# 第11章

# 串

**[11-1] 在微软 Office 套件中，Excel 针对字符串操作提供了一系列的函数。**

**a） 查阅手册了解 len(S)、left(S, k)、right(S, k)、mid（S, i, k)和 exact(S, T)的功能；**

**b） 这些功能分别对应于本章所讨论的哪些问题？**

【解答】

len(S)：

计算字符串S的长度，等效于S.length()。

left(S, k)：

在字符串S中取长度为k的前缀，等效于S.prefix(k)。

right(S, k)：

在字符串S中取长度为k的后缀，等效于S.suffix(k)。

mid(S, i, k)：

在字符串S中自第i个字符起取长度为k的子串，等效于S.substr(i, k)。

exact(S, T)：

判断字符串S和T是否相等，等效于S.equal(T)。

**[11-2] 考查教材 309 页代码 11.1 和 310 页代码 11.2 中，match()算法的两个版本。**

**试验证，它们的返回值均为最后一轮比对时串 P 与串 T 的对齐位置，故通过表达式**

```
! ( strlen(T) < match(P, T) + strlen(P) )
```

**即可判断匹配是否成功。**

【解答】

请读者阅读并分析相关代码，并独立给出结论。

**[11-3] 考查由 26 个大写英文字母组成的字母表。**

**试针对以下模式串，构造对应的 next[]表、改进的 next[]表、bc[]表、ss[]表以及 gs[]表：**

"MIAMI"、"BARBARA"、"CINCINNATI"、"PHILADELPHIA"



【解答】

具体解答如以下各表所示。

其中的bc[]定义于字符集Σ上，其长度应等于|Σ|。然而为简洁起见，这里省略了未在P中出现的字符（根据定义其BC值均为-1），而仅仅考查了的确在P中出现的字符，并标记出其在P中的最后一次出现（对应的秩即为该字符对应的BC值）。

| j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| P[j] | M | I | A | M | I |
| next[j] | -1 | 0 | 0 | 0 | 1 |
| 改进的next[j] | -1 | 0 | 0 | -1 | 0 |
| bc[] | | | A | M | I |
| ss[j] | 0 | 2 | 0 | 0 | 5 |
| gs[j] | 3 | 3 | 3 | 5 | 1 |

| j | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| P[j] | B | A | R | B | A | R | A |
| next[j] | -1 | 0 | 0 | 0 | 1 | 2 | 3 |
| 改进的next[j] | -1 | 0 | 0 | -1 | 0 | 0 | 3 |
| bc[] | | | | B | | R | A |
| ss[j] | 0 | 1 | 0 | 0 | 1 | 0 | 7 |
| gs[j] | 7 | 7 | 7 | 7 | 7 | 2 | 1 |

| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| P[j] | C | I | N | C | I | N | N | A | T | I |
| next[j] | -1 | 0 | 0 | 0 | 1 | 2 | 3 | 0 | 0 | 0 |
| 改进的next[j] | -1 | 0 | 0 | -1 | 0 | 0 | 3 | 0 | 0 | 0 |
| bc[] | | | | C | | | N | A | T | I |
| ss[j] | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 10 |
| gs[j] | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 5 | 1 |

| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P[j] | P | H | I | L | A | D | E | L | P | H | I | A |
| next[j] | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 |
| 改进的next[j] | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 3 |
| bc[] | | | | | | D | E | L | P | H | I | A |
| ss[j] | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 12 |
| gs[j] | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 7 | 1 |

**[11-4] 为评估 KMP 算法的效率，11.3.7 节引入一个随迭代过程严格单调递增的观察量 k = 2i - j，从而简捷地证明了迭代的次数不可能超过 $O$(n)。这一初等的证明虽无可辩驳，但毕竟未能直观地展示出其与计算成本之间的本质联系。**

**试证明，在算法执行的整个过程中：**

**❶ 观察量 i 始终等于已经做过的成功比对（含与最左端虚拟通配符的“比对”）次数；**

**❷ 观察量 i - j 始终不小于已经做过的失败比对次数。**

【解答】

反观KMP主算法（教材313页代码11.3），循环中if判断语句的两个分支，分别对应于题中所定义的成功和失败比对。其中，只有成功的分支会修改观察量i——更准确地说，观察量i加一，当且仅当当前的比对是成功的。考虑到观察量i的初始值为0，故在整个算法过程中，它始终忠实地记录着成功比对的次数。

观察量i - j的初始值也是0。对于成功分支，变量i和j会同时递增一个单位，故i - j的数值将保持不变。而在失败分支中，首先观察量i保持不变。另一方面，因为必有：

next[j] < j

故在将变量j替换为next[j]之后，观察量i - j亦必严格单调地增加。综合以上两种情况，观察量i - j必然可以作为失败比对次数的上界。

**[11-5] 针对坏字符在模式串 P 中位置太靠右，以至位移量为负的情况，11.4.2 节建议的处理方法是直接将 P 右移一个字符。然而如图 11.10(f)所示，此后并不能保证原坏字符位置能够恢复匹配。为此，或许你会想到：可在 P[j]的左侧找到最靠右的字符'X'，并将其与原坏字符对齐。**

**a） 试具体实现这种处理方法；**

【解答】

请读者按照以上思路，独立完成编码和调试任务。

**b） 为什么我们不倾向于使用这种方法？**

【解答】

尽管以上思路的实现方式可能不尽相同，但本质上都等效于将原先一维的bc[]表，替换为二维的bc[][]表。具体地，这是一张m × |Σ|的表格，其中bc[j]['X']指向“在P[j]左侧并与之最近的字符'X'”。

如此，尽管预处理时间和所需空间的增长量并不大，但匹配算法的逻辑控制却进一步复杂化。最重要的是，此类二维bc[][]表若能发挥作用，则当时的好后缀必然很长——此类情况，同时使用的gs[]表完全可以替代bc[][]表。



**[11-6] 考查 gs[]表构造算法（教材 326 页代码 11.8），记模式串的长度|P| = m。试证明：**

**a） buildSS()过程的运行时间为 $O$(m)；**

**（提示：尽管其中存在“两重”循环，但内循环的累计执行次数不超过变量 lo 的变化幅度）**

【解答】

该算法的运行时间，主要消耗于其中的“两重”循环。

暂且忽略内（while）循环，首先考查外（for）循环。若将j视作其控制变量，则不难验证：

a. j的初始值为m - 2
b. 每经过一步迭代，j都会递减一个单位
c. 在其它的任何语句中，j都没有作为左值被修改
d. 一旦j减至负数，外循环随即终止

由此可知，外循环至多迭代$O(m)$步，累计耗时不超过$O(m)$。

尽管从表面的形式看，外循环的每一步都有可能执行一趟内循环，但实际上所有内循环的累计运行时间也不超过$O(m)$。为此，只需将lo视作其控制变量，即不难验证：

a. lo的初始值为m - 1
b. 每经过一步内循环的迭代，lo都会递减一个单位
c. 在其它部分，lo只在“lo = __min(lo, hi)”一句中作为左值被修改，但仍是非增
d. 一旦lo减至负数，内循环就不再启动

由此可知，内循环累计至多迭代$O(m)$步，相应地累计耗时不超过$O(m)$。

综合以上两项，即得题中结论。

b） buildGS()过程的运行时间为$O(m)$。

（提示：尽管其中存在“两重”循环，但内循环的累计执行次数不超过变量i的变化幅度）

【解答】

仿照a）中的分析技巧。只要以j作为外循环的控制变量，则可知外循环至多迭代$O(m)$步，耗时$O(m)$；以i作为内循环的控制变量，则可知内循环累计至多迭代$O(m)$步，累计耗时$O(m)$。

请读者根据以上实例及提示，独立补充证明的细节。

[11-7] 在模式枚举（pattern enumeration）类应用中，需要从文本串T中找出所有的模式串P（|T| = n，|P| = m），而且有时允许模式串的两次出现位置之间相距不足m个字符。

类似于教材310页图11.3中的实例，比如在"000000"中查找"000"。若限制多次出现的模式串之间至少相距|P| = 3个字符，则应找到2处匹配；反之，若不作限制，则将找到4处匹配。

a） 试举例说明，若采用后一约定，则教材11.4.3节BM算法的好后缀策略，可能需要$\Omega(nm)$时间；

【解答】

将题中所举实例一般化，取模式串P = "00...0"，|P| = m，则P对应的gs[]表应如下：

| j | 0 | 1 | 2 | 3 | 4 | ... | m - 2 | m - 1 |
|---|---|---|---|---|---|---|---|---|
| gs[j] | 1 | 2 | 3 | 4 | 5 | ... | m - 1 | m |

再取文本串T = "00000...0"，|T| = n >> m。

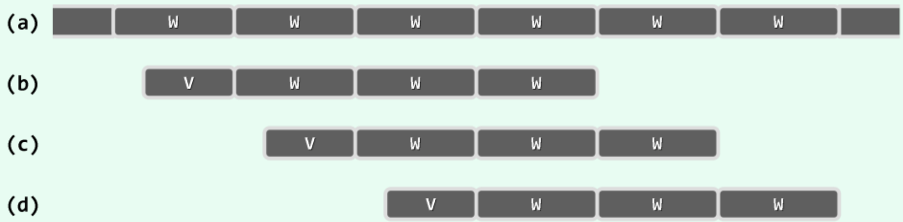
于是自然地，在每一对齐位置，经过m次比对之后都可以找到一次完全匹配。然而接下来，只能右移gs[0] = 1位并重新对齐，经过m次比对之后方可找到下一次完全匹配。

如上过程将反复进行，直到文本串被扫描完毕。整个过程共有n - m + 1个对齐位置，而且在每个位置都需要经过m次比对，方可发现一次完全匹配。鉴于n和m取值的任意性，在此类最坏

情况下，该算法的累计耗时量应为：

$$(n - m + 1) \times m \quad = \quad \Theta(nm)$$

以上实例仍然非常极端，更具一般性的例子则如图x11.1所示。




图x11.1 BM算法的最坏情况

这里以两个基本的字符串W和V作为“积木”。为简化起见，假定字符串W中的彼此字符互异，$|W| = w$为常数；V是W的一个非空子串，$|V| = v \leq w$。文本串T如图(a)所示，由n/w个W顺次串接而成。模式串P如图(b)所示，由一个V和$(m - v)/w$个W顺次串接而成。当然，与通常情况一样，这里也有$2 \ll m \ll n$。

于是，P对应的gs[]表应如下：

$$gs[j] \quad = \quad \begin{cases} \lceil (j+1)/w \rceil \cdot w & (0 \leq j < m-v) \\ m & (m-v \leq j < m-1) \\ 1 & (j = m-1) \end{cases}$$

其中特别地，有：

$$gs[0] \quad = \quad w$$

因此，在每次发现一个完全匹配后，P都会右移w位并与T重新对齐，然后找到下一个完全匹配。如此，总共会有n/w个对齐位置（各对应于一个完全匹配）；而重要的是，每次对齐之后都需要经过m次比对。由此可见，整个过程所做比对的次数累计为：

$$n/w \times m \quad = \quad \Theta(nm)$$

**b）试针对这一缺陷改进好后缀策略，使之即便在采用后一约定时，最坏情况下也只需线性时间；（提示：Galil规则）**

【解答】

反观以上一般性实例可见，其中模式串P每一次右移，都属于如教材321页图11.12(d)所示的情况：在前一轮比对中，成功次数过多，以致好后缀过长（甚至如上例，就是P整体）。

这里的技巧是，在此类对齐位置，不必一直比对至P的最左端。实际上不难看出，一旦自右向左比对到原文本串T中好后缀的最右端，即可马上判定是否完全匹配。仍以图x11.1为例，除了第一轮比对，在后续的各轮比对中，均只需比较模式串P中最靠右的w个字符——根据gs[]表的定义，其余的$m - w$个字符必然是匹配的。

利用这一所谓的Galil规则加以改进之后，文本串T的每个字符都不再会重复接受比对。既然累计不超过线性次比对，总体耗时也就不致超过线性的规模。

**c） 在本章所给相关代码的基础上，实现以上改进。**

请读者参照以上分析和介绍，独立完成编码和调试任务。

**[11-8] 在讲解 gs[]表的构造算法时，为简洁起见，教材图 11.14、图 11.15 和图 11.16 中所绘出 MS[j] 均与其所对应的最长匹配后缀没有任何重叠。然而，这种表示方法并不足以代表一般性的情况。**

**a） 试举一例说明，这两个子串有可能部分重叠；**

【解答】

请读者仿照教材所列的基本情况，独立给出具体实例。

**b） 试证明，即便二者有所重叠，教材 11.4.4 节所做的原理分析，以及 326 页代码 11.8 所给的算法实现，均依然成立。**

【解答】

请读者对照相关的代码及分析，独立给出证明。

**[11-9] 教材 309 页代码 11.1、310 页代码 11.2 所实现的两个蛮力算法，在通常情况下的效率并不算低。现假定所有字符出现的概率均等，试证明：**

**a） 任意字符比对的成功与失败概率分别为 $1/s$ 和 $(s - 1)/s$，其中 $s = |\Sigma|$ 为字符表的规模；**

【解答】

每个字符各有$1/s$的概率出现，故任何一对字符相同、不同的概率分别为为$1/s$和$(s - 1)/s$。

**b） 在 P 与 T 的每一对齐位置，需连续执行恰好 k 次字符比对操作的概率为 $(s - 1)/s^k$；**

【解答】

恰好执行k次字符对比，当且仅当前$k - 1$次成功，但最后一次失败。根据a）的分析结论，这类事件发生的概率应为：

$$(1/s)^{k-1} \cdot (s - 1)/s = (s - 1)/s^k$$

**c） 在 P 与 T 的每一对齐位置，需连续执行字符比对操作的期望次数不超过 $s/(s - 1) \le 2 = \mathcal{O}(1)$。**

【解答】

由b）的分析结论，每一次字符比对都可视作一次伯努利实验（Bernoulli trial），成功与失败的概率分别为$1/s$和$(s - 1)/s$；而每趟比对的次数X，则符合几何分布（geometric distribution）——亦即，其中前$X-1$次实验成功的概率各为$1/s$，最终一次实验失败的概率为$(s - 1)/s$。因此，X的期望值不超过$s/(s - 1)$。

直接由期望值的定义出发，也可得出同样结论。具体地，连续执行字符比对操作的期望次数，应该就是所有可能的次数，关于其对应概率的加权平均，亦即：

$$\sum_{k=1}^{m} k\cdot(s - 1)/s^k = (s - 1)\cdot\sum_{k=1}^{m} k/s^k \le (s - 1)\cdot\sum_{k=1}^{\infty} k/s^k = s/(s - 1)$$

### [11-10] BM 算法与 KMP 算法分别擅长于处理何种类型的字符串？为什么？

【解答】

正如教材第11.4.5节所指出的，在评价不同串匹配算法各自的实用范围时，在不同应用中单次比对的成功概率，扮演着重要的角色。而根据习题[11-9]的分析结论，在通常的情况下，这一概率首先并直接取决于字符集的规模。

当字符集规模较小时，单次比对的成功概率较高，蛮力算法的效率较低。此时，KMP算法稳定的线性复杂度，更能体现出优势；而采用BC表的BM算法，却并不能大跨度地向前移动。

反之，若字符集规模较大，则单次比对的成功概率较小，蛮力算法也能接近于线性的复杂度。此时，KMP算法尽管依然保持线性复杂度，但相对而言的优势并不明显；而采用BC表的BM算法，则会因比对失败的概率增加，可以大跨度地向前移动。