一种基于口胡的KMP算法介绍（雾）

23.计科.周致远

QQ = 2091079816

众所周知，字符串模式匹配是计算机科学最为古老的问题之一。朴素的字符串匹配算法可以描述为：对于待匹配字符串（下称之为文本）的每一个位置，逐个字符比较从该位置开始是否与模式串（下称之为模板）完全匹配。时间复杂度为$O(n*m)$，十分低效。于是，我斗胆用我笨拙的语言介绍一种更快、更优雅，而且基本是线性复杂度的字符串匹配算法——KMP算法(The Knuth-Morris-Pratt Algorithm)。

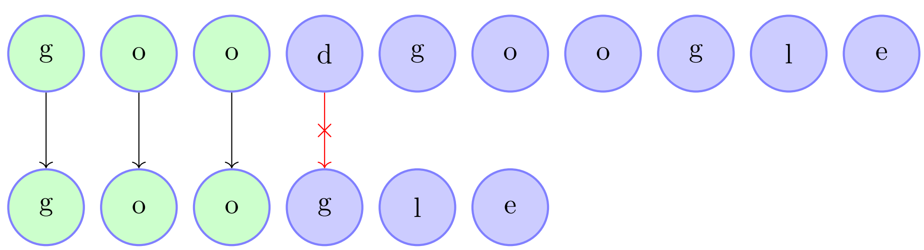
KMP的思想是充分利用朴素算法忽略的信息来加快匹配。假设有如下文本：

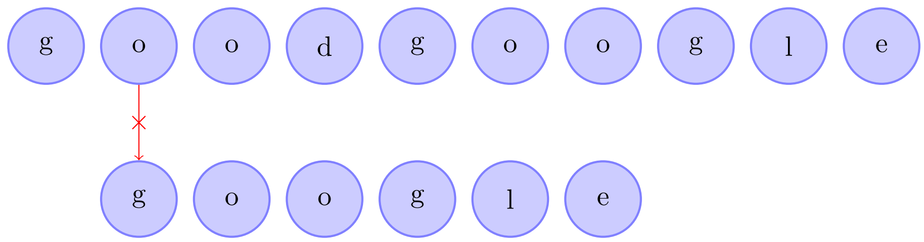
$$goodgoogle$$

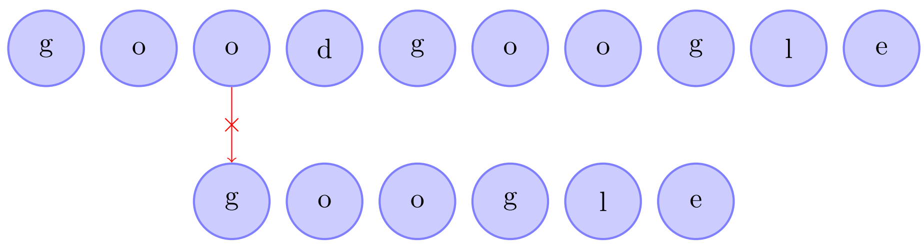
和以下模板：

$$google$$

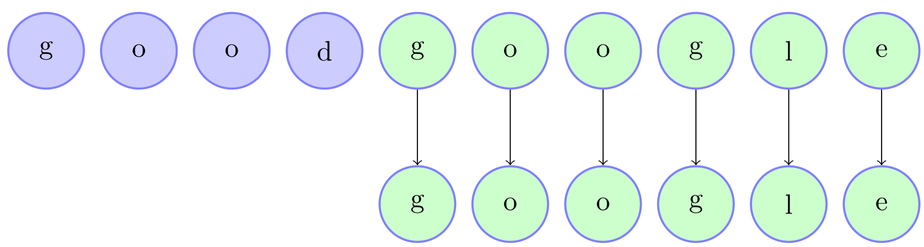
则朴素的方法可表示为：







2 steps later...



设文本表示为 $\{s_1, s_2, ..., s_1 0\}$；模板表示为 $\{t_1, t_2, ..., t_6\}$（下同），我们发现，即便出现三个字符"goo"="goo"匹配的特殊情况，朴素算法也会无视这一点，直接跳到文本下一位和模板初始位。实际上，其中仍有可挖掘的信息：部分匹配给予了算法关于文本的一部分知识。基于朴素匹配算法，设模板长度为n，文本长度为n_t，如果把模板看成一个n维向量：

$$T = \begin{bmatrix} \mathbf{g} \\ \mathbf{o} \\ \mathbf{o} \\ g \\ l \\ e \end{bmatrix}$$

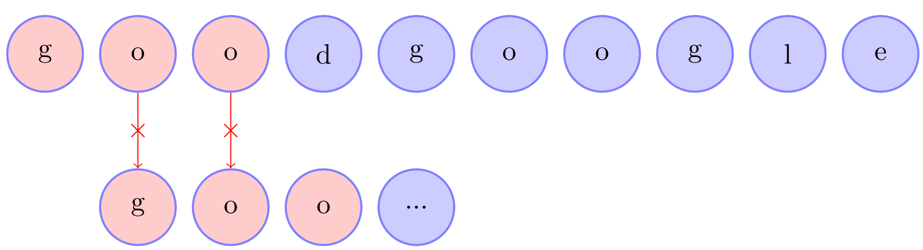
把朴素算法匹配文本的过程看成n维空间：

$$S = \begin{bmatrix} \mathbf{g} & \mathbf{o} & \mathbf{o} & d & g & o \\ o & o & d & g & o & o \\ o & d & g & o & o & g \\ d & g & o & o & g & l \\ g & o & o & g & l & e \end{bmatrix}$$

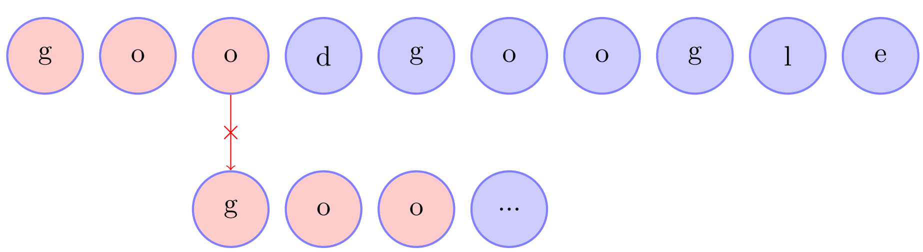
字符串模式匹配$S \times T$过程中观察矩阵S，第一行已经匹配出”goo”，接下来要匹配的几行的前三个字符与”goo”都不完全相同：

$$\begin{bmatrix} \mathbf{g} & \mathbf{o} & \mathbf{o} & d & g & o \\ \mathbf{o} & \mathbf{o} & \mathbf{d} & g & o & o \\ \mathbf{o} & \mathbf{d} & \mathbf{g} & o & o & g \\ \mathbf{d} & \mathbf{g} & \mathbf{o} & o & g & l \\ g & o & o & g & l & e \end{bmatrix}$$

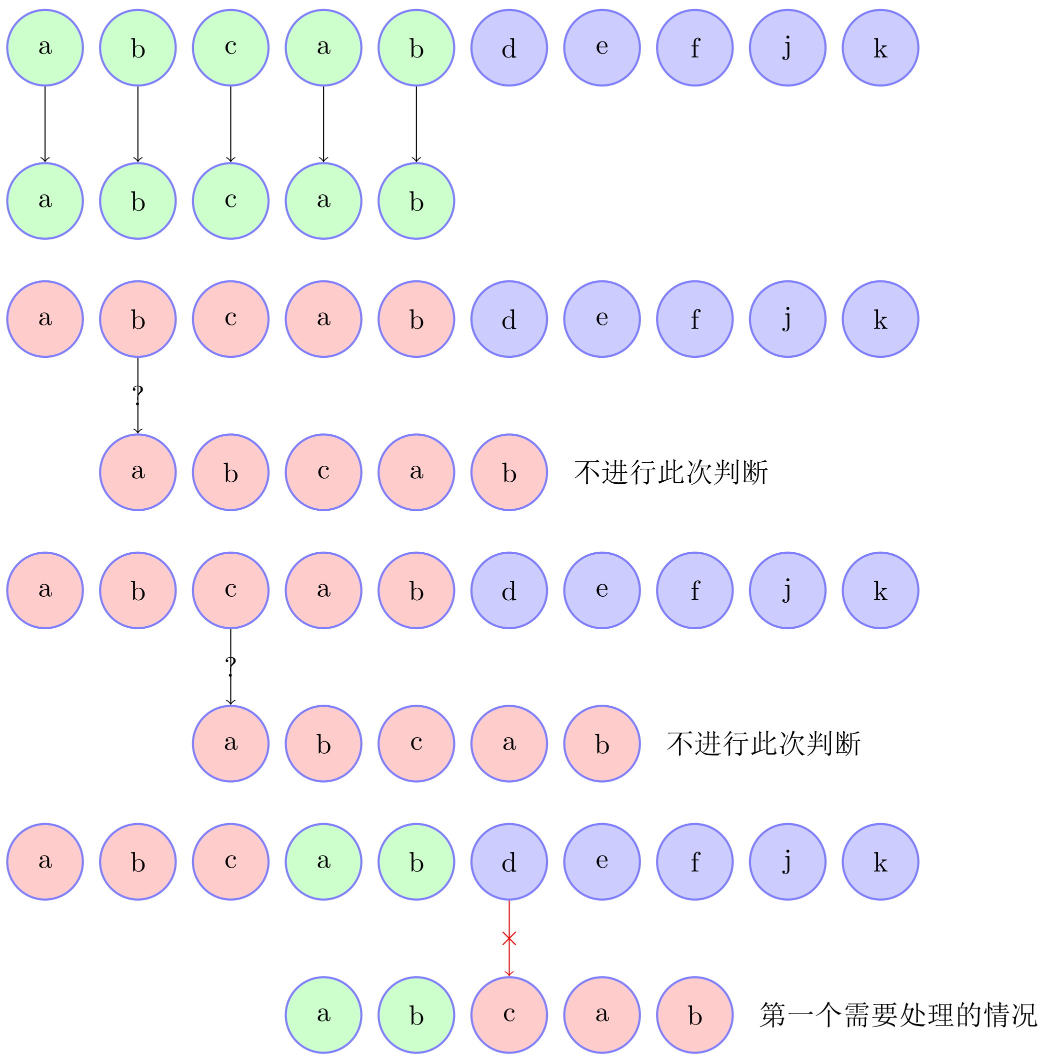
若称从开头开始的连续子串为前缀，从末尾开始的连续子串为后缀,标为红色的元素与模板的后缀相同，而且接下来还要和模板前缀匹配。因此，由于前缀匹配是完整匹配的必要条件，如果能让模板自己的后缀和自己前缀比较，那么就可以在不处理文本接下来字符的情况下直接略过下几次判断，来找到满足必要条件的需要处理的情况：

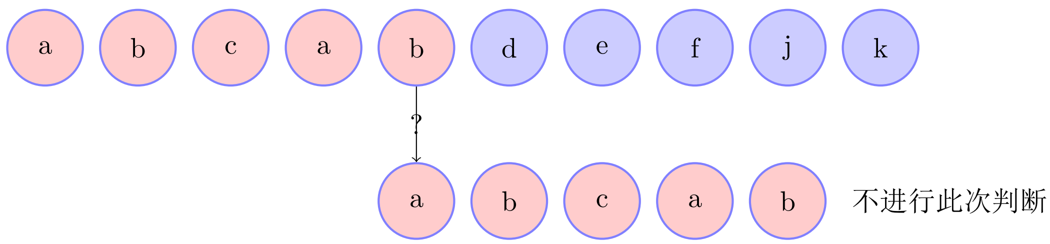


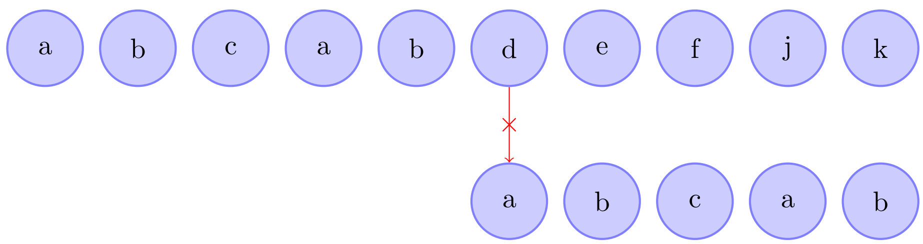
和



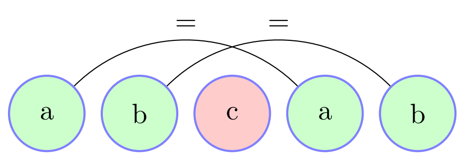
滑动的过程中，我们发现”oo”和”go”分别是部分匹配的后两位、前两位；”o”和”g”则分别是最后一位、第一位，正是模版的子串的长度为2、1的前后缀。于是可以看出，**根据已知信息的判断其实是判断已知部分的前后缀是否相等！**如果相等，那么在这个位置可以匹配，否则不需要考虑。更好的是，模版子串的前后缀是否相等是可以进行预处理得到的，我们可以只处理那些相等的情况。而这就是KMP算法的主要思想所在，如果我们能预先处理出模板每个子串的最大前后缀长度，就能预先判断每次部分匹配失败后下次第一个需要处理的情况。让我们再看一组例子：







此时满足：



称"abcab"有长度为2的最大相等前后缀。遇到不匹配的情况，直接跳转到已经匹配部分之后的下一个字符，让最大相等前后缀来决定此时模板的指针在何处。无论如何，**每一个文本字符只被匹配一次**（已经匹配部分不会再次匹配 ~~kmp也是线性筛~~），时间复杂度为线性，这就是kmp的原理。

具体而言，命题：

文本字符串$\{s_1, s_2 ... s_k, ..., s_{k+q-1}, ..., s_n\}$和模版字符串$\{t_1, ..., t_q, ... t_m\}$的匹配部分 $\{t_1, ..., t_q\}$的最大相同前后缀长度为max,则$\{s_k, ..., s_{k+max-1}\}$到$\{s_{k+max-1}, ..., s_{k+max+max-2}\}$一定不能与$\{t_1, ..., t_{max}\}$匹配。

一定成立。否则，会有新的最大相同前后缀。

对模版进行预处理，每个前缀存储它的最大相等前后缀长度，设next[i]为模版的前i个字符的最大相同前后缀长度，此时有如下转移方程：

$$\begin{cases} next[i] = next[i-1] + 1,\ t_i = t_{next[i-1]} \\ next[i] = next[next[i-1]] + 1,\ t_i = t_{next[next[i-1]]} \end{cases} \tag{1}$$

当最大相等前后缀可以延续时，下一个next加一；当无法延续时，尝试使用上一个最大相等前后缀A的最大相等前后缀B来延续（因为B仍是当前子串相等前后缀之一）。获得next数组后，在进行文本匹配时就可以：

- 匹配成功：继续匹配，匹配成功子串长度+1
- 匹配失败：继续匹配，但模版指针跳到next[成功子串长度]位置

可以看出，无论匹配是否成功，算法总会跳到文本串的下一个字符，并且根据最长相同前后缀确定模版串指针位置，这使得复杂度成为线性。根据以上解析，可写出预处理模版串代码：

```
void getnext(){
 next[0] = 0;
 for(int i = 1;i < ns; i++){
   if(sub[i] == sub[next[i-1]]) next[i] = next[i-1]+1;
   else if(next[i-1] == 0) continue;
   else if(sub[next[next[i-1]-1]] == sub[i]){
  next[i] = next[next[i-1]-1] + 1;
   }
   else if(sub[0] == sub[i]) next[i] = 1;
 }
  }
```

以及匹配过程：

```
 void kmp(){
  int cur = 0;
  for(int i = 0 ; i < n ; i++){
   if(str[i] == sub[cur]){
    cur ++;
    if(cur == ns){
     cout << i-ns+2 << endl;
     cur = next[cur-1];
    }
   }
   else if(cur != 0){
    cur = next[cur-1];
```

```
    i--;
   }
  }
  return;
 }
```

时间复杂度$O(n+m)$。值得一提的是，kmp算法在数据库、ide、文本编辑器中广泛使用，你的vscode中或许就不时跑着这个算法。至此，KMP算法介绍完毕。