参考博客：<http://blog.csdn.net/v_july_v/article/details/7041827>

一.首先明确KMP算法是干嘛的

在目标串(长度为n)中查找子串(长度为m)

为什么需要他

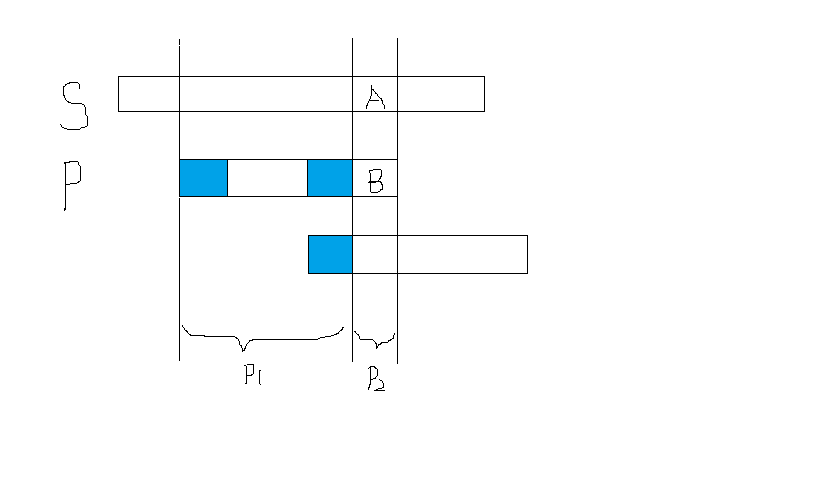
暴力匹配算法的时间复杂度是0(nm)

而KMP算法的时间复杂度是0(n+m)

所以为啥需要KMP算法就不言而喻了

二.KMP算法大致思想

首先我们用子串去匹配目标串



如图所示

S为目标串，长度为sL，P为子串，长度为pL。

P1部分匹配，在p2部分失配

按照暴力匹配算法，此时我们应该将p串向右移一个字符，继续进行匹配。

但我们不这样做，我们已经知道了子串的信息，而S串和P串的P1部分是相同的，所以S串的P1信息我们是已知的。

假设图中P1部分蓝色方格的字符串是相等的，长度为L，且不可能出现比其更长的相等字符串。

则我们可以将子串相对于目标串向右移动pL-L个字符长度，然后进行匹配，那么能否在这个长度之前进行匹配？很显然不可能，否则两个相等的蓝色部分字符串就不是最长的。

按照这种方式匹配下去，我们可以做到0(n+m)的复杂度实现查找。

三.那么怎么求蓝色部分字符串长度呢。

这里我们用0(m)的时间对P串进行预处理。

定义一个next[i]：表示P串前i个字符的前缀长度和后缀长度相等的最大长度。

假如已知next[i] = k

1.p[i] == p[k]，则next[i+1] = k;

2.p[i] != p[k]，进行迭代，k = next[k]，继续比较p[i]是否等于p[k]

代码如下：

void GetNext(char\* p)

{

int pLen = strlen(p);

nt[0] = -1;

int k = -1,j = 0;

while (j < pLen - 1){

//p[k]表示前缀，p[j]表示后缀

if (k == -1 || p[j] == p[k]){

++k;

++j;

nt[j] = k;

}else{

k = nt[k];

}

}

}

这里有一个关于next的小优化

当我们进行匹配时，假如在p[i]处失配

且有p[i] == p[next[i]]，那么我们将p串相对s串向右移动

pL-next[i]个字符后同样会失配。

所以我们这样处理下

当p[i] == p[next[i]]时，next[i] = next[next[i]]

整个优化后的KMP算法代码如下：

void GetNext(char\* p)

{

int pLen = strlen(p);

nt[0] = -1;

int k = -1,j = 0;

while (j < pLen - 1){

//p[k]表示前缀，p[j]表示后缀

if (k == -1 || p[j] == p[k]){

++k;

++j;

if (p[j] != p[k])

nt[j] = k; //之前只有这一行

else

//因为不能出现p[j] = p[ nt[j ]]

//所以当出现时需要继续递归，k = nt[k] = nt[nt[k]]

nt[j] = nt[k];

}else{

k = nt[k];

}

}

}

int KMP(char\* s, char\* p)

{

int i = 0,j = 0,ans = 0;

int sLen = strlen(s);

while (i < sLen){

if(j == pLen){

ans++;

j = nt[pLen];

}

//如果j = -1,或者当前字符匹配成功(即S[i] == P[j]),都令i++，j++

if (j == -1 || s[i] == p[j]){

i++;

j++;

}else{

//如果j != -1,且当前字符匹配失败(即S[i] != P[j]),则令 i 不变，j = nt[j]

//nt[j]即为j所对应的nt值

j = nt[j];

}

}

if(j == pLen){

ans++;

}

return ans;

}