1. 字符串
2. 匹配算法
   1. 朴素BF算法
   2. KMP算法

【朴素BF算法】

【KMP算法】

1. KMP的算法是避免不必要的回溯
2. 问题是有模式串决定，不是由目标串决定

我的总结：

1. 如何判断跳得对不对？只要看你跳之后你刚刚匹配错误的开始点到是不是我们已经判断过的了。错误不能出现在已经判断过的，否则就要回溯了
2. 如何判断跳得对不对？只要看你跳之后你刚刚匹配错误的开始点到第一个是不是存在错误匹配，一旦存在错误匹配，该跳是不必要的
3. 如何判断跳得对不对？只要看你跳之后你刚刚匹配错误的开始点(在目标串的)到第一个是不是存在错误匹配，一旦存在错误匹配，该跳是不必要的
4. 如何判断跳得对不对？只要看你跳之后的错误匹配点(在目标串的)的改变只能是不动或者前进

我的疑惑：

1.真的只有4种情况吗？，真的，小甲鱼推出的4种情况，已经可以保罗万象了，有不服的再说！

2.模式串问题所在如果是存在相等的元素，那么，在寻找相等元素的过程中，岂不是某一方面大大降低了时间复杂度？

这里你指的时间复杂度，其实就是小甲鱼第二课的k的数组长度，

3.

问题描述(对象是模式串)：

因为X前面都匹配，如果前面第一个跟后面没有相等，那就直接跳到匹配点；如果前面第一个跟后面有相等，不管有多少个这样的相等，先从头一个一个看，看它匹配相等有没有连续到错误匹配点的(中间没有断)，如果断了，就下一个看断不断，如果没断，那就跳到这个相等的匹配点；如果到最后都是断的，那就跳到错误匹配点开始

下一步难题：如何用计算机来表达上面的规则？

1. k数组的引入：k数组的作用是引导下一个该匹配的位置
   1. 如何计算这个k的不匹配，
   2. 不匹配，我的：错误匹配点的k该填多少：不断的连续的个数(不包括错误匹配点本身)+1
   3. 不匹配，小甲鱼的：通过前缀后缀匹配的数量来决定改k值(其实能够判断出断不断了，因为能够与后缀存在匹配的前缀就是存在着这么一个不断的情况了，而匹配的个数我说的不断的连续的个数)而且前缀后缀匹配的方式，同时又解决了我规则上的“从头一个一个看”，因为，
2. 小甲鱼老师的规则：本身就是我规则的体现，如何表达我规则的这个断不断就是看k数组的计算，一旦出现错误匹配，那么我就写数组k，错误匹配点的K值=前缀后缀匹配的个数+1，

注意前后缀匹配的个数计算：划分好前后缀，长度最大开始是总长度-1(自所以从最大长度，体现了我规则的从第一个开始，因为假设我的从第一个开始就出现不断，就意味了跳一跳就行了，也就是意味着匹配个数较大，k值不需要回溯到1而是要回溯前面一个就行了)，从分配方案的第一个开始匹配起，如果第一个分配方案不匹配那么匹配长度剪1，形成第二个分配方案，出现成功匹配的方案的，就看长度，长度就是匹配的个数，k就是长度+1，没出现，那就是0个数，k为3，

k数组的决定因素取决于模式串（体现关键句的本质），所以匹配前先算出k数组，这里的时间复杂度取决于该模式串本身，就是一个O(n)

我的规则的时间复杂度比较：我的规则+k法=小甲鱼算法，

我的规则中的“断不断”的确认如果是遍历循环来确认的话，时间复杂度就是小甲鱼的n倍了

终极解决方案：难点在于k值的填写：

1. NEXT数组：当模式串匹配串T失配的时候，NEXT数组对应的元素知道应该用T串的哪个元素进行下一轮的匹配

本质：利用递归自身的方法，找出我所有能匹配与第一个为前缀的所有片段，根据这些片段的长度，逆向得出next的值

(j值的体现，所以J值是能匹配我就++，不能我就回溯改变前缀匹配片段和它本身的j值再匹配，能匹配我++，不能我就再回溯，因为j从一开始是0的，只要出现连回溯到第一个元素了还不能匹配，那我就为0,也就意味着前面再无匹配现象出现，符合我规则的情况1，直跳错误匹配点，)==》这样就可以遍历了所有与第一个打头的前缀片段，和所有不能与第一个打头的然后我们直接给1的情况，也就遍历了所有的断不断，牛逼的地方在于，通过叠加的方法，体现它的匹配个数，记录它的回溯值的方法来填这个K值。

递归分析：

首先我要递归的方法是：以第一个为打头(前缀是固定的)，不断动态改变前缀在T中的目标串，把所有能走的路走一遍，边走边计算数据，直到结束。

递归结束条件：后缀i一旦超过T本身的长度

递归方法：以前缀为基础进行长度变化，从左到右，i到T[0]去匹配所有这样的片段，并且记录好回溯值，(回溯值是可以提前记的因为已知0开头)

为什么匹配这些片段能记录回溯值？

我们要记录回溯值，那么就得看它前面匹配结果所给出的匹配个数，要看它前面匹配结果所给出的匹配个数，就必须拿他们匹配，要拿他们匹配，就必须能遍历找出匹配方案，刚好，改变后缀固定前缀的方法可以找出所有这样的匹配方案，而找出这些匹配方案的同时也记录了匹配个数(j++的条件是启动引擎的j=0和能匹配成功的情况)所以能计算匹配个数，也就能记录回溯值，一路递归下来的回溯值总能告诉我们当它匹配错误时下一步指导方案，我们就能根据这个回溯值，回溯再匹配，回溯在匹配，这样去求出当前一步的匹配个数，然后进入下一步递归。。。。。。。

j=0;//前缀

i=1;//后缀

while(i<T[0])

{

if(j=0||T[i]==T[j])//

//如果出现T[i]！=T[j]也就是这个片段到头了，新的片段要产生(新片段就是前一个片段的长度+1，体现了我们的前缀是固定的，后缀是相对的，怎么让前缀的旧片段+1？就是把旧片段+1当做T，跳到下一个匹配点，)，

{

i++;

j++;

next[i]=j;//填法：第一个的下标(就是前一步失配导致回溯到它这步的值为第一)或者能够与之匹配的从1到正常匹配的个数+1，刚好j在计算着这个数值

}

}

else{

j=next[j];//匹配错误，前缀要回溯，回溯多少？取决于这时的前缀片段(前缀片段是模式串！决定了回溯多少)，为什么已经有答案？因为前一步已经告诉我们回溯值了

}

【KMP算法的优化】

第4种情况，如果出现连续相等的情况，那就利用第一个回溯的方式，直接回溯到第一个

如

aaaaaaaaaax

aaaaab

typedef char\* string;

int get\_next(string T,int \*next)

{

int i=1;

int j=0;

next[1]=0;

while(i<T[0])

{

if(0==j||T[i]==T[j])

{

i++;

j++;

if(T[i]!=T[j])

{

next[i]=j;

}

else

{

next[i]=next[j];

}

}

else

{

j=next[i];

}

}

}

int Index\_KMP(string S, string T,int pos)

{

int i = pos;

int j = 1;

int next[255];

get\_next(T,next);

while(i<=S[0]&&j<=T[0])

{

if(0==j||S[i]==T[j])

{

i++;

j++;

}

else

{

j=next[j];

}

}

if(j>T[0])

{

return i-T[0];

}

else

{

return 0;

}

}