哈爾濱工業大學 海量数据计算研究中心

Massive Data Computing Lab @ HIT

算法设计与分析

最大流的延伸知识

丁小欧
dingxiaoou@hit.edu.cn

1.最大流与线性规划

- 试将网络最大流问题表示为一个线性规划问题。
- 对于给定的网络*G*=*(V,E)*，源为*s*，汇为*t*。设边(i,j)的容量为*uij*，边*(i,j)*的流量为*xij*，可将网络最大流问题表示为如下线性规划问题：

$$\max f$$

$$\text{s. t.}\quad \sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(i,j)\in E} x_{ji} = \begin{cases} f & i = s \\ -f & i = t \\ 0 & i \neq s,t \end{cases}$$

$$0 \leqslant x_{ij} \leqslant u_{ij}$$

1.最大流的变换和应用

- 带下界约束的最大流问题
  - 更一般的情况下，边容量有上界约束和下界约束，即***(u,v)***还存在一个边流量的下界约束$caplow(u,v)$,则可行流$flow$的容量约束变为：
    $caplow(u,v) \le flow(u,v) \le cap(u,v)$
  - 如何求解？
    - 两阶段求解：一、先找满足约束条件的可行流；二、把找到的可行流扩展为最大流

1.最大流的变换和应用

- 带下界约束的最小流问题
  - 找网络中满足流量上下界约束的最小可行流
  - 如何求解?
    - 两阶段求解：一、先找满足约束条件的可行流；二、以t为源，s为汇，用增广路算法反向求解找到最小可行流

1.最小费用流与线性规划

- 试将网络最小费用流问题表示为一个线性规划问题。
- 对于给定的网络*G=(V,E)*，源为*s*，汇为*t*。设边*(i,j)*的容量为*uij*，边*(i,j)*的流量为*xij*，边*(i,j)*的单位流量费用为*cij*，顶点*i*的流供需量为*di*，可将网络最小费用流问题表示为如下线性规划问题：

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij}$$

$$\text{s. t.} \quad \sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(i,j)\in E} x_{ji} = d_i$$

$$0 \leqslant x_{ij} \leqslant u_{ij}$$

1.最小费用流的变换和应用

- 带下界约束的最小费用流问题
    - 如何求解？
        - 先找满足约束条件的可行流；
        - 把找到的可行流扩展为最小费用流

1.最小费用流的变换和应用

- 带下界约束的最小费用最小流问题
    - 如何求解？
        - 先找满足约束条件的可行流；
        - 以t为源，s为汇，用最小费用流算法反向求解找到最小费用最小流

1.最小费用流的变换和应用

- 最小权二分图匹配问题
  - 给定一个带权二分图*G*，找出*G*的一个最小权二分匹配
  - 增设源*s*，汇*t*，*s*到*V1*的每个顶点有边，容量为1，费用为0；*V2*中每个顶掉到*t*有边，容量为1，费用为0. *G*中每条边相当于*G’*中的一条边，容量为1，费用为该边的权值。
  - *G’*的最小费用流相当于*G*的一个最小权二分匹配

2.最大流问题的其他延伸

- 将单源最短路径问题表示为一个线性规划问题
- 将单源最短路径问题表示为一个最小费用流问题
- $G=(V,E)$是一个以s为源，t为汇，且容量均为整数的网络。已知*flow*是G的一个最大流
  - 假设一条已有边(u,v)的容量增加1，设计一个在$O(V+E)$时间内更新最大流*flow*的算法
  - 假设一条已有边(u,v)的容量减少1，设计一个在$O(V+E)$时间内更新最大流*flow*的算法

算法设计与分析

第九章 字符串算法

丁小欧
dingxiaoou@hit.edu.cn

本章内容

9.1概念与定义

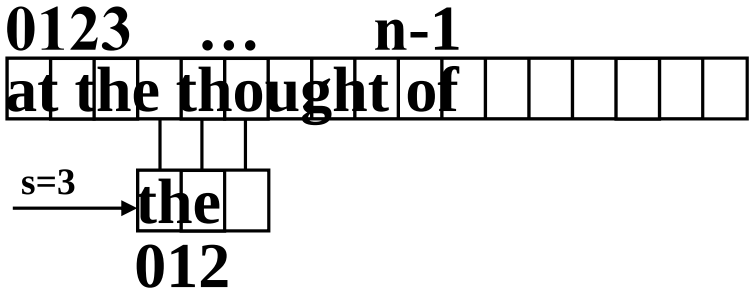
9.2Rabin-Karp算法

9.3 BM算法

9.4 KMP算法

字符串匹配问题

- 输入:
  - *文本* T = “at the thought of”
    - n = length(T) = 17
  - *模式* P = “the”
    - m = length(P) = 3
- 输出:
  - *移动到* s – 最小的整数 ($0 \le s \le n - m$) 满足 $T[s .. s+m-1] = P[0 .. m-1]$. 返回 –1, 如果不存在这样的 s



字符串匹配问题

- 字符串A称为**主串**，把字符串B称为**模式串**：

字符串A：abcdefgh

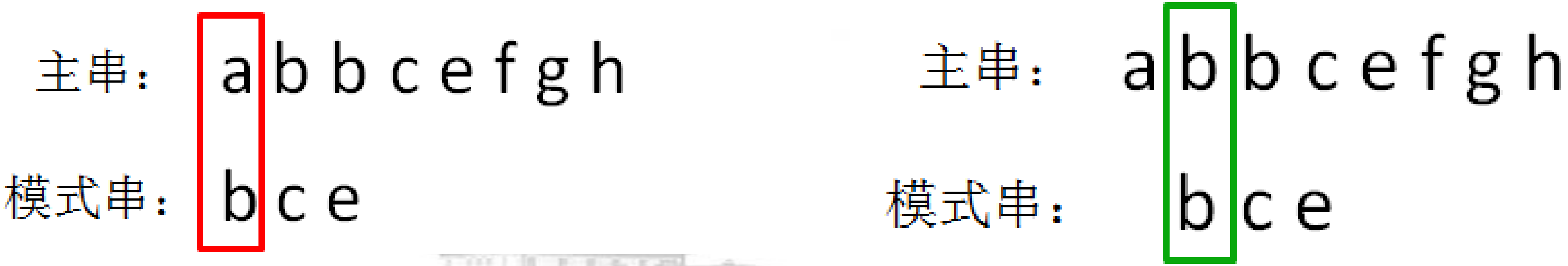
字符串B：cdef

字符串A：abcdefgh

字符串B：bcdg

字符串匹配问题

- 从主串的首位开始，把主串和模式串的字符逐个比较：
  - 主串的首位字符是a，模式串的首位字符是b，两者并不匹配
- 把模式串后移一位，从主串的第二位开始，把主串和模式串的字符逐个比较：
  - 主串的第二位字符是b，模式串的第二位字符也是b，两者匹配，继续比较



主串： a b b c e f g h

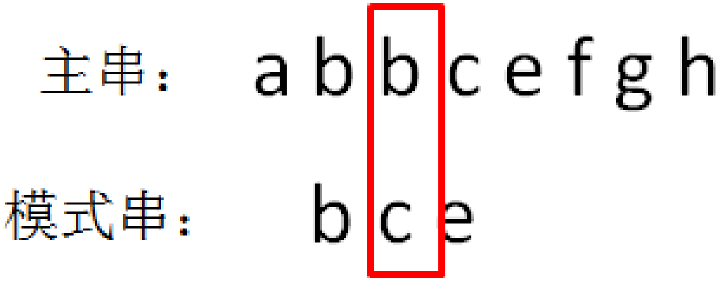
模式串： b c e

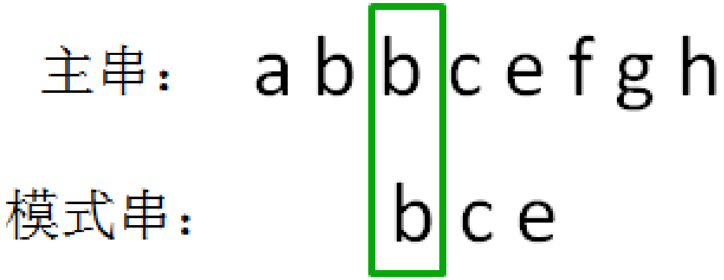
主串： a b b c e f g h

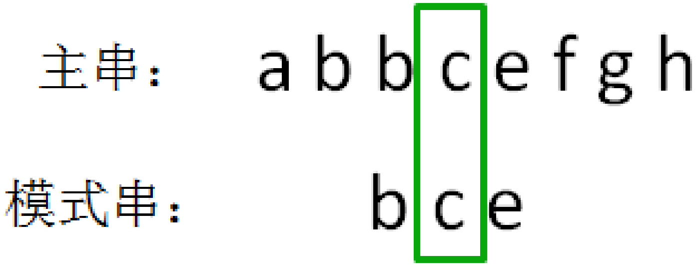
模式串： b c e

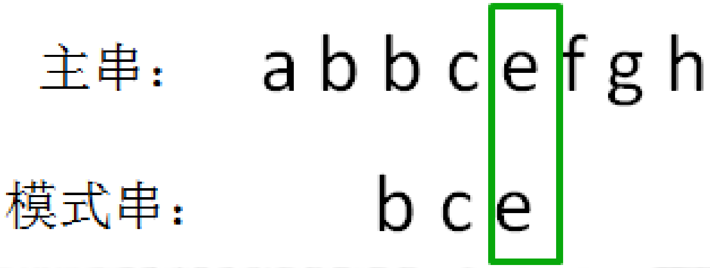
字符串匹配问题

- 两者匹配，比较完成！由此得到结果，模式串 **bce** 是主串 **abbcefgh** 的子串









简单匹配算法

- 暴力搜索 BF算法(Brute Force)
  - 检查从0 到 n – m的所有值

```
Naive-Search(T,P)
for s ← 0 to n – m
    j ← 0
    // check if T[s..s+m–1] = P[0..m–1]
    while T[s+j] = P[j] do
        j ← j + 1
        if j = m return s
return –1
```

- 令 T = “at the thought of” ,
- P = “though”
  - 需要多少次比较?

简单匹配算法的分析

- 最坏情况:
  - 外层循环: $n - m$
  - 内层循环: m
  - 总计 $(n-m)m = O(nm)$
  - 何种输入产生最坏情况?
- 最好情况: $O(n-m)$
  - 何时?
- 完全随机的文本和模式:
  - $O(n-m)$

极端情况会怎样？

- 在每一轮进行字符匹配时，模式串的前三个字符a都和主串中的字符相匹配，一直检查到模式串最后一个字符b，才发现不匹配：

第一轮
主串：aaaaaaaaaaaaaaaaab
模式串：aaab

第二轮
主串：aaaaaaaaaaaaaaaaab
模式串：aaab

第三轮
主串：aaaaaaaaaaaaaaaaab
模式串：aaab

……

第N轮
主串：aaaaaaaaaaaaaaaaab
模式串：aaab

两个字符串在每一轮都需要白白比较4次，显然非常浪费。

假设主串的长度是m，模式串的长度是n，那么在这种极端情况下，*BF*算法的最坏时间复杂度是$O(mn)$

极端情况会怎样？

- 在每一轮进行字符匹配时，模式串的前三个字符a都和主串中的字符相匹配，一直检查到模式串最后一个字符b，才发现不匹配：

第一轮

主串： aaaaaaaaaaaaaaaaab

模式串： aaab

第二轮

主串： aaaaaaaaaaaaaaaaab

模式串： aaab

第三轮

主串： aaaaaaaaaaaaaaaaab

模式串： aaab

· · · · · ·

第N轮

主串： aaaaaaaaaaaaaaaaab

模式串： aaab

两个字符串在每一轮都需要白白比较4次，显然非常浪费。

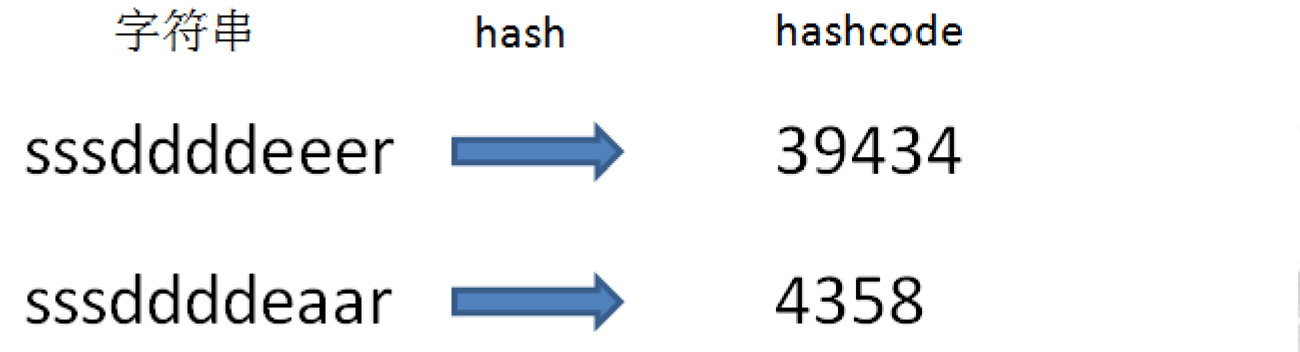
假设主串的长度是*m*，模式串的长度是*n*，那么在这种极端情况下，***BF***算法的最坏时间复杂度是***O(mn)***

是否有更优的匹配算法呢？

本章内容

9.2Rabin-Karp算法

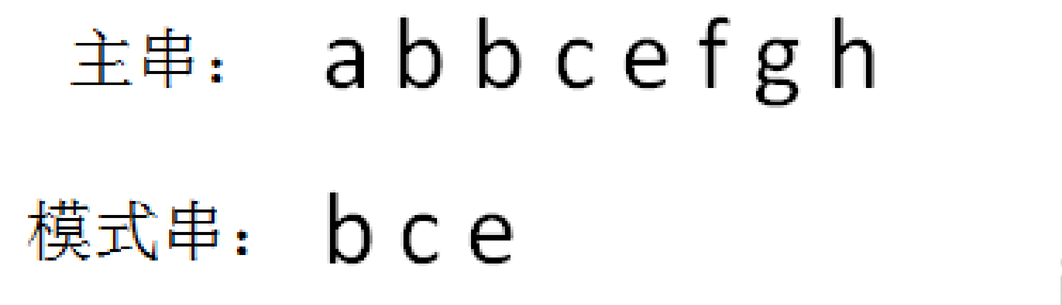
- **Rabin和Karp两位学者命名**
- 思想：比较两个字符串的哈希值
  - 每一个字符串都可以通过某种哈希算法，转换成一个整型数
  - **hashcode：hashcode = hash（string）**
  - 相对于逐个字符比较两个字符串，仅比较两个字符串的**hashcode**要容易得多



字符串	hash	hashcode
sssdddeeer	→	39434
sssdddeaar	→	4358

9.2Rabin-Karp算法

- 给定主串和模式串如下（假定字符串只包含26个小写字母）：
- Step1:生成模式串的hashcode
  - 策略一：按位相加：bce = 2 + 3 + 5 = 10
    - 优点：简单
    - 缺点：很容易产生哈希冲突：bce、bec、cbe的hashcode是一样的

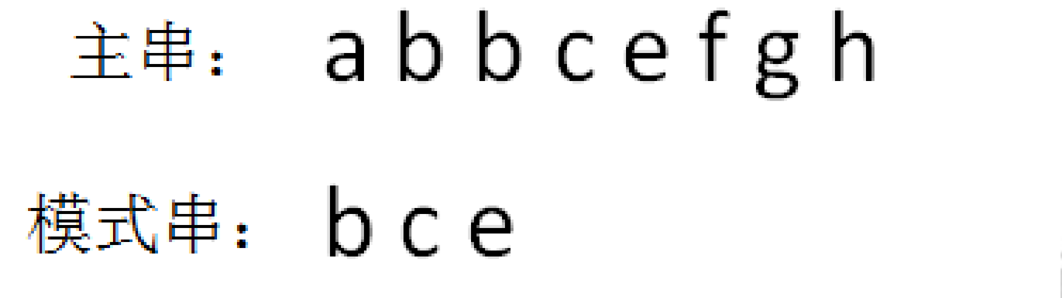


主串： a b b c e f g h

模式串： b c e

9.2Rabin-Karp算法

- 给定主串和模式串如下（假定字符串只包含26个小写字母）：
- **Step1:生成模式串的hashcode**
  - 策略一：按位相加：**bce = 2 + 3 + 5 = 10**
  - 策略二：转换成26进制数：**bce = 2*(26^2) + 3*26 + 5 = 1435**
    - 优点：大幅减少了**hash**冲突
    - 缺点：计算量较大，而且有可能出现超出整型范围的情况，需要对计算结果进行取模

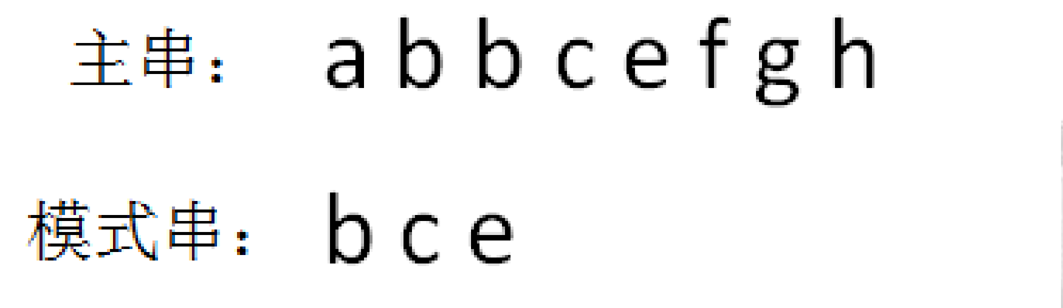


主串： a b b c e f g h

模式串： b c e

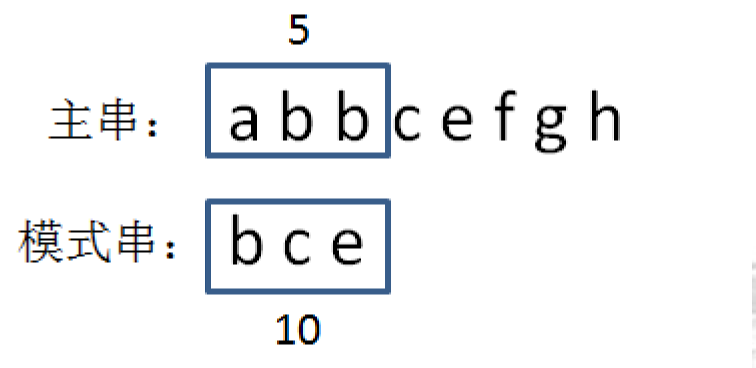
9.2Rabin-Karp算法

- **Step1:生成模式串的hashcode**
  - **策略一：按位相加：bce = 2 + 3 + 5 = 10**
- **Step2:生成主串当中第一个等长子串的hashcode**
  - 由于主串通常要长于模式串，把整个主串转化成hashcode是没有意义的，只有比较主串当中**和模式串等长的子串**才有意义
  - **首先生成主串中第一个和模式串等长的子串hashcode，即abb = 1 + 2 + 2 = 5**



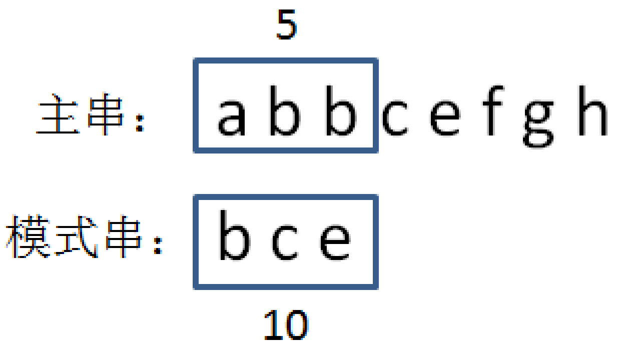
主串： a b b c e f g h

模式串： b c e



9.2Rabin-Karp算法

- **Step1:生成模式串的hashcode**
  - 策略一：按位相加：**bce = 2 + 3 + 5 = 10**
- **Step2:生成主串当中第一个等长子串的hashcode**
- **Step3:比较两个hashcode**
  - 模式串和第一个子串不匹配，继续下一轮比较



5

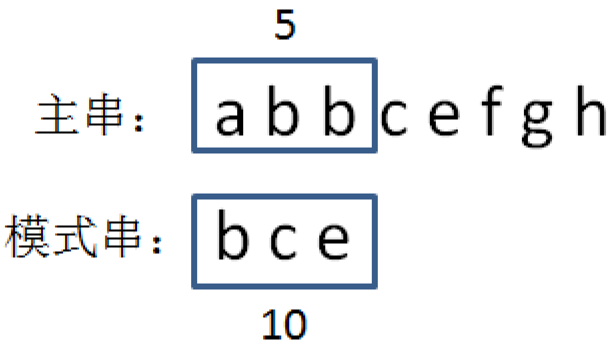
主串： a b b c e f g h

模式串： b c e

10

9.2Rabin-Karp算法

- Step1:生成模式串的hashcode
  - 策略一：按位相加：bce = 2 + 3 + 5 = 10
- Step2:生成主串当中第一个等长子串的hashcode
- Step3:比较两个hashcode
- Step4:生成主串当中第二个等长子串的hashcode，比较
  - 不匹配

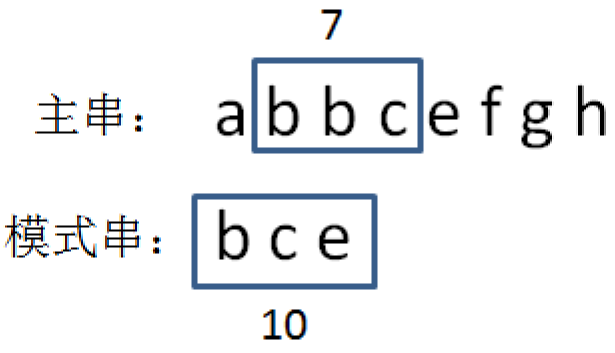


5

主串：abbcefgh

模式串：bce

10



7

主串：abbcefgh

模式串：bce

10

9.2Rabin-Karp算法

- **Step5：生成主串当中第三个等长子串的hashcode，比较**
  - 两个hash值相等！这是否说明两个字符串也相等呢？
  - **逐个字符比较两个字符串：**对两个字符串逐个字符比较，最终判断出两个字符串匹配

10

主串： a b b c e f g h

模式串： b c e

10

主串： a b b c e f g h

模式串： b c e

9.2Rabin-Karp算法：复杂性的思考

- 哈希过程的时间花销：把全部子串进行**hash**的总时间复杂度最坏可达到***O(mn)***
  - 如何优化？
  - 考虑到子串的**hash**计算并不独立！因此可以由上一子串增量计算得到

26

主串：a b b c e f g d e a q f w

?

主串：a b b c e f g d e a q f w

9.2Rabin-Karp算法：复杂性的思考

- 哈希过程的时间花销：把全部子串进行**hash**的总时间复杂度最坏可达到*O(mn)*
  - 如何优化？
  - 考虑到子串的**hash**计算并不独立！因此可以由上一子串增量计算得到
    - 新**hashcode** = 旧**hashcode - 1 + 4 = 26-1+4 = 29**
    - 再下一个子串bcefgde的计算也是同理：
    - **新hashcode = 旧hashcode - 2 + 5 = 29-2+5 = 32**

9.2Rabin-Karp算法：复杂性的思考

- 子串进行**hash**的时间复杂度$O(n)$
- 后续的子串**hash**是增量计算，因此总时间复杂度为$O(n)$
  - 优点：
    - 规避了字符的直接比较，改用**hash**值比较
    - **RK**算法用**hash**比较的方式免去许多无谓的字符比较
  - 缺点：存在哈希冲突：
    - 如果冲突频繁，**RK**算法易退化成**BF**算法

本章内容

9.1概念与定义

9.2Rabin-Karp算法

9.3 BM算法

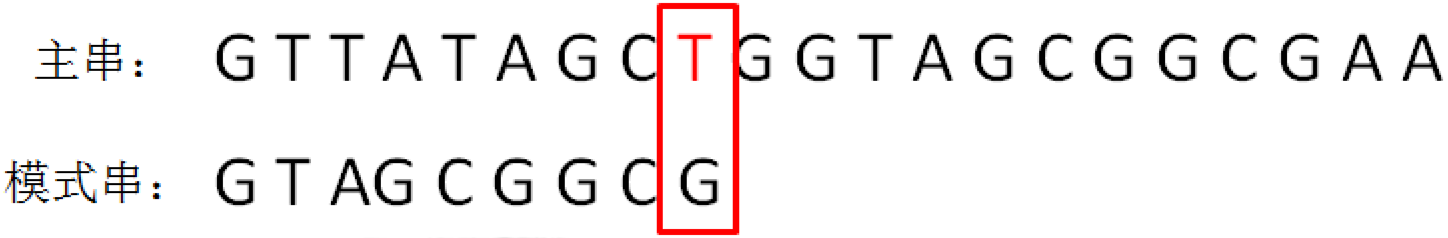
9.4 KMP算法

9.3BM算法

- 如何仍采取字符比较，并减小无谓的比较呢？
- **BM算法命名来源于计算机科学家Bob Boyer和J Strother Moore**
  - 坏字符原则
  - 好后缀原则

9.3BM算法

- 如何仍采取字符比较，并减小无谓的比较呢？
  - **坏字符:**模式串和子串当中不匹配的字符
  - 当模式串和主串的第一个等长子串比较时，子串的最后一个字符T就是坏字符：
    - **注意：BM算法的检测顺序是从最右往左检测(反向检测)**

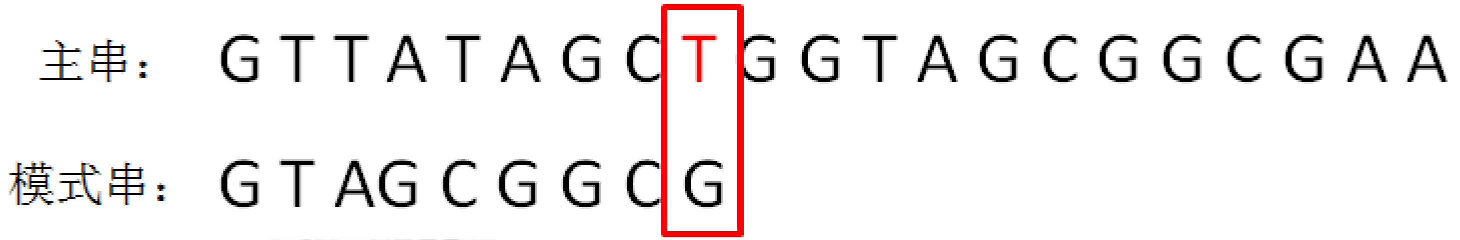


主串： G T T A T A G C T G G T A G C G G C G A A

模式串： G T A G C G G C G

9.3BM算法

- **坏字符原则**
  - 当检测到第一个坏字符之后，不必让模式串一位一位向后挪动和比较！
  - 只有**模式串与坏字符T对齐的位置也是字符T的情况下**，两者才有匹配的可能
  - **如何尽可能向后移动模式串？**

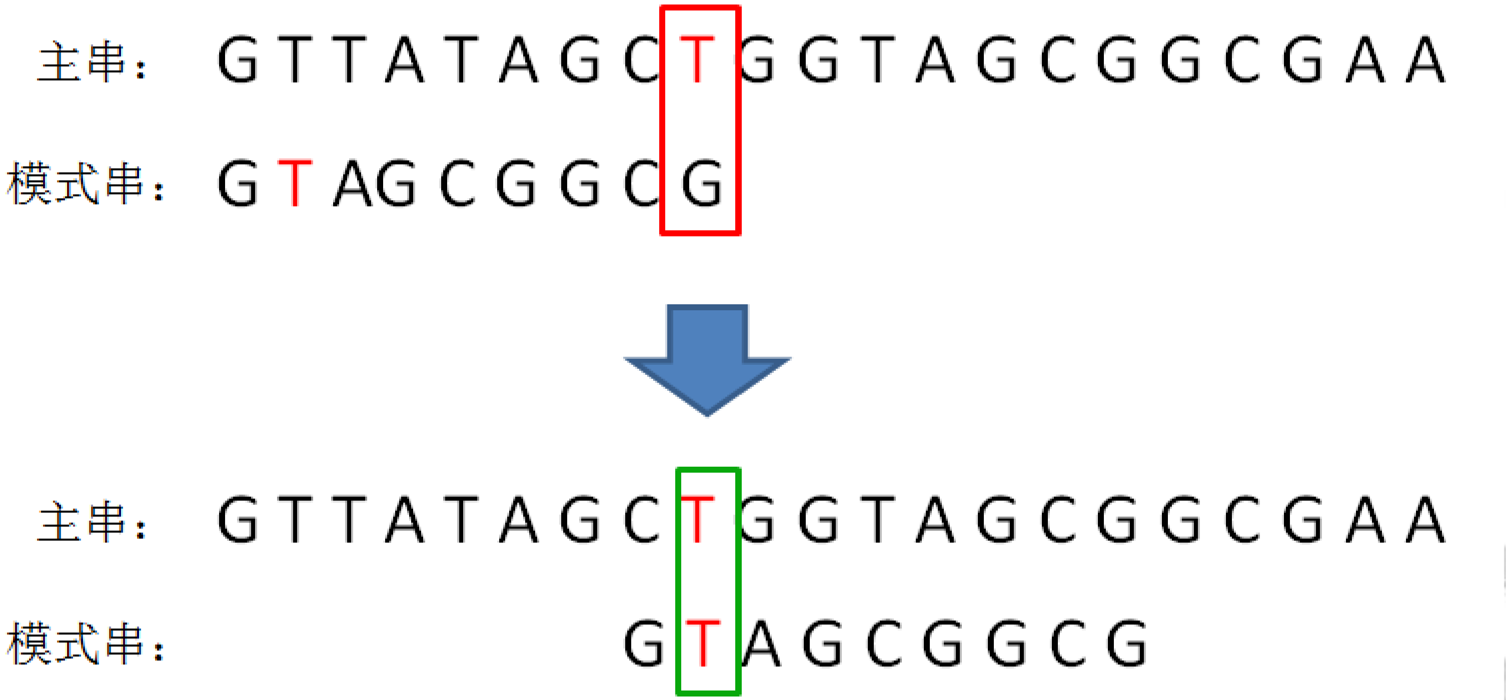


主串： GTTATAGCTGGTAGCGGCGAA

模式串： GTAGCGGCG

9.3BM算法

- **坏字符原则**
  - 直接把模式串当中的字符T和主串的坏字符对齐，进行下一轮的比较
  - 算法亮点：坏字符的位置越靠右，下一轮模式串的挪动跨度就可能越长，节省的比较次数也就越多



主串：GTTATAGCTGGTAGCGGCGAA

模式串：GTAGCGGCG

主串：GTTATAGCTGGTAGCGGCGAA

模式串：GTAGCGGCG

9.3BM算法

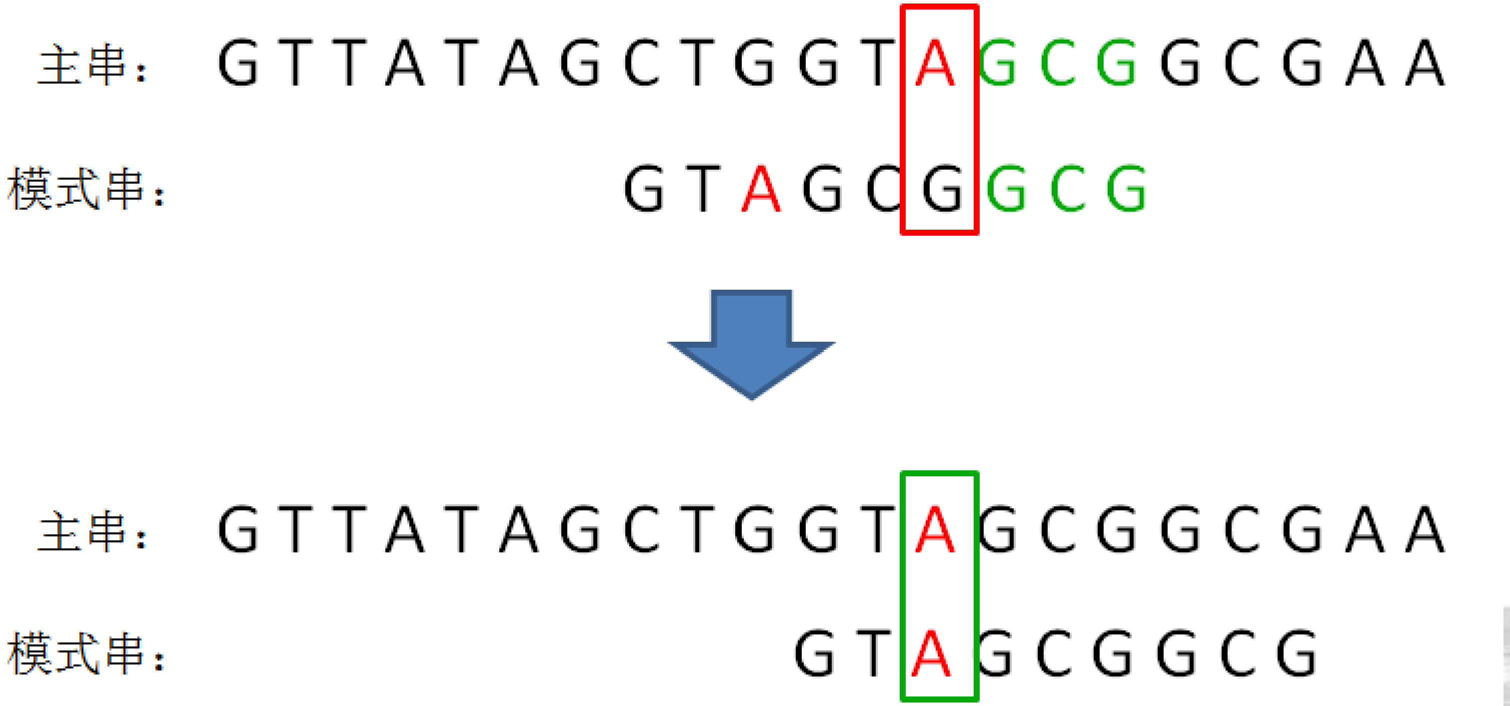
- **坏字符原则**
  - 继续逐个字符比较，发现右侧的G、C、G都是一致的，但主串当中的字符A，是又一个坏字符

主串： G T T A T A G C T G G T A G C G G C G A A

模式串： G T A G C G G C G

9.3BM算法

- **坏字符原则**
  - 找到模式串的A字符，把模式串的字符A和主串中的坏字符对齐，进行下一轮比较：



主串：GTTATAGCTGGTAGCGGCGAA

模式串：GTAGCGGCG

主串：GTTATAGCTGGTAGCGGCGAA

模式串：GTAGCGGCG

9.3BM算法

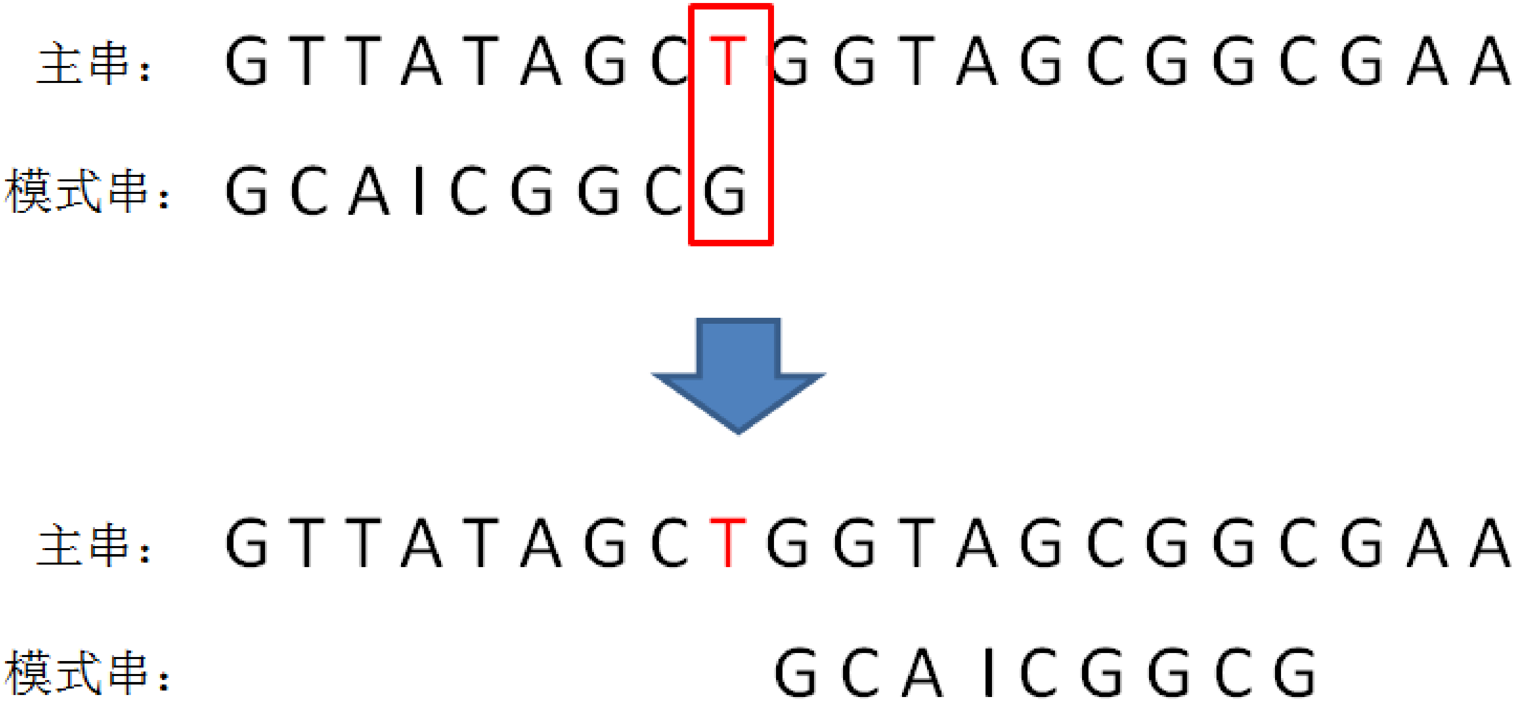
- **坏字符原则**
  - 继续逐个字符比较，发现全部字符都是匹配的，比较完成：

主串： GTTATAGCTGGTAGCGGCGAA

模式串： GTAGCGGCG

9.3BM算法

- 如果坏字符在模式串中不存在，如何移动？
  - 把模式串移动到主串坏字符的下一位：



主串：GTTATAGCTGGTAGCGGCGAA

模式串：GCAICGGCG

主串：GTTATAGCTGGTAGCGGCGAA

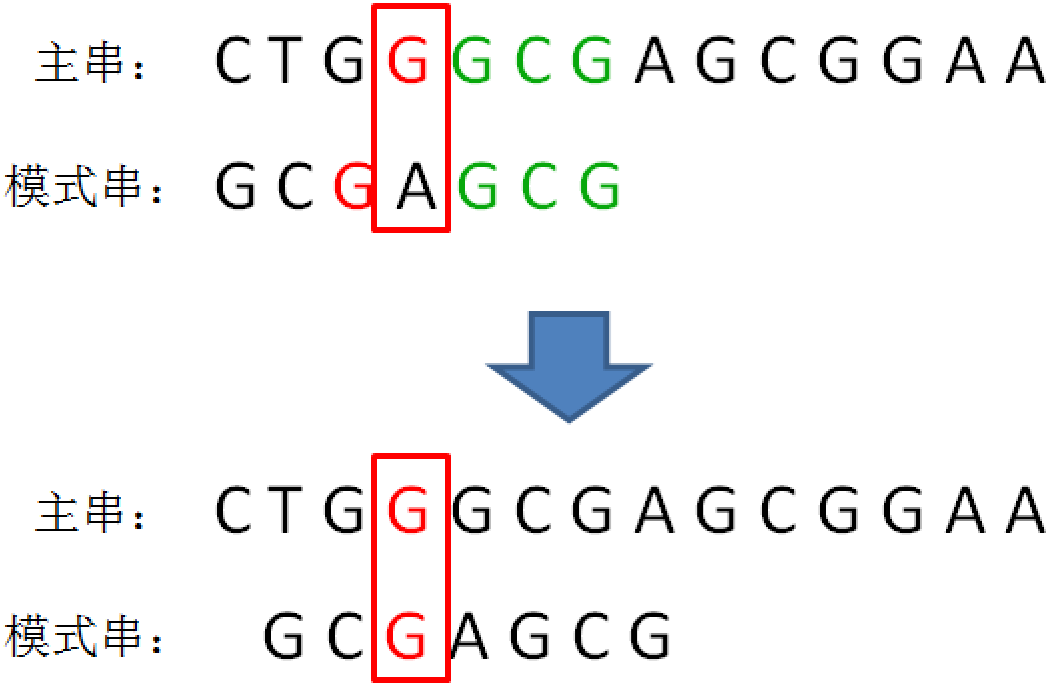
模式串：GCAICGGCG

9.3BM算法

- 如何仍采取字符比较，并减小无谓的比较呢？
  - 坏字符原则
  - **好后缀原则：**模式串和子串当中相匹配的后缀

9.3BM算法

- **好后缀原则：** 模式串和子串当中相匹配的后缀
  - 如果仅利用“坏字符原则”匹配下例会怎样？



主串： C T G G G C G A G C G G A A

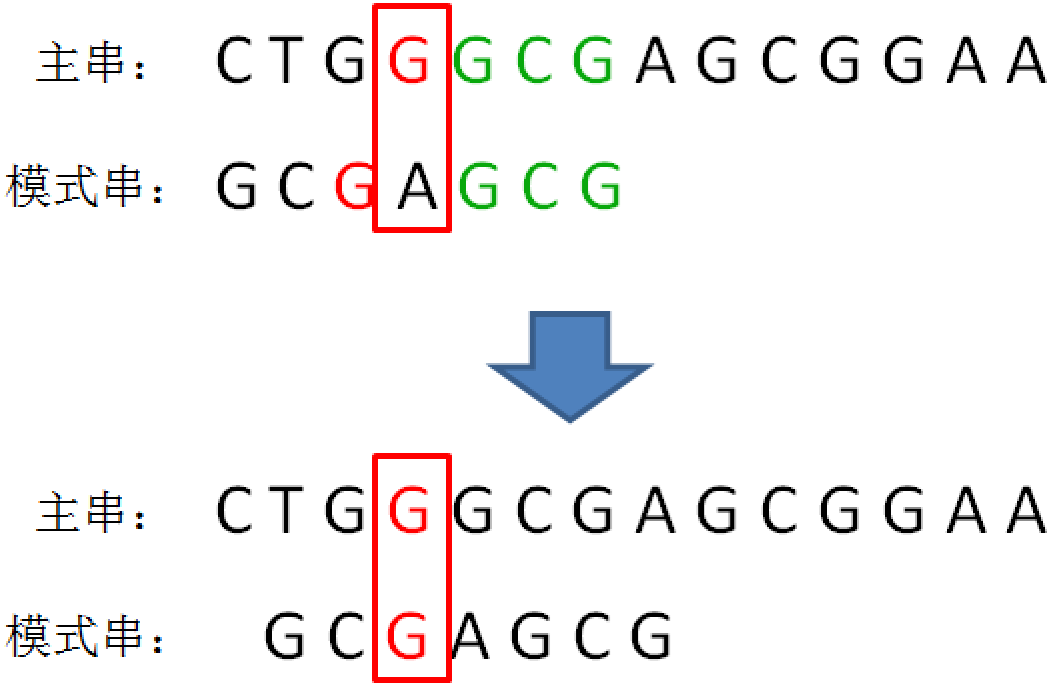
模式串： G C G A G C G

主串： C T G G G C G A G C G G A A

模式串： G C G A G C G

9.3BM算法

- **好后缀原则：** 模式串和子串当中相匹配的后缀
  - 如果仅利用“坏字符原则”匹配下例会怎样？
  - 后面三个字符都是匹配的，到了第四个字符的时候，发现坏字符G，按照坏字符规则，模式串仅仅能够向后挪动一位：



主串： C T G G G C G A G C G G A A

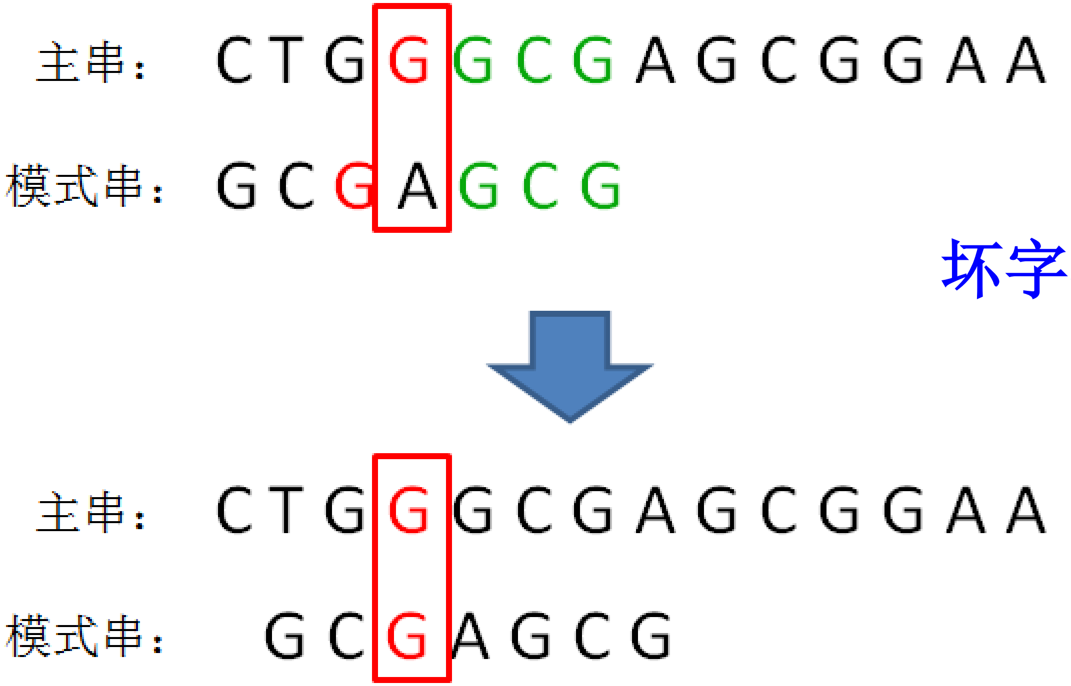
模式串： G C G A G C G

主串： C T G G G C G A G C G G A A

模式串： G C G A G C G

9.3BM算法

- **好后缀原则：** 模式串和子串当中相匹配的后缀
  - 如果仅利用“坏字符原则”匹配下例会怎样？
  - 后面三个字符都是匹配的，到了第四个字符的时候，发现坏字符G，按照坏字符规则，模式串仅仅能够向后挪动一位：



主串：　C T G G G C G A G C G G A A

模式串：G C G A G C G

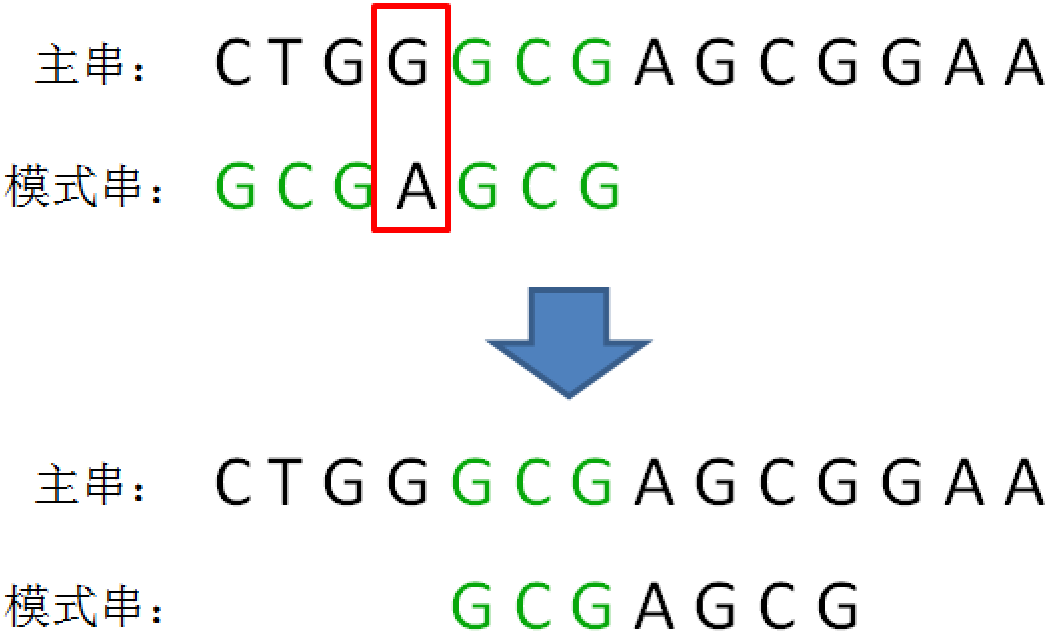
坏字符原则对这种情况的作用不明显！

主串：　C T G G G C G A G C G G A A

模式串：　G C G A G C G

9.3BM算法

- **好后缀原则：模式串和子串当中相匹配的后缀**
  - **为进一步减少比较次数，引入好后缀原则**
  - 主串和模式串都有共同的后缀“GCG”，这就是所谓的“好后缀”
  - 如果模式串其他位置也包含与“GCG”相同的片段，那么我们就可以挪动模式串，让这个片段和好后缀对齐，进行下一轮的比较



主串： C T G G G C G A G C G G A A

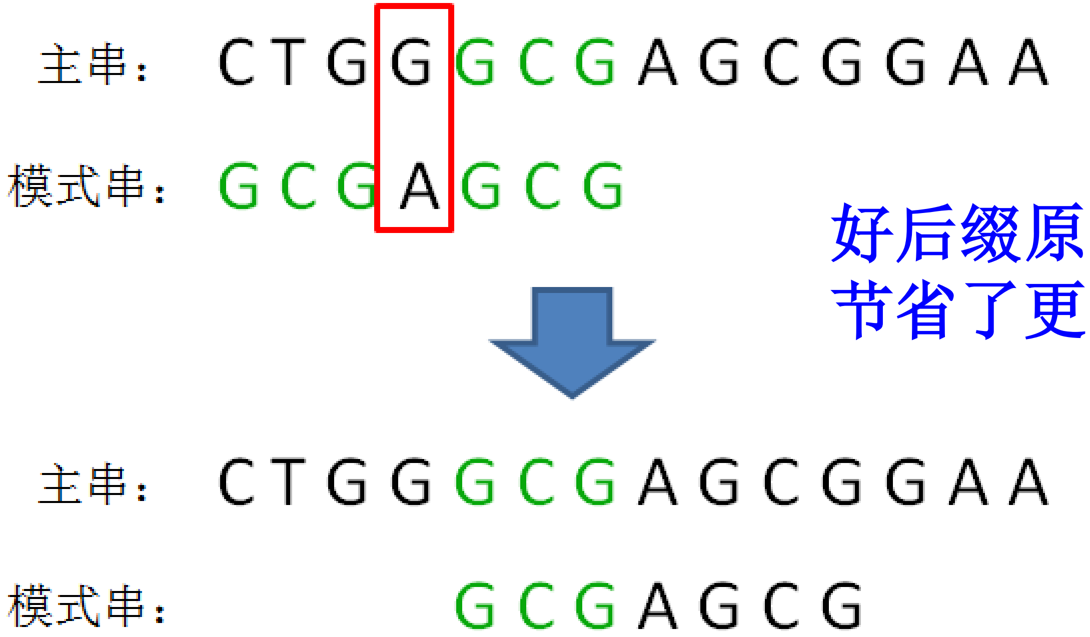
模式串： G C G A G C G

主串： C T G G G C G A G C G G A A

模式串： G C G A G C G

9.3BM算法

- 好后缀原则：**模式串和子串当中相匹配的后缀**
  - **为进一步减少比较次数，引入好后缀原则**
  - 主串和模式串都有共同的后缀“GCG”，这就是所谓的“好后缀”
  - 如果模式串其他位置也包含与“GCG”相同的片段，那么我们就可以挪动模式串，让这个片段和好后缀对齐，进行下一轮的比较


主串： CTGGGCGAGCGGAA
模式串：GCGAGCG

主串： CTGGGCGAGCGGAA
模式串：   GCGAGCG

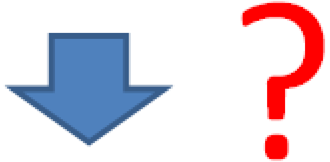
好后缀原则让模式串向后移动更多位，节省了更多无谓的比较。

9.3BM算法

- 如果模式串中不存在与好后缀相同的片段如何处理?
  - 直观：把模式串挪到好后缀之后

主串： T G G **G C G** A G C G G A A

模式串： C G A **G C G**

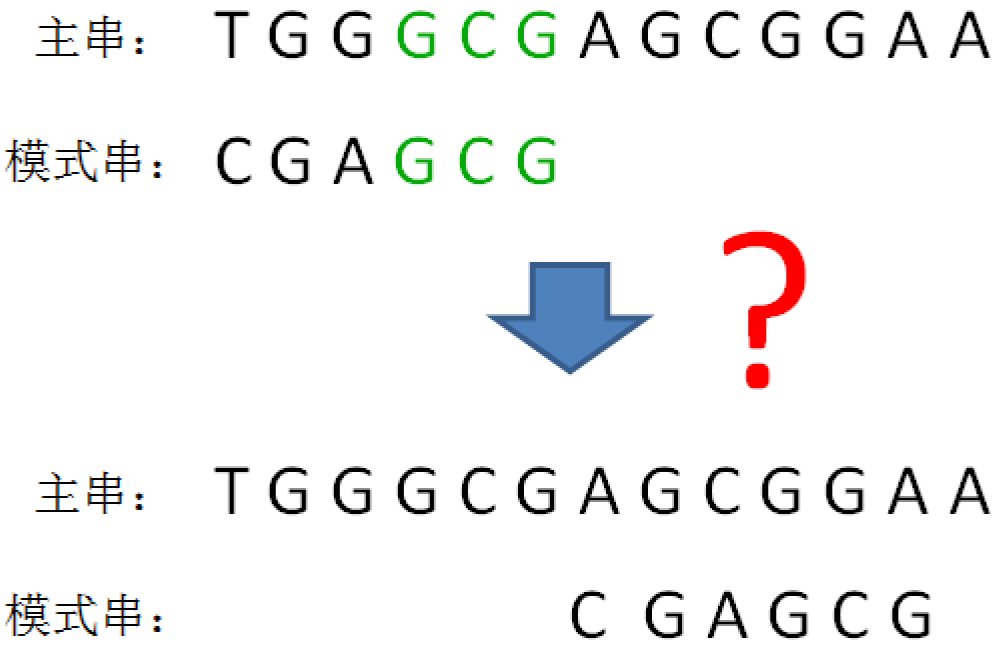


主串： T G G G C G A G C G G A A

模式串： C G A G C G

9.3BM算法

- 如果模式串中不存在与好后缀相同的片段如何处理？
  - ~~直观：把模式串挪到好后缀之后~~
  - 需判断模式串的前缀是否和好后缀的后缀匹配，避免“移动过度”



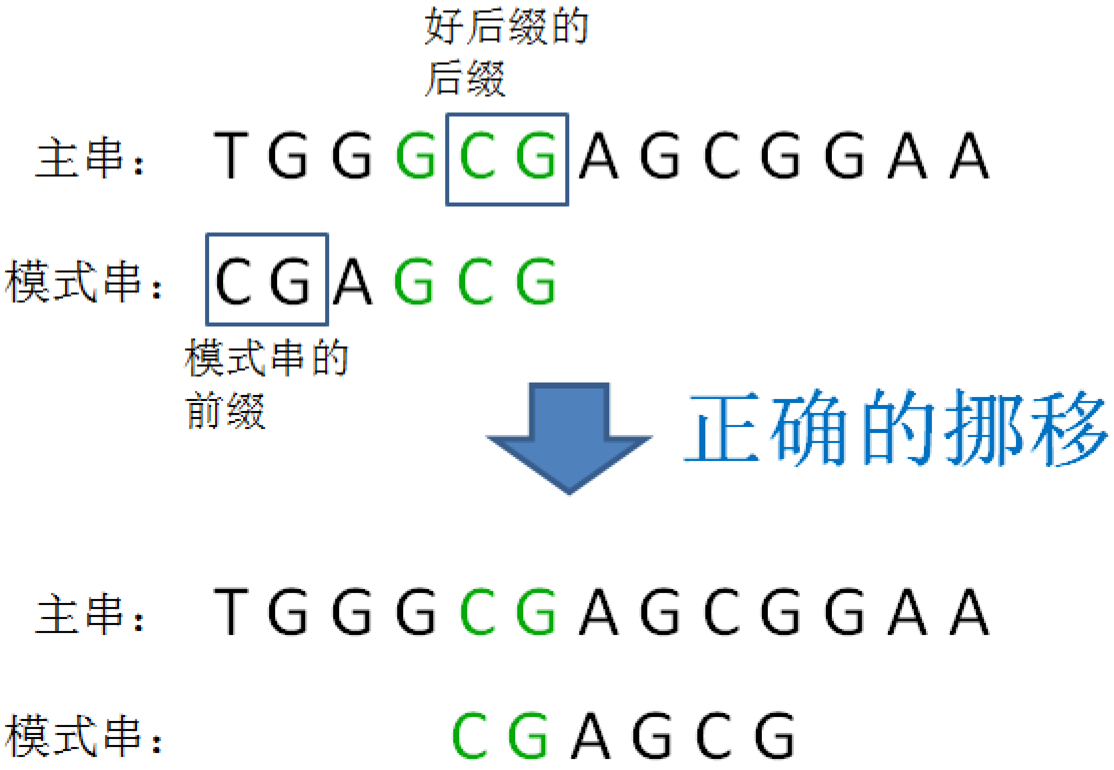
主串： T G G G C G A G C G G A A

模式串： C G A G C G

?

主串： T G G G C G A G C G G A A

模式串： C G A G C G



好后缀的后缀

主串： T G G G C G A G C G G A A

模式串： C G A G C G

模式串的前缀

正确的挪移

主串： T G G G C G A G C G G A A

模式串： C G A G C G

9.3BM算法

- 如何协同使用这两种方式?
  - 坏字符原则
  - 好后缀原则

  - 在每次比较后，分别按坏字符和好后缀计算移动距离，取距离更长者进行移动

本章内容

- 9.1概念与定义
- 9.2Rabin-Karp算法
- 9.3 BM算法
- 9.4 KMP算法

9.4KMP算法

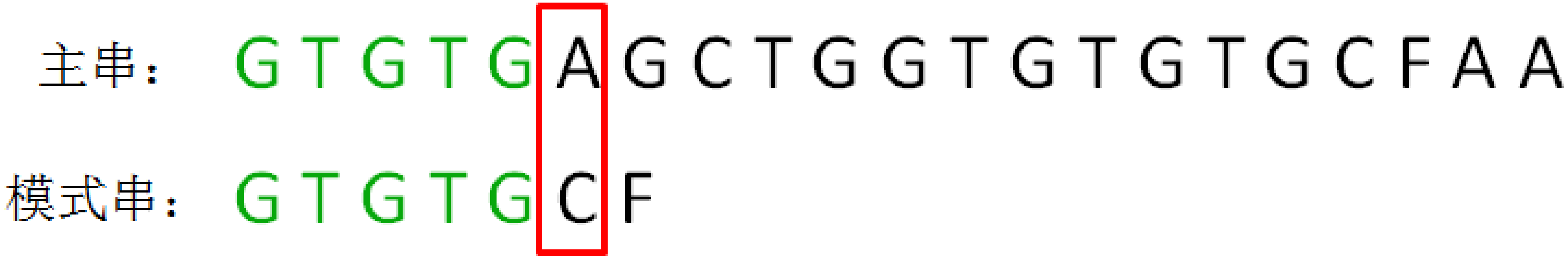
- **命名来源于三位计算机科学家D. B. Kunth, J. H. Morris和V. R. Pratt**
- **目标：让模式串每一轮尽量多移动几位,并更关注已匹配的前缀**
  - 把主串和模式串的首位对齐，从左到右对逐个字符进行比较
  - 模式串和主串的第一个等长子串比较，发现前5个字符都是匹配的，第6个字符不匹配，是一个“坏字符”

主串： GTGTGAGCTGGTGTGTGCFAA

模式串： GTGTGCF

9.4KMP算法

- 目标：让模式串每一轮尽量多移动几位
  - 把主串和模式串的首位对齐，从左到右对逐个字符进行比较
  - 模式串和主串的第一个等长子串比较，发现前5个字符都是匹配的，第6个字符不匹配，是一个“坏字符”
  - 能否对已匹配的前缀很好的利用起来?

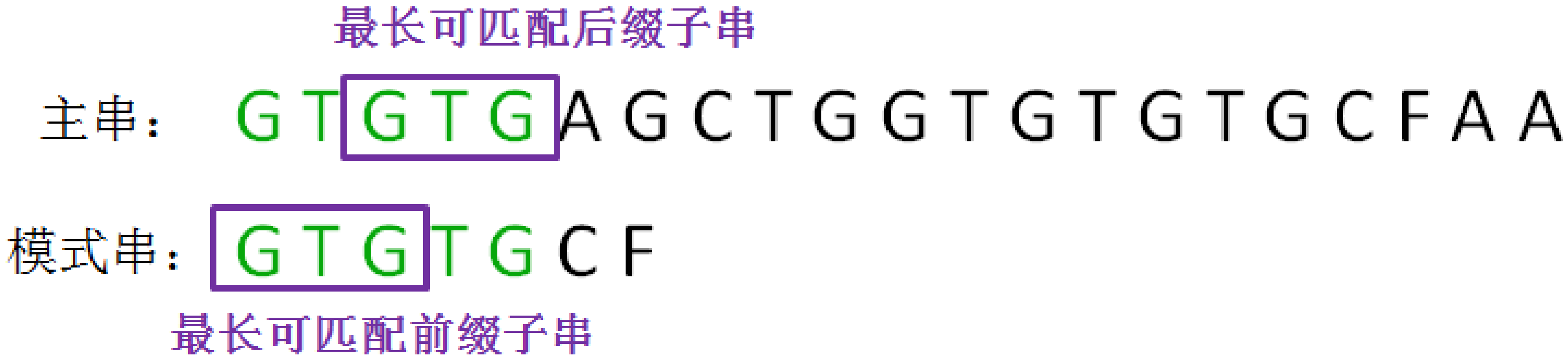


主串： GTGTGAGCTGGTGTGTGCFAA

模式串： GTGTGCF

9.4KMP算法

- **目标：让模式串每一轮尽量多移动几位**
  - 在前缀“GTGTG”当中，后三个字符“GTG”和前三位字符“GTG”是相同的
  - 在下一轮的比较时，只有把这两个相同的片段对齐，才有可能出现匹配
    - **最长可匹配后缀子串**
    - **最长可匹配前缀子串**



最长可匹配后缀子串

主串： G T G T G A G C T G G T G T G T G C F A A

模式串： G T G T G C F

最长可匹配前缀子串

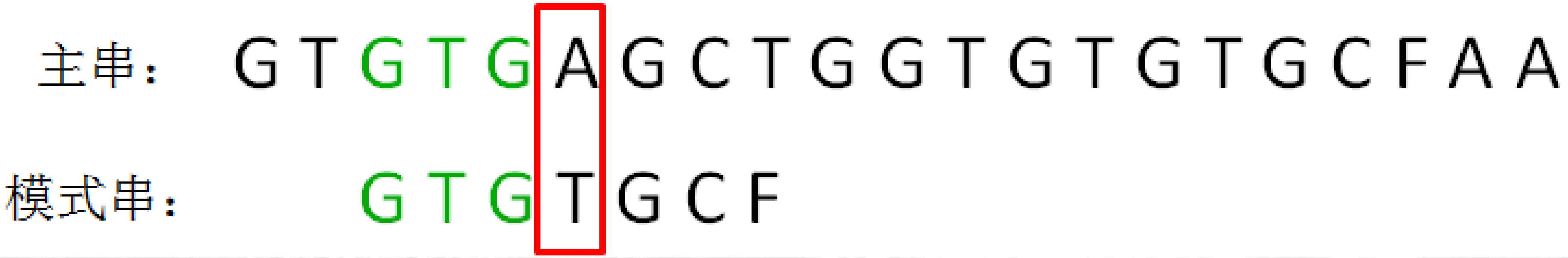
9.4KMP算法

- **目标：让模式串每一轮尽量多移动几位**
  - 直接把模式串向后移动两位，让两个“GTG”对齐，继续从刚才主串的坏字符A开始进行比较

主串：　G T G T G A G C T G G T G T G T G C F A A

模式串：　G T G T G C F

  - 主串的字符A仍然是坏字符，这时候的匹配前缀缩短成了GTG



主串：　G T G T G A G C T G G T G T G T G C F A A

模式串：　G T G T G C F

9.4KMP算法

- **目标：让模式串每一轮尽量多移动几位**
  - 重新确定最长可匹配后缀子串和最长可匹配前缀子串

最长可匹配后缀子串

主串： G T G T G A G C T G G T G T G T G C F A A

模式串： G T G T G C F

最长可匹配前缀子串

9.4KMP算法

- **目标：让模式串每一轮尽量多移动几位**
  - 再次把模式串向后移动两位，让两个“G”对齐，继续从刚才主串的坏字符A开始进行比较
  - 在已匹配的前缀当中寻找到**最长可匹配后缀子串**和**最长可匹配前缀子串**，在下一轮直接把两者对齐，从而实现模式串的快速移动

主串：G T G T G A G C T G G T G T G T G C F A A

模式串：G T G T G C F

9.4KMP算法

- 目标：让模式串每一轮尽量多移动几位
  - 关键问题：如何高效地找最长**可匹配后缀子串和**最长**可匹配前缀子串**
  - **算法流程：**

1. 对模式串预处理，生成next数组

2. 进入主循环，遍历主串

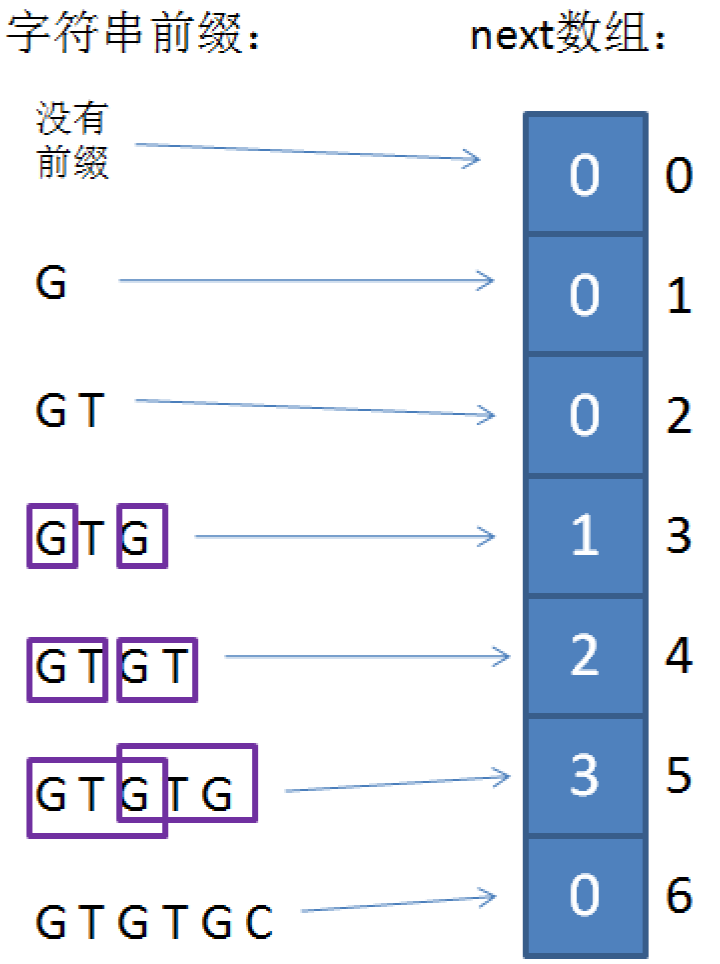
2.1. 比较主串和模式串的字符

2.2. 如果发现坏字符，查询next数组，得到匹配前缀所对应的最长可匹配前缀子串，移动模式串到对应位置

2.3.如果当前字符匹配，继续循环

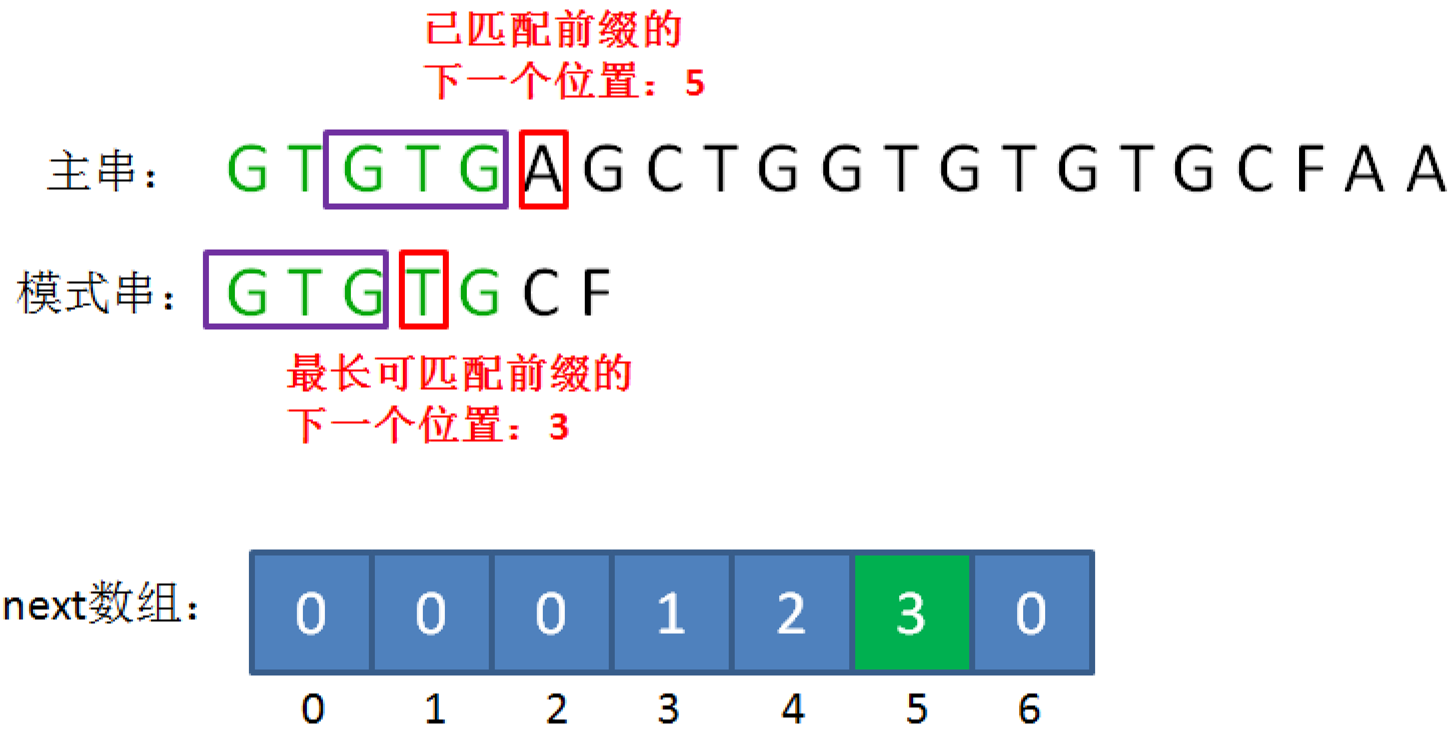
9.4KMP算法

- 关键问题：next数组的设计和实现
  - next：一维整型数组
  - 数组的下标代表了“已匹配前缀的下一个位置”
  - 元素的值则是“最长可匹配前缀子串的下一个位置”



9.4KMP算法

- **关键问题：next数组的设计和实现**
  - 有了next数组，通过已匹配前缀的下一个位置（坏字符位置），快速寻找到最长可匹配前缀的下一个位置，然后把这两个位置对齐。
- 通过坏字符下标5，可以找到next[5]=3，即最长可匹配前缀的下一个位置：



已匹配前缀的下一个位置：5

主串：G T G T G A G C T G G T G T G T G C F A A

模式串：G T G T G C F

最长可匹配前缀的下一个位置：3

next数组：	0	0	0	1	2	3	0
	0	1	2	3	4	5	6

Knuth-Morris-Pratt 算法

```
KMP-Search(T,P)
π ← Compute-Prefix-Table(P)
q ← 0          // number of characters matched
for i ← 0 to n-1   // scan the text from left to right
    while q > 0 and P[q] ≠ T[i] do
        q ← π[q]
    if P[q] = T[i] then q ← q + 1
    if q = m then return i – m + 1
return –1
```

- **Compute-Prefix-Table是P上执行KMP算法的本质.**

KMP的分析

- 最坏运行时间: $O(n+m)$
  - 生成next数组：$O(m)$
  - 主算法(对主串的遍历): $O(n)$
  - 整体：$O(m+n)$
- 空间:
  - KMP算法用了next数组
  - $O(m)$