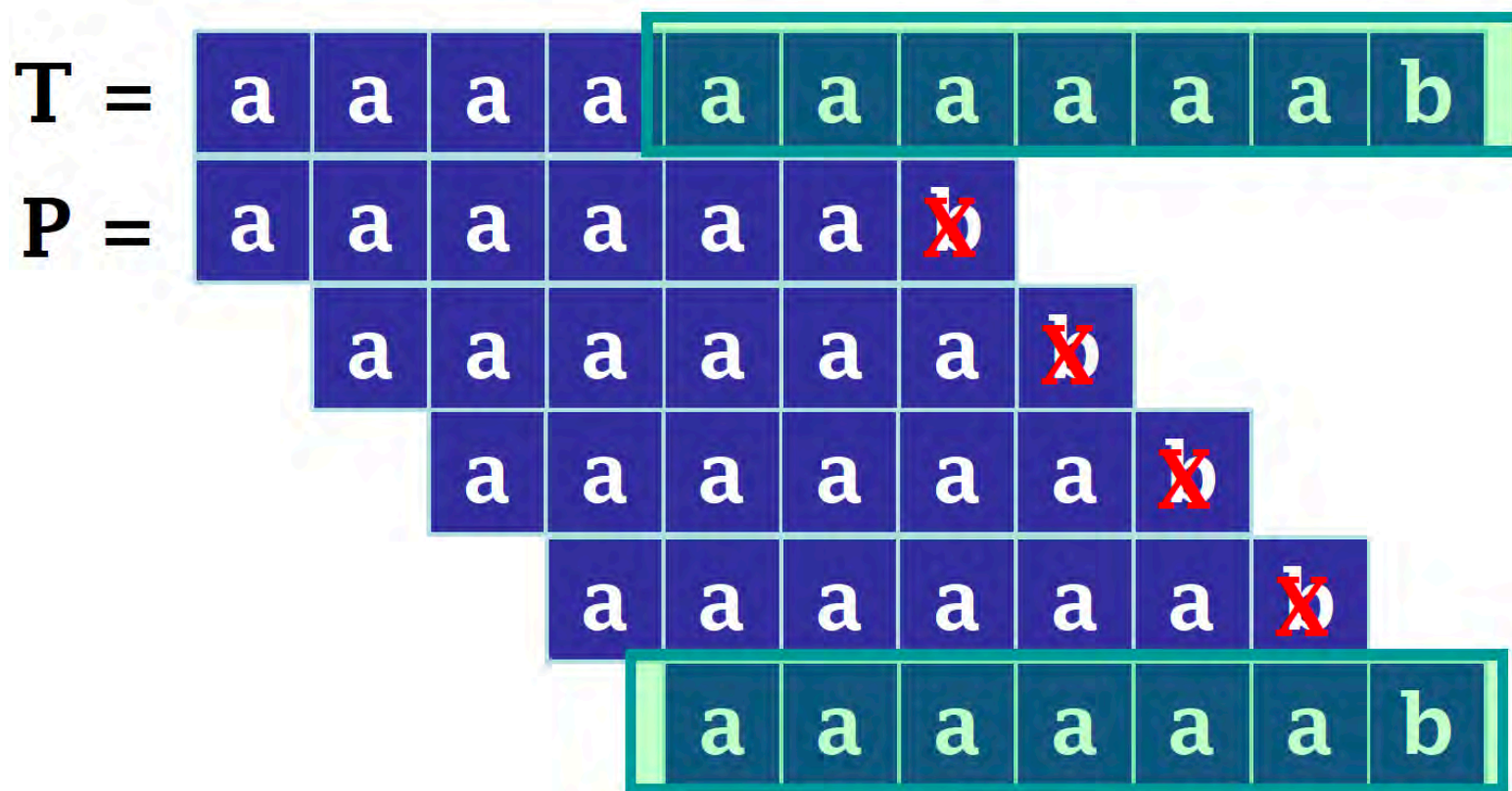
解决模式匹配问题的算法

- 朴素（称为“Brute Force”，也称“Naive”）
- Knuth-Morrit-Pratt（KMP 算法）
-

朴素串匹配算法

- 设 $T = t_0 t_1, t_2, \dots, t_{n-1}$, $P = p_0, p_1, \dots, p_{m-1}$
 - i 为 T 中字符的下标, j 为 P 中字符的下标
 - 匹配成功 ($p_0 = t_i, p_1 = t_{i+1}, \dots, p_{m-1} = t_{i+m-1}$)
 - 即, $T.\text{substr}(i, m) == P.\text{substr}(0, m)$
 - 匹配失败 ($p_j \neq t_i$) 时,
 - 将 P 右移再行比较
 - 尝试所有的可能情况

朴素串匹配示例



简单实现

■ 方法1 : C的库函数 strstr

`char *strstr(char *string, char *pattern)`

```
#include <stdio.h>
#include <string.h>
typedef char* Position;
```

```
int main()
{
    char string[] = "This is a simple example.";
    char pattern[] = "simple";
    Position p = strstr(string, pattern);
    if ( p == NotFound ) printf("Not Found.\n");
    else printf("%s\n", p);
    return 0;
}
```

simple example.

Process exited after 0.665 seconds with return value: 0
请按任意键继续. . .

简单实现

■ 方法1 : C的库函数 strstr

`char *strstr(char *string, char *pattern)`

```
#include <stdio.h>
#include <string.h>
typedef char* Position;
#define NotFound NULL
int main()
{
    char string[] = "This is a simple example.";
    char pattern[] = "simple";
    Position p = strstr(string, pattern);
    printf("%s\n", p);
    return 0;
}
```

simple example.

Process exited after 0.665 seconds with return value: 0
请按任意键继续. . .

串查找索引

关键代码

```
int Index(String S, String T, int pos)
{
    int n,m,i;
    String sub;
    if (pos > 0)
    {
        n = StrLength(S);
        m = StrLength(T);
        i = pos;
        while (i <= n-m+1)
        {
            SubString(sub, S, i, m);
            if (StrCompare(sub,T) != 0)
                ++i;
            else
                return i;
        }
    }
    return 0;
}
```

数据变化

| | | | | | | | | | |
|---|---|---|---|---|---|--------------|---|---|---|
| g | o | o | d | g | o | o | g | l | e |
| g | o | o | g | l | e | 1. $n-m+1=5$ | | | |

串查找索引

关键代码

```
int Index(String S, String T, int pos)
{
    int n,m,i;
    String sub;
    if (pos > 0)
    {
        n = StrLength(S);
        m = StrLength(T);
        i = pos;
        while (i <= n-m+1)
        {
            SubString(sub, S, i, m);
            if (StrCompare(sub,T) != 0)
                ++i;
            else
                return i;
        }
    }
    return 0;
}
```

数据变化



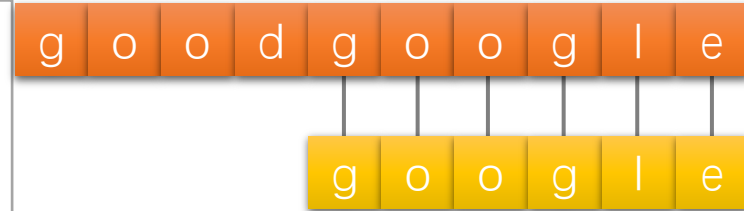
sub=goodgo, i=1, m=6

串查找索引

关键代码

```
int Index(String S, String T, int pos)
{
    int n,m,i;
    String sub;
    if (pos > 0)
    {
        n = StrLength(S);
        m = StrLength(T);
        i = pos;
        while (i <= n-m+1)
        {
            SubString(sub, S, i, m);
            if (StrCompare(sub,T) != 0)
                ++i;
            else
                return i;
        }
    }
    return 0;
}
```

数据变化



sub=google, i=5, m=6

朴素串匹配算法演示

关键代码

```
int Index(String S, String T, int pos)
{
    int i = pos;
    int j = 1;
    while (i <= S[0] && j <= T[0])
    {
        if (S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            i = i-j+2;
            j = 1;
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

数据变化



朴素串匹配算法演示

关键代码

```
int Index(String S, String T, int pos)
{
```

```
    int i = pos;
```

```
    int j = 1;
```

```
    while (i <= S[0] && j <= T[0])
```

```
    {
```

```
        if (S[i] == T[j])
```

```
        {
```

```
            ++i;
```

```
            ++j;
```

```
        }
```

```
        else
```

```
        {
```

```
            i = i - j + 2;
```

```
            j = 1;
```

```
        }
```

```
    }
```

```
    if (j > T[0])
```

```
        return i - T[0];
```

```
    else
```

```
        return 0;
```

```
}
```

数据变化



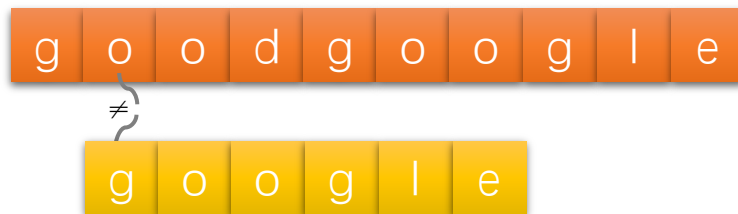
pos=1, i=2, j=1, S[0]=10, T[0]=6, S[4]=d, T[4]=g

朴素串匹配算法演示

关键代码

```
int Index(String S, String T, int pos)
{
    int i = pos;
    int j = 1;
    while (i <= S[0] && j <= T[0])
    {
        if (S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            i = i - j + 2;
            j = 1;
        }
    }
    if (j > T[0])
        return i - T[0];
    else
        return 0;
}
```

数据变化



pos=1, i=3, j=1, S[0]=10, T[0]=6, S[2]=o, T[1]=g

朴素串匹配算法演示

关键代码

```
int Index(String S, String T, int pos)
```

```
{
```

```
    int i = pos;
```

```
    int j = 1;
```

```
    while (i <= S[0] && j <= T[0])
```

```
    {
```

```
        if (S[i] == T[j])
```

```
        {
```

```
            ++i;
```

```
            ++j;
```

```
        }
```

```
        else
```

```
        {
```

```
            i = i-j+2;
```

```
            j = 1;
```

```
        }
```

```
    }
```

```
    if (j > T[0])
```

```
        return i-T[0];
```

```
    else
```

```
        return 0;
```

```
}
```

数据变化



pos=1, i=4, j=1, S[0]=10, T[0]=6, S[3]=o, T[1]=g

朴素串匹配算法演示

关键代码

```
int Index(String S, String T, int pos)
```

```
{
```

```
    int i = pos;
```

```
    int j = 1;
```

```
    while (i <= S[0] && j <= T[0]) pos=1, i=11, j=7, S[0]=10, T[0]=6, S[4]=g, T[1]=g, i-T[0]=5
```

```
    {
```

```
        if (S[i] == T[j])
```

```
        {
```

```
            ++i;
```

```
            ++j;
```

```
        }
```

```
    else
```

```
    {
```

```
        i = i-j+2;
```

```
        j = 1;
```

```
    }
```

```
    }
```

```
    if (j > T[0])
```

```
        return i-T[0];
```

```
    else
```

```
        return 0;
```

```
}
```

数据变化



朴素串匹配算法演示

朴素串匹配算法过程示意

Cooling

目标串

模式串

请输入目标串 **abcbbcbabc**

请输入模式串 **bb**



朴素串匹配算法效率分析

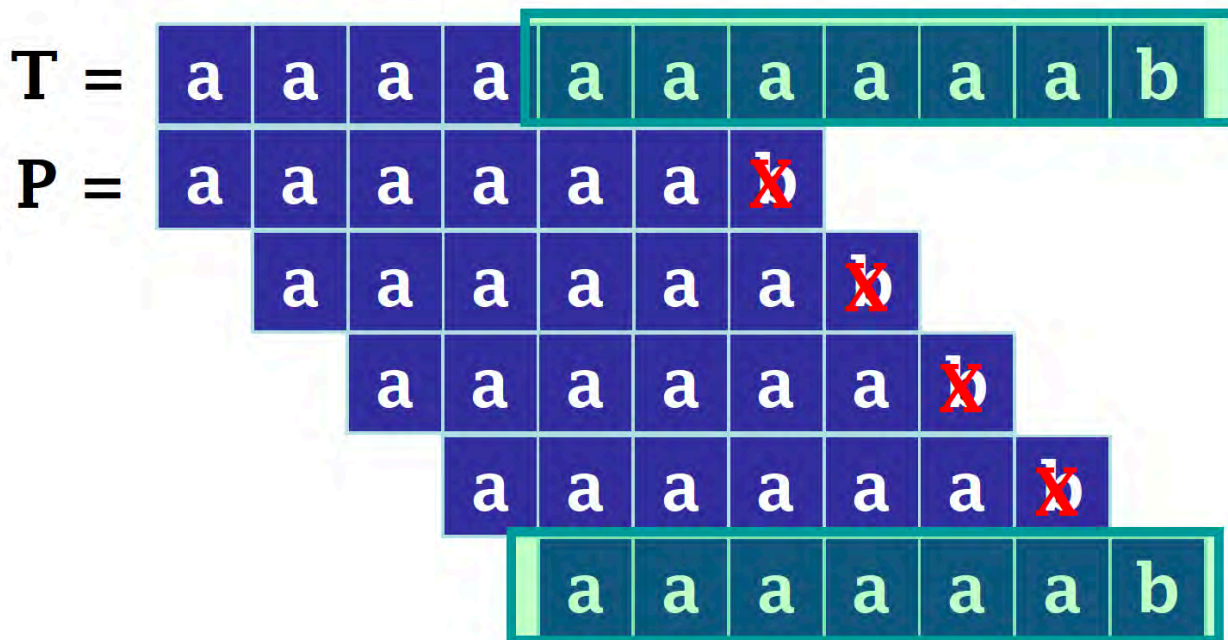
- 假定目标 T 的长度为 n ，模式 P 长度为 m ， $m \leq n$
 - 在最坏的情况下，每一次循环都不成功，则一共要进行比较 $(n - m + 1)$ 次
 - 每一次“相同匹配”比较所耗费的时间，是 P 和 T 逐个字符比较的时间，最坏情况下，共 m 次
 - 因此，整个算法的最坏时间开销估计为

$$O(m \cdot n)$$

最坏情况

- 模式与目标的每一个长度为 m 的子串进行比较

- 目标形如 $a^{n-1}X$
- 模式形如 $a^{m-1}b$



- 总比较次数：
 - $m(n - m + 1)$
- 时间复杂度：
 - $O(mn)$

简单实现

■ 方法1：C的库函数 strstr

`char *strstr(char *string, char *pattern)`

`string` = "aa"
`pattern` = "aab"
 └─┬─┘
 m

Diagram illustrating the strstr function parameters and their lengths:

- `string` is a long string of 'a's, with a bracket above it labeled n .
- `pattern` is a short string "aab", with a bracket below it labeled m .

$$T = O(n \cdot m)$$

简单改进

■ 方法2：从末尾开始比

String 

pattern 

```
string = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"  
pattern = "aab"
```

$$T = O(n)$$

```
string = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"  
pattern = "baa"
```



朴素串匹配示例

| | | | | | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T= | a | b | a | b | a | b | a | b | a | b | a | b | b |
| P= | a | b | a | b | a | b | X | | | | | | |
| | | X | b | a | b | a | b | b | | | | | |
| | | | a | b | a | b | a | b | X | | | | |
| | | | | X | b | a | b | a | b | b | | | |
| | | | | | a | b | a | b | a | b | X | | |
| | | | | | | X | b | a | b | a | b | b | |
| | | | | | | | a | b | a | b | a | b | b |

朴素串匹配示例

T =

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | a | b | c | d | e | f | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

P =

| | | | | | | |
|---|---|---|---|---|---|--------------|
| a | b | c | d | e | f | f |
|---|---|---|---|---|---|--------------|

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | b | c | d | e | f | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

 ✓

最好情况-找到

- 在目标的前 m 个位置上找到模式，设 $m = 5$

AAAAA AAAAAAAAAAAAAAAAAAAAH

AAAAA

5次比较

- 总比较次数： m
- 时间复杂度： $O(m)$

最好情况-没找到

- 总是在第一个字符上不匹配

- 总比较次数：

- $n - m + 1$

- 时间复杂度：

- $O(n)$

A A A A A A A A A A A A A A A A A H

O O O O H

1次比较

A A A A A A A A A A A A A A A A A H

O O O O H

1次比较

A A A A A A A A A A A A A A A A A H

O O O O H

1次比较

A A A A A A A A A A A A A A A A A H

O O O O H

1次比较

.....

A A A A A A A A A A A A A A A A A H

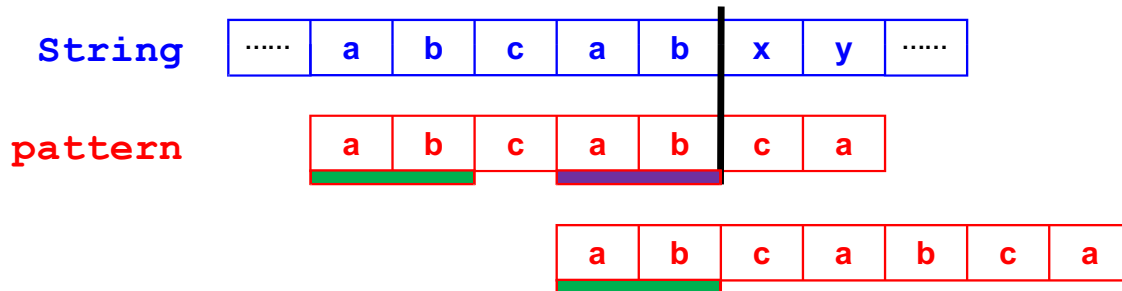
O O O O H 1次比较

8.4 KMP算法实现

大师改进

■ 方法3 : KMP (Knuth、Morris、Pratt) 算法

$$T = O(n+m)$$



$$match(j) = \begin{cases} \text{满足 } p_0 \cdots p_i = p_{j-i} \cdots p_j \text{ 的} \underline{\text{最大 } i} (< j) \\ -1 & \text{如果这样的 } i \text{ 不存在} \end{cases}$$

| | | | | | | | | | | |
|---------|----|----|----|---|---|---|---|----|---|---|
| pattern | a | b | c | a | b | c | a | c | a | b |
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| match | -1 | -1 | -1 | 0 | 1 | 2 | 3 | -1 | 0 | 1 |

KMP算法

无回溯匹配

- 匹配过程中，一旦 p_j 和 t_i 比较不等时，即
$$P.substr(1, j-1) == T.substr(i-j+1, j-1)$$
但 $p_j \neq t_i$
 - 该用 P 中的哪个字符 p_k 和 t_i 进行比较？
 - 确定右移的位数
 - 显然有 $k < j$ ，且不同的 j ，其 k 值不同
- Knuth-Morrit-Pratt (KMP)算法
 - k 值仅仅依赖于模式 P 本身，而与目标对象 T 无关

KMP算法

T = a b c d e f a b c d e f f

P = a b c d e f f
 a b c d e f f

T t_0 t_1 ... t_{i-j-1} t_{i-j} t_{i-j+1} t_{i-j+2} ... t_{i-2} t_{i-1} t_i ... t_{n-1}
 || || || || ||
 X

P p_0 p_1 p_2 ... p_{j-2} p_{j-1} p_j

则有 $t_{i-j} t_{i-j+1} t_{i-j+2} \dots t_{i-1} = p_0 p_1 p_2 \dots p_{j-1}$ (1)

朴素下一趟 $p_0 p_1 \dots p_{j-2} p_{j-1}$

如果 $p_0 p_1 \dots p_{j-2} \neq p_1 p_2 \dots p_{j-1}$ (2)

则立刻可以断定

$p_0 p_1 \dots p_{j-2} \neq t_{i-j+1} t_{i-j+2} \dots t_{i-1}$

(朴素匹配的)下一趟一定不匹配，可以跳过去

$p_0 p_1 \dots p_{j-2} p_{j-1}$

KMP算法

同样，若 $p_0 p_1 \dots p_{j-3} \neq p_2 p_3 \dots p_{j-1}$
 则再下一趟也不匹配，因为有

$p_0 p_1 \dots p_{j-3} \neq t_{i-j+2} t_{i-j+3} \dots t_{i-1}$
 直到对于某一个“ k ”值（首尾串长度），使得

且 $p_0 p_1 \dots p_k \neq p_{j-k-1} p_{j-k} \dots p_{j-1}$
 $p_0 p_1 \dots p_{k-1} = p_{j-k} p_{j-k+1} \dots p_{j-1}$

模式右滑 $j-k$ 位

| | | | | |
|-------------|-------------|---------|-------------|-------|
| t_{i-k} | t_{i-k+1} | \dots | t_{i-1} | t_i |
| \parallel | \parallel | | \parallel | × |
| p_{j-k} | p_{j-k+1} | \dots | p_{j-1} | p_j |
| \parallel | \parallel | | \parallel | ? |
| p_0 | p_1 | \dots | p_{k-1} | p_k |

$T =$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | a | b | c | d | e | f | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

 $P =$

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | c | d | e | f | f |
| | | c | d | e | f | |
| | | a | b | c | d | e |



则 $p_0 p_1 \dots p_{k-1} = t_{i-k} t_{i-k+1} \dots t_{i-1}$

特征向量

设模式 P 由 m 个字符组成，记为

$$P = p_0 p_1 p_2 p_3 \dots p_{m-1}$$

令 **特征向量** N 用来表示模式 P 的字符分布特征，简称 **N 向量** 由 m 个特征数 $n_0 \dots n_{m-1}$ 整数组成，记为

$$N = n_0 n_1 n_2 n_3 \dots n_{m-1}$$

N 在很多文献中也称为 `next` 数组，每个 n_j 对应 `next` 数组中的元素 `next[j]`

特征向量

· P 第 j 个位置的特征数 n_j , 首尾串最长的 k

- 首串 : $p_0 \ p_1 \quad \dots \ p_{k-2} \ p_{k-1}$

- 尾串 : $p_{j-k} \ p_{j-k+1} \quad \dots \ p_{j-2} \ p_{j-1}$

$$\text{next}[j] = \begin{cases} -1, & j=0 \text{时候} \\ \max\{k: 0 < k < j \ \& \ P[0\dots k-1] = P[j-k\dots j-1]\}, & \text{首尾配串最长}k \\ 0, & \text{其他} \end{cases}$$

特征向量

P =

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| P | a | a | a | a | b | a | a | a | a | c |

N =

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|----|---|---|---|---|---|---|---|---|---|
| N | -1 | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 4 |

X (应为3)

T =

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|---|
| T | a | a | b | a | a | a | a | a | a | b | a | a | a | a | c | b |

P =

| | | | | | | | | | | | | | | | |
|---|---|--------------|---|---|---|---|--------------|---|---|---|---|---|--|--|--|
| a | a | X | a | b | a | a | a | a | c | | | | | | |
| | | | a | a | a | a | X | a | a | a | a | c | | | |

i=2, j=1, N[j]=0

i=7, j=4, N[4]=~~1~~
X

错过了!

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | b | a | a | a | a | c |
|---|---|---|---|---|---|---|---|---|---|

匹配示例

$P =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | b |

$N =$

| | | | | | | |
|----|---|---|---|---|---|---|
| -1 | 0 | 0 | 1 | 2 | 3 | 4 |
|----|---|---|---|---|---|---|

$T =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| a | b | a | b | a | b | a | b | a | b | a | b | b |

$P =$

| | | | | | | |
|---|---|---|---|---|---|--------------|
| a | b | a | b | a | b | x |
|---|---|---|---|---|---|--------------|

$i=6, j=6, N[j]=4$

| | | | | | | |
|---|---|---|---|---|---|--------------|
| a | b | a | b | a | b | x |
|---|---|---|---|---|---|--------------|

$i=8, j=6, N[j]=4$

| | | | | | | |
|---|---|---|---|---|---|--------------|
| a | b | a | b | a | b | x |
|---|---|---|---|---|---|--------------|

$i=10, j=6, j'=4$

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | b |
|---|---|---|---|---|---|---|

✓

算法框架

- 特征数 n_j ($j > 0, 0 \leq n_{j+1} \leq j$) 是递归定义的, 定义如下:
 1. $n_0 = -1$, 对于 $j > 0$ 的 n_{j+1} , 假定已知前一位置的特征数 n_j , 令 $k = n_j$;
 2. 当 $k \geq 0$ 且 $p_j \neq p_k$ 时, 则令 $k = n_k$; 让步骤2循环直到条件不满足
 3. $n_{j+1} = k + 1$; // 此时, $k == -1$ 或 $p_j == p_k$

特征向量求法

$N =$

| | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$P =$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | b | a | a | a | a | c |
|---|---|---|---|---|---|---|---|---|---|

$j = 9$ $k = 0$

首串→

| |
|---|
| a |
|---|

首串→

| | |
|---|---|
| a | a |
|---|---|

首串→

| | | |
|---|---|---|
| a | a | a |
|---|---|---|

首串→

| |
|---|
| a |
|---|

首串→

| | |
|---|---|
| a | a |
|---|---|

首串→

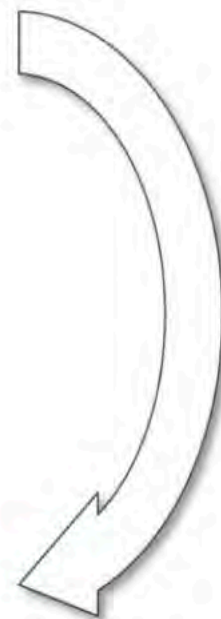
| | | |
|---|---|---|
| a | a | a |
|---|---|---|

首串→

| | | | |
|---|---|---|---|
| a | a | a | a |
|---|---|---|---|

模式右滑j-k位

$$\begin{array}{ccccccc}
 t_{i-j} & t_{i-j+1} & t_{i-j+2} & \dots & t_{i-k} & t_{i-k+1} & \dots & t_{i-1} & t_i \\
 \parallel & \parallel & \parallel & & \parallel & \parallel & & \parallel & \times \\
 p_0 & p_1 & p_2 & \dots & p_{j-k} & p_{j-k+1} & \dots & p_{j-1} & p_j \\
 & & & & \parallel & \parallel & & \parallel & ? \\
 & & & & p_0 & p_1 & \dots & p_{k-1} & p_k
 \end{array}$$



$$p_0 p_1 \dots p_{k-1} = t_{i-k} t_{i-k+1} \dots t_{i-1}$$

$$t_i \neq p_j, \quad p_j == p_k?$$

KMP匹配

| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| P | a | b | c | a | a | b | a | b | c |
| K | | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |

目标 $a a b c b a b c a a b c a a b a b c$
 $a \times b c a a b a b c$ $N[1] = 0$
 $a b c \times a b a b c$ $N[3] = 0$
这行冗余 $\times b c a a b a b c$ $N[0] = -1$
 $a b c a a b \times b c$
 $a b c a a b a b c$ $N[6] = 2$

上面 $P[3] = P[0]$, $P[3] \neq T[4]$, 再比冗余

[illegible]

字符串匹配是一个非常基本的操作

KMP模式匹配样例，next数组值推导

关键代码

T=abcabx

```
void get_next(String T, int *next)
```

```
{
```

```
    int i,k;
```

```
    i=1;
```

```
    k=0;
```

```
    next[1]=0;
```

```
    while (i<T[0])
```

```
    {
```

```
        if(k == 0 || T[i] == T[k])
```

```
        {
```

```
            ++i;
```

```
            ++k;
```

```
            next[i] = k;
```

```
        }
```

```
        else
```

```
            k = next[k];
```

```
    }
```

```
}
```

T[0]=6, i=1, k=0, next[1]=0, next[2]=0

T[0]=6, i=2, k=1, next[1]=0, next[2]=1

i=2 k=1 next=010000

i=2 k=0 next=010000

i=3 k=1 next=011000

i=3 k=0 next=011000

i=4 k=1 next=011100

i=5 k=2 next=011120

i=6 k=3 next=011123

KMP模式匹配样例，next数组值推导

关键代码

```
void get_next(String T, int *next)
```

```
{
```

```
    int i,k;
```

```
    i=1;
```

```
    k=0;
```

```
    next[1]=0;
```

```
    while (i<T[0])
```

```
    {
```

```
        if(k == 0 || T[i] == T[k])
```

```
        {
```

```
            ++i;
```

```
            ++k;
```

```
            next[i] = k;
```

```
        }
```

```
        else
```

```
            k = next[k];
```

```
    }
```

```
}
```

T=ababaaaba

T[0]=6, i=1, k=0, next[1]=0, next[2]=0

T[0]=6, i=2, k=1, next[1]=0, next[2]=1

i=2 k=1 next=010000000

i=2 k=0 next=010000000

i=3 k=1 next=011000000

i=4 k=2 next=011200000

i=5 k=3 next=011230000

i=6 k=4 next=011234000

i=6 k=2 next=011234000

i=6 k=1 next=011234000

i=7 k=2 next=011234200

i=7 k=1 next=011234200

i=8 k=2 next=011234220

i=9 k=3 next=011234223

KMP模式匹配样例，next数组值推导

关键代码

```
void get_next(String T, int *next)
```

```
{
```

```
    int i,k;
```

```
    i=1;
```

```
    k=0;
```

```
    next[1]=0;
```

```
    while (i<T[0])
```

```
    {
```

```
        if(k == 0 || T[i] == T[k])
```

```
        {
```

```
            ++i;
```

```
            ++k;
```

```
            next[i] = k;
```

```
        }
```

```
        else
```

```
            k = next[k];
```

```
    }
```

```
}
```

T=aaaaaaab

T[0]=6, i=1, k=0, next[1]=0, next[2]=0

T[0]=6, i=2, k=1, next[1]=0, next[2]=1

i=2 k=1 next=010000000

i=3 k=2 next=012000000

i=4 k=3 next=012300000

i=5 k=4 next=012340000

i=6 k=5 next=012345000

i=7 k=6 next=012345600

i=8 k=7 next=012345670

i=9 k=8 next=012345678

KMP算法实现

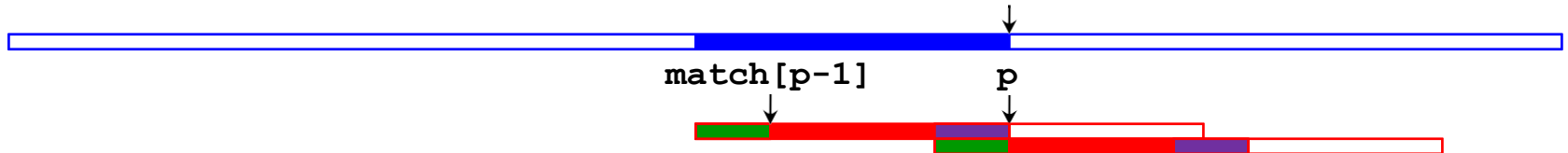


```
#include <stdio.h>
#include <string.h>

typedef int Position;
#define NotFound -1

int main()
{
    char string[] = "This is a simple example.";
    char pattern[] = "simple";
    Position p = KMP(string, pattern);
    if ( p == NotFound ) printf("Not Found.\n");
    else printf("%s\n", string+p);
    return 0;
}
```

KMP算法实现



```
Position KMP( char *string, char *pattern )
{
    int n = strlen(string);
    int m = strlen(pattern);
    int s, p, *match;

    match = (int *)malloc(sizeof(int) * m);
    BuildMatch(pattern, match);
    s = p = 0;
    while (s < n && p < m) {
        if (string[s] == pattern[p]) { s++; p++; }
        else if (p > 0) p = match[p-1] + 1;
        else s++;
    }
    return (p == m) ? (s - m) : NotFound;
}
```

KMP算法实现

$$T = O(n+m+T_m)$$

```
Position KMP( char *string, char *pattern )
{
    int n = strlen(string);      /* O(n) */
    int m = strlen(pattern);     /* O(m) */
    int s, p, *match;
    if ( n < m ) return NotFound;
    match = (int *)malloc(sizeof(int) * m);
    BuildMatch(pattern, match); /* Tm = O(?) */
    s = p = 0;
    while (s<n && p<m) { /* O(n) */
        if (string[s]==pattern[p]) { s++; p++; }
        else if (p>0) p = match[p-1]+1;
        else s++;
    }
    return (p == m)? (s-m) : NotFound;
}
```

KMP模式匹配样例

关键代码

```
int Index_KMP(String S, String T, int pos)
{
    int i = pos;
    int j = 1;
    int next[255];
    get_next(T, next);
    while (i <= S[0] && j <= T[0])
    {
        if (j==0 || S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            j = next[j];
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

数据变化

S: a b c d e f g a b ...

T: a b c d e x

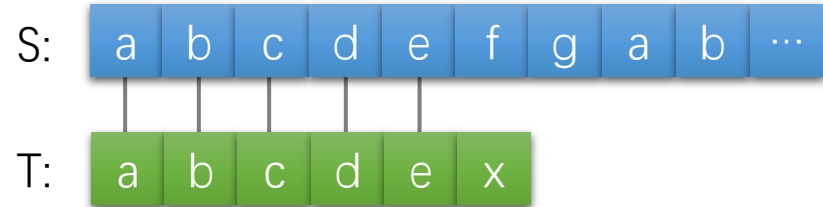
pos=1, i=1, j=1, S[0]=10, T[0]=6, next=[0,1,1,1,1,1,.....]

KMP模式匹配样例

关键代码

```
int Index_KMP(String S, String T, int pos)
{
    int i = pos;
    int j = 1;
    int next[255];
    get_next(T, next);
    while (i <= S[0] && j <= T[0])
    {
        if (j==0 || S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            j = next[j];
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

数据变化



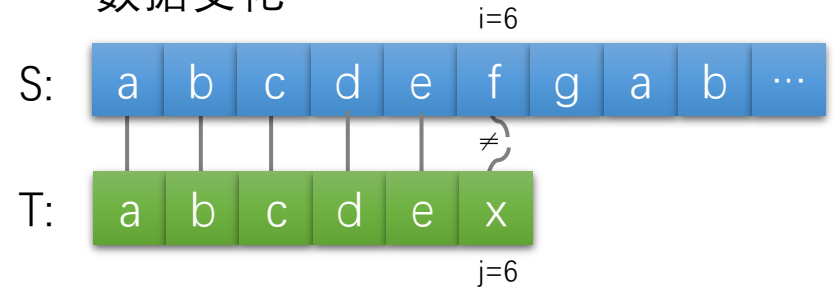
pos=1, i=5, j=5, S[0]=10, T[0]=6, next=[0,1,1,1,1,1,.....]

KMP模式匹配样例

关键代码

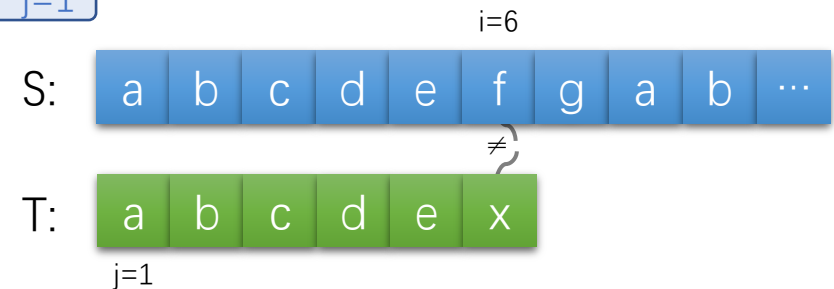
```
int Index_KMP(String S, String T, int pos)
{
    int i = pos;
    int j = 1;
    int next[255];
    get_next(T, next);
    while (i <= S[0] && j <= T[0])
    {
        if (j==0 || S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            j = next[j];
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

数据变化



pos=1, i=6, j=6, S[0]=10, T[0]=6, next=[0,1,1,1,1,1,.....]

next[j]=1, j=1



KMP改进：nextval数组值推导

关键代码

```
void get_nextval(String T, int *nextval)
{
    int i,k;
    i=1;
    k=0;
    nextval[1]=0;
    while (i<T[0])
    {
        if(k==0 || T[i]== T[k])
        {
            ++i;
            ++k;
            if (T[i]!=T[k])
                nextval[i] = k;
            else
                nextval[i] = nextval[k];
        }
        else
            k= nextval[k];
    }
}
```

T=ababaaaba

T[0]=6, i=1, k=0, next[1]=0, nextval[2]=0

T[0]=6, i=2, k=1, next[1]=0, nextval[2]=1

| | | |
|-----|-----|--------------------|
| i=2 | k=1 | nextval =010000000 |
| i=2 | k=0 | nextval =010000000 |
| i=3 | k=1 | nextval =010000000 |
| i=4 | k=2 | nextval =010100000 |
| i=5 | k=3 | nextval =010100000 |
| i=6 | k=4 | nextval =010104000 |
| i=6 | k=1 | nextval =010104000 |
| i=7 | k=2 | nextval =010104200 |
| i=7 | k=1 | nextval =010104200 |
| i=8 | k=2 | nextval =010104210 |
| i=9 | k=3 | nextval =010104210 |

KMP改进：nextval数组值推导

关键代码

```
void get_nextval(String T, int *nextval)
{
    int i,k;
    i=1;
    k=0;
    nextval[1]=0;
    while (i<T[0])
    {
        if(k==0 || T[i]== T[k])
        {
            ++i;
            ++k;
            if (T[i]!=T[k])
                nextval[i] = k;
            else
                nextval[i] = nextval[k];
        }
        else
            k = nextval[k];
    }
}
```

T=aaaaaaab

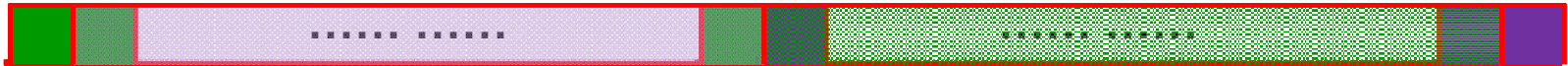
T[0]=6, i=1, k=0, next[1]=0, nextval[2]=0

T[0]=6, i=2, k=1, next[1]=0, nextval[2]=1

| | | |
|-----|-----|---------------------|
| i=2 | k=1 | nextval = 000000000 |
| i=3 | k=2 | nextval = 000000000 |
| i=4 | k=3 | nextval = 000000000 |
| i=5 | k=4 | nextval = 000000000 |
| i=6 | k=5 | nextval = 000000000 |
| i=7 | k=6 | nextval = 000000000 |
| i=8 | k=7 | nextval = 000000000 |
| i=9 | k=8 | nextval = 000000008 |

BuildMatch 的实现

```
for ( j=0; j<m; j++ )
```

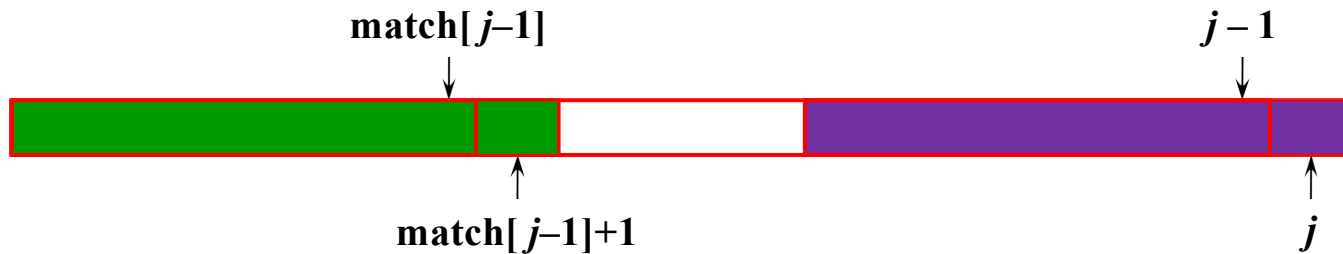


$$1 + 2 + \dots + \frac{j+1}{2} + \dots + j = O(j^2)$$

$$T_m = O(m^3)$$

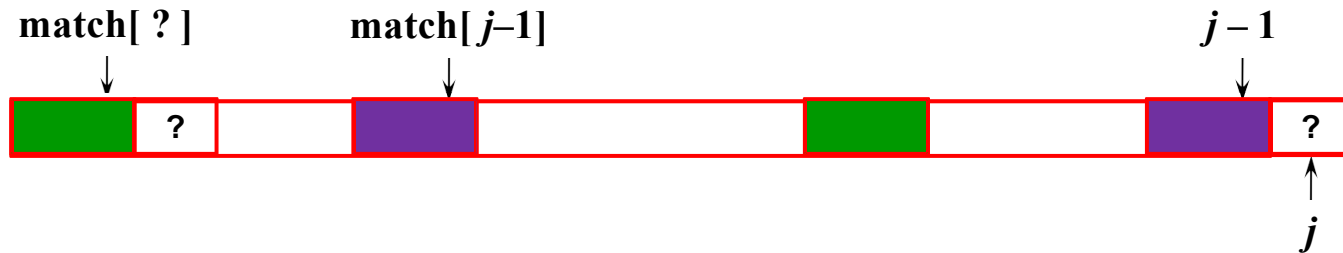


BuildMatch 的实现



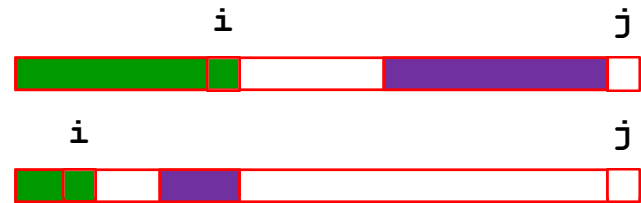
if (pattern[match [j-1]+1] == pattern[j])

match[j] = match[j-1] + 1



BuildMatch 的实现

```
void BuildMatch(char *pattern, int *match)
{
    int i, j;
    int m = strlen(pattern);
    match[0] = -1;
    for (j=1; j<m; j++) {
        i = match[j-1];
        while ((i>=0) && (pattern[i+1]!=pattern[j]))
            i = match[i];
        if (pattern[i+1]==pattern[j])
            match[j] = i+1;
        else match[j] = -1;
    }
}
```



BuildMatch 的实现

```
void BuildMatch(char *pattern, int *match)
{
    int i, j;
    int m = strlen(pattern); /* O(m) */
    match[0] = -1;
    for (j=1; j<m; j++) { /* O(m) */
        i = match[j-1];
        while ((i>=0) && (pattern[i+1]!=pattern[j]))
            i = match[i];
        if (pattern[i+1]==pattern[j])
            match[j] = i+1;
        else match[j] = -1;
    }
}
```

$T_m = O(m^2)$?

i 回退的总次数不会超过 i 增加的总次数

$$T_m = O(m)$$

模式匹配比较



主串S= “000000000200000000020000000002000000000200000000020000000001”

子串T= “0000000001”

从主串S中查找是否有子串存在，返回对应位置

朴素模式匹配算法 循环 285 次 得到查询结果

KMP模式匹配算法 循环 110 次 得到查询结果

KMP模式匹配改进算法 循环 70 次 得到查询结果

KMP算法效率分析

- 循环体中“ $j = N[j];$ ” 语句的执行次数不能超过 n 次。否则，
 - 由于 “ $j = N[j];$ ” 每执行一次必然使得 j 减少(至少减1)
 - 而使得 j 增加的操作只有 “ $j++$ ”
 - 那么，如果 “ $j = N[j];$ ” 的执行次数超过 n 次，最终的结果必然使得 j 为**比-1小很多的**负数。这是不可能的（ j 有时为-1,但是很快+1回到0）。
- 同理可以分析出求 N 数组的时间为 $O(m)$
故，KMP算法的时间为 $O(n+m)$

单模式的匹配算法

| 算法 | 预处理时间效率 | 匹配时间效率 |
|---|------------------------|---------------------------------|
| 朴素匹配算法 | O (无需预处理) | $\Theta(n \cdot m)$ |
| KMP算法 | $\Theta(m)$ | $\Theta(n)$ |
| BM算法 | $\Theta(m)$ | 最优 (n/m) , 最差 $\Theta(nm)$ |
| 位运算算法 (<i>shift-or</i> , <i>shift-and</i>) | $\Theta(m + \Sigma)$ | $\Theta(n)$ |
| Rabin-Karp算法 | $\Theta(m)$ | 平均 $(n+m)$, 最差 $\Theta(nm)$ |
| 有限状态自动机 | $\Theta(m + \Sigma)$ | $\Theta(n)$ |