# 3，字符串

- 字符串的相关概念
- Python 字符串（回顾）
- 字符串匹配和算法
- 进一步的模式匹配问题
- 正则表达式
- Python 的正则表达式
- 应用举例



## 字符串

- 讨论字符串，首先要有一个确定的字符集
  - “字符”是一个抽象概念，字符集是有穷的一组字符构成的集合
  - 人们经常考虑在计算机里使用的的标准字符集，实际上，完全可以拿任意数据元素的集合作为字符集
- 字符串（简称串）是特殊的线性表，表中元素取值选定的字符集

  其不同于一般表的特点是支持一组以串为对象的操作
- 长度：串中字符的个数称为串的长度
  - 长度为 0 的串称为空串
  - 在任意一个字符集里，只有唯一的一个空串
- 与一般的表类似：
  - 字符串里的字符顺序排列，串里的每个字符有其确定位置
  - 我们用 0 开始的自然数表示位置

## 字符串

- 串相等：串的相等基于字符集里的字符定义

  **s1** 和 **s2** 相等，如果其长度相等，而且对应位置的各对字符分别相同

- 假定字符集中的字符有一个全序，串的字典序定义如下：

  对于串 $s_1 = a_0 a_1 \cdots a_{n-1}$，$s_2 = b_0 b_1 \cdots b_{m-1}$

  定义 $s_1 < s_2$，如果存在一个 $k$ 使 $a_i = b_i$ $(i = 0,1, \ldots k-1)$ 而且 $a_k < b_k$

  或者 $n < m$ 而且对 $i = 0,1, \ldots n-1$ 都有 $a_i = b_i$

  显然，字典序是字符串集合上的一个全序

- 串与串的最重要运算是拼接（**concatenate**）

  上面 $s_1$ 和 $s_2$ 的拼接是串 $s = a_0 a_1 \cdots a_{n-1} b_0 b_1 \cdots b_{m-1}$

  显然，$s$ 的长度等于 $s_1$ 和 $s_2$ 的长度之和

  在 **Python** 里拼接运算用 **+** 表示



## 字符串

- 两个串之间还有一个重要的关系是“子串关系”
- 称 $s_1$ 为 $s_2$ 的一个子串，如果存在两个串 **s** 和 **s'** 使下式成立

  $s_2 = s + s_1 + s'$ （借用 **Python** 的写法）

  - 子串也是串。直观看，子串是原串中连续的一段字符序列形成的串。显然，一个串可以是或者不是另一个串的子串
  - 如果 $s_1$ 是 $s_2$ 的子串，也说 $s_1$ 在 $s_2$ 里出现，称 $s_2$ 里与 $s_1$ 相同的字符段的第一个字符的位置为 $s_1$ 在 $s_2$ 里出现的位置
  - $s_2$ 里可能出现多个与 $s_1$ 相同的段，这时说 $s_1$ 在 $s_2$ 里多次出现

- 注意：**s1** 在 **s2** 中的多个出现可能不独立，相互重叠。例如

  **babb** 在 **babbabbbbabb** 里有三个出现，前两个有重叠

- 根据定义，很显然，空串是任何字符串的子串；另一方面，任何字符串 **s** 也都是该串本身的子串

## 字符串

- 两种特殊子串：
  - 如果存在 s' 使 $s_2 = s_1 + s'$，称 $s_1$ 为 $s_2$ 的一个前缀
  - 如果存在 s 使得 $s_2 = s + s_1$，称 $s_1$ 为 $s_2$ 的一个后缀

  直观说，一个串的前缀就是该串开头的一段字符构成的子串，后缀就是该串最后的一段字符构成的子串

  显然，空串和 s 既是 s 的前缀，也是 s 的后缀
- 其他有用的串运算：
  - 串 s 的 n 次幂 $s^n$ 是连续 n 个 s 拼接而成的串（在 Python 语言里用 s * n 表示）
  - 串替换，指将一个串里的一些（互不重叠的）子串代换为另一些串得到的结果（由于可能重叠，需规定代换的顺序，如从左到右）
  - 还有许多有用的串运算，可以参考 Python 的 str 类型，或其他语言的字符串类型（经典是 SNOBOL 语言）



## 字符串的理论

- 字符串集合和拼接操作构成了一种代数结构
  - 空串是拼接操作的“单位元”（幺元）

    有结合律，无交换律
  - 串集合加上拼接操作，构成一个半群

    一种典型的非交换半群
  - 有单位元，因此是一个幺半群
- 关于串的理论有许多研究工作
  - 基于串和串替换，研究者提出了 post 系统

    这是一种与图灵机等价的计算模型
  - （串）重写系统（rewriting system）是计算机理论的一个研究领域，一直非常活跃，有许多重要结果和应用

## 字符串的实现

- 串是字符的线性序列：
  - 可采用线性表的实现方式，用顺序表示和链接表示。例如用能动态变化大小的顺序表作为实现方式（如果需要表示可变的字符串）
  - 还可以根据串本身的特点和串操作的特点，考虑其他表示方式。当然，无论如何还是基于顺序存储或/和链接
  - 关键问题：表示方式应能很好支持串的管理和相关操作的实现
- 字符串表示的两个问题：
  - 串内容存储。两个极端：1，连续存储在一块存储区；2，一个字符存入一个独立存储块，链接起来。也可以采用某种中间方式，把串中字符分段保存在一组存储块里，链接起这些存储块
  - 串结束的表示，不同字符串长度可能不同，必须表示串到哪里结束。两种基本方式：1，用专门数据域记录字符串长度；2，用一个特殊符号表示串结束（例如 C 语言的字符串采用这种方式）



## 字符串的实现

- 现在考虑字符串的操作

  许多串操作是线性表操作的具体实例，包括串拼接

  下面考虑一个特殊的操作
- 串替换
  - 牵涉到三个串：被处理的主串 s，作为被替换对象需要从 s 中替换掉的子串 t，以及用于代换 t 的 t'
  - 由于 t 可能在 s 中出现多次，因此需要通过一系列具体的子串代换完成整个替换
  - 由于多次出现可能重叠（回忆前面的例子），只能规定一种代换顺序（例如从左到右），一次代换破坏的子串不应再代入新串
  - 一次子串代换后，应该从代入的新串之后继续工作。即使代入新串之后形成的部分能与 t 匹配，也不应在本次替换中考虑
  - 易见：串替换的关键是找到串匹配，这是后面讨论的主题

## 实际语言里的字符串

- 许多语言提供了标准的字符串功能，如
  - C语言标准库有一组字符串函数（string.h），一些C语言系统提供的扩展的字符串库；C++ 语言标准库里的字符串库 <string>
  - Java 标准库的字符串库
  - 许多脚本语言（包括 Python）提供了功能丰富的字符串库
- 许多实际字符串库用动态顺序表结构作为字符串的表示方式
  - 这样既能支持任意长的字符串
  - 又能比较有效地实现各种重要的字符串操作
- 实际上，支持不同的字符串操作，可能需要不同的实现，例如
  - 有些计算工作需要记录和处理极长的字符串，如支持操作 MB（大约为 $10^6$）或更长的字符串，采用连续存储可能带来管理问题
  - 被编辑文本也是字符串，实现编辑器操作要考虑专门的字符串表示



## Python 字符串

- Python 内部类型 str 是抽象字符串概念的一个实现
  - str 是不变类型，str 对象创建后的内容（和长度）不变
  - 但不同的 str 对象长度不同，需要记录
- Python 采用一体式的连续形式表示 str 对象，见下图


- str 对象的操作分为两类：
  - 获取 str 对象的信息，如得到串长，检查串内容是否全为数字等
  - 基于 str 对象构造新的 str 对象，包括切片，构造小写/大写拷贝，各种格式化等。切分是构造包含多个字符串的表
- 一些操作属子串匹配，如 count 检查子串出现次数，endwith 检查后缀，find/index 找子串位置等。这类操作最重要，后面专门讨论

## 字符串操作的实现

- 检查字符串内容的操作可以分为两类
  - O(1) 时间的简单操作，包括 len 和定位访问（也构造新字符串）
  - 其他都需要扫描整个串的内容，包括不变序列的共有操作（in、not in、min/max），各种字符类型判断（如是否全为数字）
    - 需通过一个循环逐个检查串中字符完成工作，O(n) 操作
    - 子串查找和匹配的问题后面讨论
- 需要构造新字符串的操作情况比较复杂，基本模式都包括两部分工作

  1, 为新构造的串安排一块存储

  2, 根据被操作串（和可能的参数串）构造出一个新串
- 以 s[a, b, k] 为例，算法：

  1, 根据 a、b、k 算出新字符串的长度

  2, for i in range(a, b, k) : 拷贝 s[i] 到新串里的下一个空位



## 字符串匹配（子串查找）

- 最重要的字符串操作是子串匹配，称为字符串匹配（string matching）或字符串查找（string searching）{ 有教科书称为模式匹配（pattern match），但实际上模式匹配是内涵更广的概念。}

  wiki：http://en.wikipedia.org/wiki/String_searching_algorithm
- 字符串匹配问题：

  假设有两个串（$t_i$, $p_j$ 是字符）

  $t = t_0t_1t_2 \ldots t_{n-1}$ 称为目标串

  $p = p_0p_1p_2 \ldots p_{m-1}$ 称为模式串

  通常有 m << n。字符串匹配就是在 t 中查找与等于 p 的子串的过程

  （这一定义可以推广，后面讨论）
- 如前所述，串匹配是最重要的字符串操作，也是其他许多重要字符串操作的基础。实际中 n 可能非常大，m 也可以有一定规模，也可能需要做许多模式串和/或许多目标串的匹配，有关算法的效率非常重要

## 串匹配

- 许多计算机应用的最基本操作是字符串匹配。如
  - 用编辑器或字处理系统工作时，在文本中查找单词或句子（中文字或词语），在程序里找拼写错误的标识符等
  - email 程序的垃圾邮件过滤器，google 等网络检索系统
  - 各种防病毒软件，主要靠在文件里检索病毒模式串
- 在分子生物学领域：DNA 细胞核里的一类长分子，在遗传中起着核心作用。DNA 内有四种碱基：腺嘌呤(adenine)，胞嘧啶(cytosine)，鸟嘌呤(guanine)，胸腺嘧啶(thymine)。它们的不同组合形成氨基酸、蛋白质和其他更高级的生命结构
  - DNA 片段可看作是a,c,g,t构成的模式，如 acgatactagacagt
  - 考查在蛋白质中是否出现某个 DNA 片段，可看成与该 DNA 片段的串匹配问题。DNA 分子可以切断和拼接，切断动作由各种酶完成，酶也是采用特定的模式确定剪切位置



## 串匹配

- 实际中模式匹配的规模（n 和 m）可能非常大，而且有时间要求
  - 被检索的文本可能很大
  - 网络搜索需要处理亿万的网页
  - 防病毒软件要在合理时间内检查数以十万计的文件（以 GB 计）
  - 运行在服务器上的邮件过滤程序，可能需要在一秒钟的时间内扫描数以万计的邮件和附件
  - 为疾病/药物研究/新作物培养等生物学工程应用，需要用大量 DNA 模式与大量 DNA 样本（都是 DNA 序列）匹配
- 由于在计算机科学、生物信息学等许多领域的重要应用，串模式匹配问题已成为一个极端重要的计算问题。高效的串匹配算法非常重要
  - 有几个集中关注字符串匹配问题的国际学术会议，曾经有过专门的国际竞赛（见 wiki 页和万维网）
  - 目前全世界一大半的计算能力是在做串模式匹配（做 DNA 分析）

## 串匹配和算法

- 还需注意不同的实际需要，如
  - 用一个模式在很长的目标串里反复匹配（确定出现位置）
  - 一组（可能很多）模式，在一个或一组目标串里确定是否有匹配
- 不同算法在处理不同实际情况时可能有不同的表现

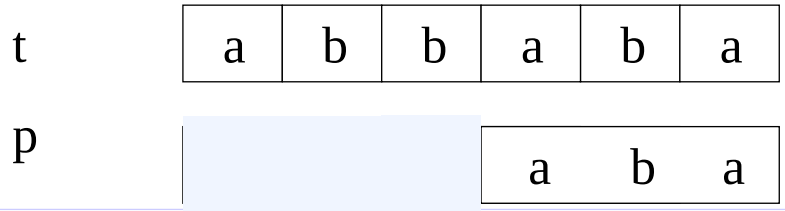
  人们已经开发出一批有意义的（有趣）算法（进一步情况见 **wiki**）
- 粗看，字符串匹配是一个很简单的问题
  - 字符串是简单数据（字符）的简单序列，结构也最简单（顺序）
  - 很容易想到最简单而直接的算法
  - 但事实是：直接而简单的算法并不是高效的算法

    因为它可能没有很好利用问题的内在结构
  - 字符串匹配“貌似简单”，但人们已开发出许多“大相径庭的”算法



## 串匹配的朴素算法

- 串匹配的基础是逐个比较字符
  - 如果从目标串的某个位置开始，模式串的每个字符都与目标串里的对应字符相同，这就是一个匹配
  - 如果出现一对不同字符，就是不匹配
- 算法设计的关键：**1**，怎样比较字符；**2**，发现不匹配后下一步怎么做
  - 对上述两点采用不同的策略，就形成了不同的算法
  - 从下面两个例子可以看到一些情况，更多实例见 **wiki**
- 朴素匹配算法：**1**，从左到右匹配；**2**，发现不匹配时，考虑目标串里的下一位置是否存在匹配



## 串匹配的朴素算法

- 朴素的串匹配算法的一个实现：

```
def nmatching(t, p):
    j, i = 0, 0
    n, m = len(t), len(p)
    while j < n and i < m:  # i==m means a matching
        if t[j] == p[i]: # ok! consider next char in p
            j, i = j+1, i+1
        else:            # no! consider next position in t
            j, i = j-i+1, 0
    if i == m: # find a matching, return its index
        return j-i
    return –1 # no matching, return special value
```

- 朴素匹配算法简单，易理解，但效率低

  造成效率的主要操作是执行中可能出现的回溯：遇字符不等时将模式串 p 右移一个字符，再次从 $p_0$（重置 j = 0 后）开始比较



## 串匹配的朴素算法

- 最坏情况是每趟比较都在最后出现不等，最多比较 n－m＋1 趟，总比较次数为 m*(n－m＋1)，所以算法时间复杂性为 O(m*n)
- 最坏情况的一个实例

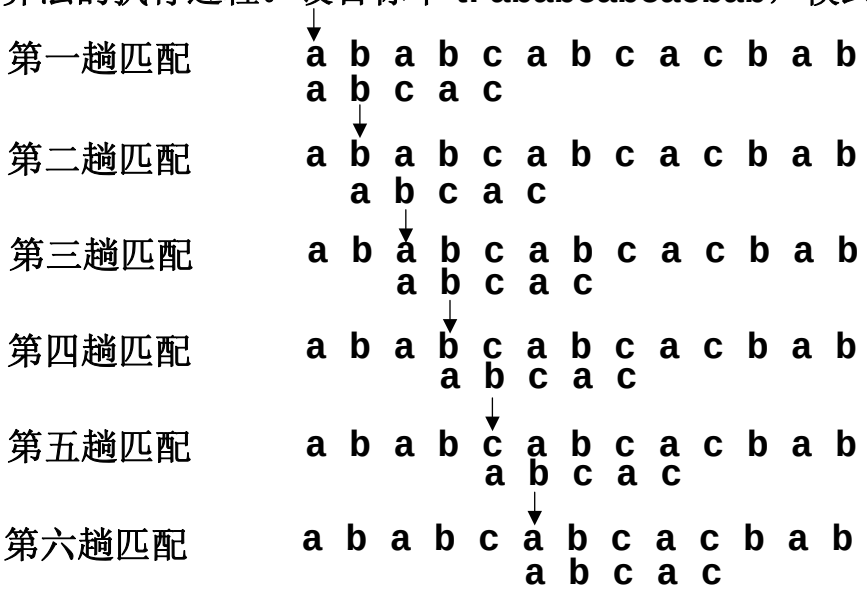
  目标串：00000000000000000000000000000000000000001

  模式串：00000001

- 朴素算法效率低的原因：把每次字符比较看作完全独立的操作
  - 完全没有利用字符串本身的特点（每个字符串都是特殊的）
  - 没有利用前面已做过的字符比较得到的信息
- 从数学上看，这样做相当于认为目标串和模式串里的字符都是随机量，而且有无穷多可能取值，两次字符比较相互无关也不可借鉴
  - 实际字符串取值来自一个有穷集合
  - 每个串都有穷。特别是模式串通常不太长，而且可能反复使用

## 无回溯匹配：KMP算法

- KMP 算法是一个高效串匹配算法。由 D. E. Knuth 和 V. R. Pratt 提出，J. H. Morris 几乎同时发现，因此称 KMP 算法。这是本课第一个非平凡算法，基于对问题的深入分析和理解。算法不易理解但效率高
- 要理解 KMP 算法，首先需要了解朴素算法的缺陷。现在仔细考察朴素算法的执行过程。设目标串 t: ababcabcacbab，模式串 p: abcac

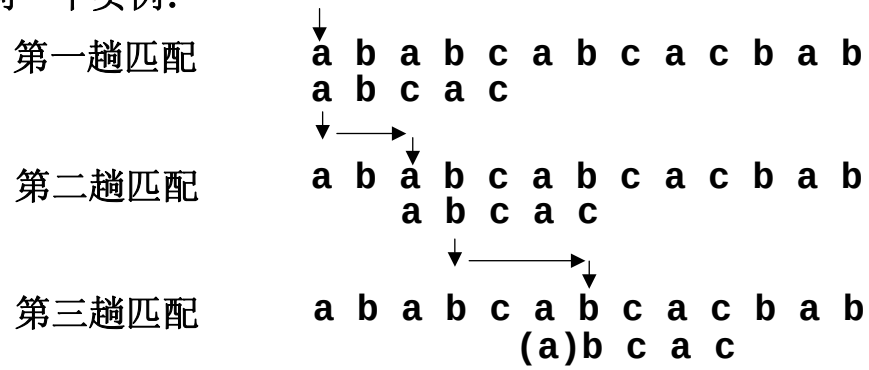



模式串为匹配前已知，在匹配中反复使用

若先对模式串做细致分析，记录有用信息（静态预处理），有可能加快匹配

## 无回溯匹配：KMP算法

- KMP 算法的基本想法：在匹配失败时，利用已做匹配中得到的信息，把模式串尽可能前移。匹配中只做不得不做的字符比较，不回溯
- 处理同一个实例：




- 这里的匹配绝不回溯，如果匹配到 $t_j$ 失败（设遇到 $p_i \neq t_j$），就去找到某个 $p_{ki}$，$0 \le k_i < i$，下一步用 $p_{ki}$ 去与 $t_j$ 比较
- 要正确前移，对模式 p 的每个 $p_i$，都应能找到相应的 $p_{ki}$。问题是，无论对怎样的 $t_j$ 失败，对相应的 i，对应 $k_i$ 都一样吗？

## 无回溯匹配：分析

- 关键认识：在 $p_i$ 匹配失败时，所有 $p_k$（$0 \le k < i$）都已匹配成功（否则不会考虑 $p_i$ 的匹配）。也就是说：

  $t_j$ 之前的 i-1 个字符就是 p 的前 i-1 个字符

  ☺：原本应该根据 t 的情况确定前移方式，但实际上可以根据 p 本身的情况确定，可以通过对模式串本身的分析在匹配之前做好

- 结论：对 p 中的每个 i，有一个唯一确定的 $k_i$，与被匹配的串无关。通过对模式串 p 的预分析，可以得到每个 i 对应的 $k_i$ 值（☺）

- 设 p 的长度为 m，需要对每个 i ($0 \le i < m$) 算出一个 $k_i$ 值并保存，以便在匹配中使用。考虑把这 m 个值（i 和 $k_i$ 的对应关系）存入一个表 pnext，用 pnext[i] 表示与 i 对应的 $k_i$ 值

- 特殊情况：在 $p_i$ 匹配失败时，有可能发现用 $p_i$ 之前的所有 p 字符与 t 字符的比较都没有利用价值，下一步应该从头开始，用 $p_0$ 与 $t_{j+1}$ 比较。遇到这种特殊情况就在 pnext[i] 里记录 –1

  显然，对任何模式都有：pnext [0] = -1



## KMP算法

- 假设已经根据模式串 p 做出了 pnext 表，考虑 KMP 算法的实现

- 匹配循环很容易写出，如下：

```
while j < n and i < m:  # i==m means a matching
    if i == -1 :        # 遇到 –1，比较下一对字符
        j, i = j+1, i+1
    elif t[j] == p[i]:  # 字符相等，比较下一对字符
        j, i = j+1, i+1
    else:               # 从 pnext 取得 p 下一字符的位置
        i = pnext[i]
```

- if 的前两个分支可以合并：

```
while j < n and i < m:  # i==m means a matching
    if i == -1 or t[j] == p[i]:  # 比较下一对字符
        j, i = j+1, i+1
    else:               # 从 pnext 取得 p 下一字符的位置
        i = pnext[i]
```

## KMP算法

- 匹配函数的定义：

```
def matchingKMP(t, p, pnext):
    j, i = 0, 0
    n, m = len(t), len(p)
    while j < n and i < m:  # i==m means a matching
        if i == -1 or t[j] == p[i]: # consider next char in p
            j, i = j+1, i+1
        else:           # no! consider pnext char in p
            i = pnext[i]
    if i == m: # find a matching, return its index
        return j-i
    return -1  # no matching, return special value
```

- 算法复杂性的关键是循环。注意循环中 j 的值递增，但加一的总次数不多于 n = len(t)。而且 j 递增时 i 值也递增。另一方面 i = pnext[i] 总使 i 值减小，但条件保证其值不小于 –1，因此 i = pnext[i] 的执行次数不会多于 i 值递增的次数。循环次数是 O(n)，算法复杂性也是 O(n)



## KMP算法：生成 pnext 表

- 现在考虑 pnext 表的构造，以下面情况为例

第二趟匹配

```
a b a b c a b c a c b a b
    a b c a c
    ⇢(a)b c a c
```

  - 新位置的前缀子串应该与匹配失败字符之前同长度的子串相同
  - 如果在模式串匹配失败时，前面一段里满足上述条件的位置不止一处，只能移到最近的那个位置（保证不遗漏可能的匹配）

- 已知 $k_i$ 值只依赖于 p 本身的前i个字符

$$t_0 \cdots t_{j-i-1} t_{j-i} \cdots t_{j-1}\ t_j \cdots$$
$$\quad\quad\quad\quad\quad\quad\quad\quad \| \quad \| \quad \not=$$
$$\quad\quad\quad\quad\quad\quad\quad\quad p_0 \cdots p_{i-1} p_i \cdots$$

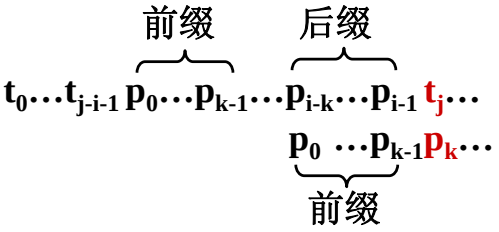
$$t_0 \cdots t_{j-i-1} p_0 \cdots p_{i-1}\ t_j \cdots$$

设匹配到 $p_i/t_j$ 时失败

t 中位置 j 之前的 i 个字符就是 p 的前 i 个字符

## KMP算法：生成 pnext 表

- 正确 k 值的情况看下面图示




模式串的正确前移位置移位，必须保证其前缀 $p_0 \cdots p_{k-1}$ 与 t 中对应那些字符匹配，而这实际上也就是与 $p_{i-k} \cdots p_{i-1}$ 匹配

- 正确 k 值由 p 前 i 个字符形成的子串里相等的前缀和后缀决定

  取这种前后缀中最长的（前移最短），就能保证不忽略可能的匹配

- 如果 $p_0 \cdots p_{i-1}$ 最长相等前后缀（不包括 $p_0 \cdots p_{i-1}$ 本身但可为空）的长度为 k ($0 \le k < i-1$)。当 $p_i \neq t_j$ 时 p 应右移 i - k 位，随后比较 $p_k$ 与 $t_j$

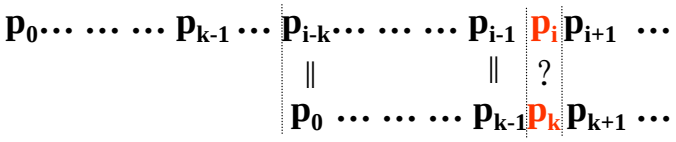
  也就是说，应该把 pnext[i] 设置为 k

- 求 pnext 的问题变成对每个 i 求 p 的（前缀）子串 $p_0 \cdots p_{i-1}$ 的最长相等前后缀的长度。KMP 提出了一种巧妙的递推算法



## KMP算法：生成 pnext 表

- 针对 i 递推计算最长相等前后缀的长度。设对 i-1 已经算出，于是




  - 如果 $p_i = p_k$，pnext[i] 应该为 k，继续
  - 否则把 $p_0 \cdots p_{k-1}$ 的最长相同前缀移过来继续检查

- 利用已知 pnext[0]= -1 直至 pnext[i] 求 pnext[i+1] 的算法：

1. 假设 next [i] = k。若 $p_k = p_i$，则 $p_0 \cdots p_{i-k} \cdots p_i$ 的最大相同前后缀的长度就是 k+1，记入 pnext[i+1]，将 i 值加一后继续递推（循环）
2. 若 $p_k \neq p_i$ 设 k 为 pnext[k] 的值（设 k 为 pnext[k]，也就是去考虑前一个更短的保证匹配的前缀，从那里继续检查）
3. 若 k 值为 -1（一定来自 pnext），得到 $p_0 \cdots p_{i-k} \cdots p_i$ 中最大相同前后缀的长度为 0，设 pnext [i+1] = 0，将 i 值加一后继续递推

## KMP算法：生成 pnext 表

- 生成 pnext 表的 Python 函数定义

```
def genPNext0(p):
    i, k, m = 0, -1, len(p)
    pnext = [-1] * m
    while i < m-1: # generate pnext[i+1]
        while k >= 0 and p[i] != p[k]:
            k = pnext[k]
        i, k = i+1, k+1
        pnext[i] = k # set a pnext entry
    return pnext
```

- 求模式串 p 在目标串 t 里的匹配，可以写：

```
i = KMPmatching(t, p, genPNext0(p))
```

- 上述 pnext 生成算法还可以改进，下面讨论



## 生成 pnext 表：改进

- 设置 pnext[i] 时有一点情况可以考虑：

$$\begin{array}{c} t_0 \cdots t_{j-i-1} p_0 \cdots p_{j-1} t_j \cdots \\ \| \quad \| \quad \not= \\ p_0 \cdots p_{i-1} p_i \cdots \end{array}$$

在 $p_i \neq t_j$ 时（假设 pnext[i] = k），如果发现 $p_i = p_k$，那么一定 $p_k \neq t_j$。所以模式串应右移到 pnext[k]，下一步用 $p_{next[k]}$ 与 $t_j$ 比较

- 改造的算法只有循环体最后的语句不同：

```
def genPNext(p):
    i, k, m = 0, -1, len(p)
    pnext = [-1] * m
    while i < m-1: # generate pnext[i+1]
        while k >= 0 and p[i] != p[k]:
            k = pnext[k]
        i, k = i+1, k+1
        if p[i] == p[k] : # 小修改，可能避免一些不必要的比较
            pnext[i] = pnext[k]
        else:
            pnext[i] = k
    return pnext
```

## 生成 pnext 表：复杂性

- 算法复杂性的主要因素是循环

```
while i < m-1: # generate pnext[i+1]
  while k >= 0 and p[i] != p[k]:
    k = pnext[k]
  i, k = i+1, k+1
```

- 与 KMP 主算法的分析类似（两个算法的循环形式可以相互改变）：
  - i 值递增，但不超过 p 的长度 m，说明大循环体执行 m 次
  - i 加一时 k 也加一，说明 k 值加一 m 次
  - 内层循环执行总导致 k 值减小，但不会小于 –1

  上面情况说明循环体的执行次数为 O(m)，算法复杂性也是 O(m)
- KMP 算法包括 pnext 表构造和实际匹配，O(m+n)。通常情况 m << n，因此可认为算法复杂性为 O(n)。显然优于 O(m*n)



## KMP 算法

- KMP 算法的一个重要优点是执行中不回溯。在处理从外部（外存/网络等）获取的文本时这种特性特别有价值，因为可以一边读一边匹配，不回头重读就不需要保存被匹配串
- KMP 算法的优势
  - KMP 算法特别适合需要多次使用一个模式串的情况和存在许多匹配的情况（如在大文件里反复找一个单词）
  - 相应 pnext 表只需建立一次。这种情况下可以考虑定义一个模式类型，将 pnext 表作为模式的一个成分
- 人们还提出了其他的模式匹配算法（参看 wiki）
  - 另一经典算法由 Boyer 和 Moore 提出，采用自右向左的匹配方式。如果字符集较大且匹配很罕见（许多实际情况如此，如在文章里找单词，在邮件里找垃圾过滤关键字），其速度可能远高于KMP算法
  - 有兴趣的同学可以自己找相关材料读一读

## 字符串的抽象模型

字符集： $\mathbb{C}$ $c \in \mathbb{C}$

字符串

| | | | |
|---|---|---|---|
| $\mathbb{C}^0$ | $\stackrel{\text{def}}{=}$ | $\{\varepsilon\}$ | 长度为 0 的串只有一个 |
| $\mathbb{C}^1$ | $\stackrel{\text{def}}{=}$ | $\mathbb{C}$ | 长度为 1 的串的集合 |
| $\mathbb{C}^{n+1}$ | $\stackrel{\text{def}}{=}$ | $\mathbb{C}^n \times \mathbb{C}$ | 长度为 n+1 的串集合 |
| $\mathbb{S}$ | $\stackrel{\text{def}}{=}$ | $\bigcup_{n\in\mathbb{N}} \mathbb{C}^n$ | 所有有限长字符串的集合 |

运算

串拼接： $\widehat{\ } : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S}$ $s \widehat{\ } t$ 的前一段是 $s$，后一段是 $t$

串长度： $\# : \mathbb{S} \longrightarrow \mathbb{N}$ $\# s = n$ iff $s \in \mathbb{C}^n$

$s, t \in \mathbb{S}$



## 字符串的抽象模型

- 代数法则： $\varepsilon \widehat{\ } s = s = s \widehat{\ } \varepsilon$

  $(s_1 \widehat{\ } s_2) \widehat{\ } s_3 = s_1 \widehat{\ } (s_2 \widehat{\ } s_3)$

  $\# \varepsilon = 0$

  $\#(s_1 \widehat{\ } s_2) = \# s_1 + \# s_2$

- 空串是拼接操作的“单位元”（幺元），有结合律，无交换律
  - 串集合加上拼接操作，构成一个半群
  - 有单位元，因此是一个么半群
- 关于串的理论有许多研究工作
  - 基于串和串替换，研究者开发了 post 系统（一种计算模型）
  - 串重写系统（rewriting system）是计算机理论的一个研究领域