HONG − LOU 2022 − 8 − 23

来学KMP怎么还能遇到巨神犇啊，$Matrix67$

$orzorz$

一、何谓模式串匹配

模式串匹配，就是给定一个需要处理的文本串（理论上应该很长）和一个需要在文本串中搜索的模式串（理论上长度应该远小于文本串），查询在该文本串中，给出的模式串的出现有无、次数、位置等。

模式串匹配的意义在于，如果我是一个平台的管理员，我可以针对一篇文章或者一句话，搜索其中某个特定脏字或者不雅词汇的出现次数、位置——次数可以帮助我决定采取何种等级对于该用户的惩罚方式，而位置则可以帮助我给每一个脏词打上“*”的标记来自动屏蔽这些脏词。

二、浅析 KMPKMP 之思想

Knuth(D. E. Knuth)&Morris(J. H. Morris)&Pratt(V. R. Pratt)

首先要理解，朴素的单模式串匹配大概就是枚举每一个文本串元素，然后从这一位开始不断向后比较，每次比较失败之后都要从头开始重新比对，大概期望时间复杂度在 $\Theta(n+m)$左右，对于一般的弱数据还是阔以跑的了滴。但是其实是可以被卡成$\Theta(nm)$的。emm并且还是比较容易卡的。

而 KMPKMP 的精髓在于，对于每次失配之后，我都不会从头重新开始枚举，而是根据我已经得知的数据，从某个特定的位置开始匹配；而对于模式串的每一位，都有唯一的“特定变化位置”，这个在失配之后的特定变化位置可以帮助我们利用已有的数据不用从头匹配，从而节约时间。

比如我们考虑一组样例：

模式串：abcab
文本串：abcacababcab

首先，前四位按位匹配成功，遇到第五位不同，而这时，我们选择将abcababcab向右移三位，或者可以直接理解为移动到模式串中与失配字符相同的那一位。可以简单地理解为，我们将两个已经遍历过的模式串字符重合，导致我们可以不用一位一位地移动，而是根据相同的字符来实现快速移动。

```
模式串:     abcab
文本串：abcacababcab
```

但有时不光只会有单个字符重复：

```
模式串：abcabc
文本串：abcabdababcabc
```

当我们发现在第六位失配时，我们可以将模式串的第一二位移动到第四五位，因为它们相同qwq。

```
模式串:     abcabc
文本串：abcabdababcabc
```

那么现在已经很明了了，KMP的重头戏就在于用失配数组来确定当某一位失配时，我们可以将前一位跳跃到之前匹配过的某一位。而此处有几个先决条件需要理解：

1、我们的失配数组应当建立在模式串意义下，**而不是文本串意义下。** 因为显然模式串要更加灵活，在失配后换位时，更灵活简便地处理。

2、如何确定位置呢？

首先我们要明白，基于先决条件 1 而言，我们在预处理时应当考虑当**模式串**的第i位失配时，应当跳转到哪里. 因为在文本串中，之前匹配过的所有字符已经没有用了——都是匹配完成或者已经失配的，所以我们的KMP数组（即是用于确定失配后变化位置的数组，下同）应当记录的是：

在模式串$str1$中，对于每一位$str1(i)$, 它的KMP数组应当是记录一个位置j，$j \leq i$并且满足$str1(i) = str1(j)$并且在$j \neq i$时理应满足$str1(1)$至$str1(j-1)$分别与$str(i-j+1) \rightarrow str(i-1)$按位相等

上述即为移位法则

3、从前缀后缀来解释KMP ：

首先解释前后缀(因为太简单就不解释了qwq)：

给定串：ABCABA
前缀：A,AB,ABC,ABCA,ABCAB,ABCABA
后缀：A,BA,ABA,CABA,BCABA,ABCABA

其实刚才的移位法则就是对于模式串的每个前缀而言，用kmp数组记录到它为止的模式串前缀的真前缀和真后缀最大相同的位置（注意，这个地方没有写错，是真的有嵌套qwq）。然而这个地方我们要考虑“模式串前缀的前缀和后缀最大相同的位置”原因在于，我们需要用到kmp数组换位时，当且仅当未完全匹配。所以我们的操作只是针对模式串的前缀——毕竟是失配函数，失配之后只有可能是某个部分前缀需要“快速移动”。所以这就可以解释KMP中前后缀应用的一个特点：

KMP中前后缀不包括模式串本身，**即只考虑真前缀和真后缀，**因为模式串本身需要整体考虑，当且仅当匹配完整个串之后；而匹配完整个串不就完成匹配了吗qwq

三、代码实现

$kmp[i]$用于记录当匹配到模式串的第i位之后失配，该跳转到模式串的哪个位置，那么对于模式串的第一位和第二位而言，只能回跳到i，因为是 KMP是要将真前缀跳跃到与它相同的真后缀上去（通常也可以反着理解），所以当$i=0$或者$i=0$时，相同的真前缀只会是$str1(0)$这一个字符，所以$kmp[0]=kmp[1]=1$。

2、对于如何和文本串比对，很简单：

```
for(int i = 1, j = 0; i <= a.length(); i++) {
  while(j && b[j + 1] != a[i]) j = kmp[j];

  if(b[j + 1] == a[i]) j++; // 如果匹配成功，对应的模式串位置后移

  if(j == b.length()) {
    std::cout << i - b.length() + 1;
    j = kmp[j]; // 继续匹配
  }
}
```

3、那么我们该如何处理kmp数组呢？我们可以考虑用模式串自己匹配自己

```
for(int i = 2, j= 0; i < b.length(); i++) {
  while(j && b[i] != b[j + 1]) j = kmp[j];
    // 此处判断 j 是否为0的原因在于如果回跳到第一个字符就不用再回跳了
    // 通过自己匹配自己来得出每一个点的kmp值
  if(b[j + 1] == b[i]) j++;

  kmp[i] = j; // i + 1后应该如何跳
}
```

那么这个“自己匹配自己”该如何理解呢？我们可以这么想： 首先，在单次循环只有一个if来判断的原因在于每次至多向后多求一位的$next$;

并且j是拥有可继承性的，由于j是用于比对前缀后缀的，那么对于一组前后缀而言，第$i-1$和第$j-1$位之前均相同或者有不同，决定着i和j匹配的结果是从0开始还是基于上一个j继续++

贴标程:

```cpp
#include <bits/stdc++.h>

const int maxn = (int)1e6 + 7;

int kmp[maxn];

char a[maxn], b[maxn];

int main() {
  std::cin >> a + 1;
  std::cin >> b + 1;

  int lena = a.length();
  int lenb = b.length();

  for(int i = 2, j = 0; i <= lenb; i++) {
    while(j && b[i] != b[j + 1]) j = kmp[j];

    if(b[j + 1] == b[i]) j++;
    kmp[i] = j;

  }

  for(int i = 1, j = 0; i <= lena; i++) {
    while(j && b[j + 1] != a[i]) j = kmp[j];

    if(b[j + 1] == a[i]) j++;

    if(j == lenb) {
      std::cout << i - lenb + 1 << "\n";
      j = kmp[j];
    }
  }

  for(int i = 1; i <= lenb; i++) {
    std::cout << kmp[i] << " \n"[i == lenb];
  }
  return 0;
}
```

那么时间复杂度为$\Theta(m+n)$,比朴素算法有了极大的优化。

$ExtraKnowledge$浅析复杂度证明

题外话：本来想扯摊还分析来着，但是**神犇**说的好像比较直接易懂，于是在这里就引用了**神犇**的话：

每次位置指针$i++$时，失配指针j至多增加$lenb$次，从而至多减少len次，所以就是$\Theta(lenN+lenM)=\Theta(N+M)$的。

其实我们也可以发现，KMP算法之所以快，不仅仅由于它的失配处理方案，更重要的是利用前缀后缀的特性，从不会反反复复地找，我们可以看到代码里对于匹配只有一重循环，也就是说KMP算法具有一种“最优历史处理”的性质，而这种性质也是基于KMP的核心思想的。