KMP

算法设计与分析

引言

在字符串初级算法里，最基础的算法是字符串匹配算法，而其中最经典的模型问题 就是判断一个串是否是另一个串的子串。关于字符串算法有一个通用的解题方法，即抓住问题相关的字符串的某种性质，通过这个字符串性质来高效解决问题。很多人可能会认为字符串题目很有难度，不知道从何入手，变化多端，难以琢磨。其实不然，字符串算法的不同模型可能有很多，

引言

但，只要能把特定模型的基本方法、原理弄透，不管题目如何变化多端，都万变不离其中，可以用特定的方法去解决，无非有时需要配上一些额外技巧、科技罢了。所以，对于初学算法的人，切勿一开始就对字符串形成望而生怯的毛病，而应该养成正确思考字符串问题的好习惯——用算法对应的字符串性质来解题.

KMP算法

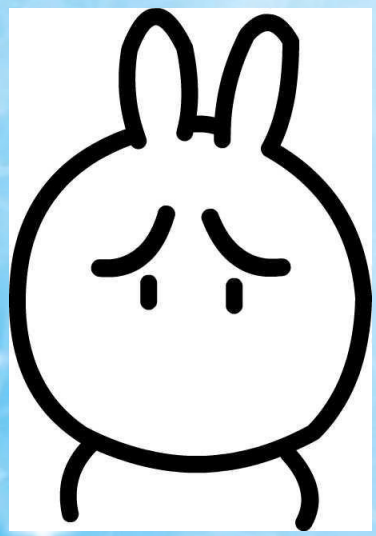
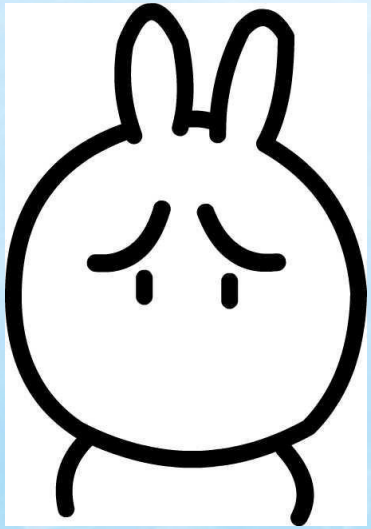
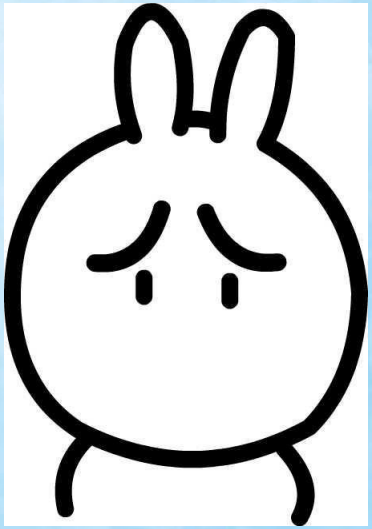
- 在字符串算法里，最简单、基础的就是KMP算法。下面就来看一下
- KMP算法能处理什么样的问题?
- KMP算法用到了什么字符串性质?
- KMP算法实现原理是什么?
- KMP算法的时间复杂度?
- KMP算法的拓展应用?

问题

下面我们先来看这么一个问题：

给两个字符串S1和S2，问S2是否是S1的子串？

数据范围：$0<|S2|<=|S1|<=100000$

输入和输出

Input

第一行输入一个字符串S1，第二行输入一个字符串S2，保证0<|S2|<=|S1|<=100000

Output

如果S2是S1的子串，输出“YES”.

否则，输出“NO”.

样例

- Sample Input
- AAABAAABAAABAAAD
- AAABAAAD

- ABAAB
- ABB

- Sample Output
- YES
- NO

一个字符串的子串指的是字符串某一段连续的部分（比如第一个例子），可以是其本身。
而不连续的部分，一般称作为子序列！（比如第二个例子，ABB是ABAAB的子序列而不是子串）

KMP算法用到了什么字符串性质

- 下面介绍下KMP用到的字符串性质
- 是否还记得刚才说的?
- 每个字符串算法都对应它的字符串性质!
- 我们定义这么一个字符串性质，叫前缀后缀最大值!
- 光从定义来看，似乎是和字符串的前缀、后缀有关系，同时告诉你，最大值指的是长度最大，什么长度最大？下面具体来看下这个性质。

前缀后缀最大值

- 一个长度为N的字符串S，它有N+1个前缀（包括空前缀），它有N+1个后缀（包括空后缀）
- 比如ABC，有4个前缀，空，A，AB，ABC
  有4个后缀，空，C，BC，ABC
- 比如AAA，有4个前缀，空，A，AA，AAA
  有4个后缀，空，A，AA，AAA

前缀后缀最大值

举一个容易看出性质的例子，S=ABABABA

前缀	后缀	相等
空	空	yes
A	A	yes
AB	BA	no
ABA	ABA	yes
ABAB	BABA	no
ABABA	ABABA	yes
ABABAB	BABABA	no
ABABABA	ABABABA	yes

容易发现，S有5个前缀与后缀相等，如果我们不算自身，即前缀ABABABA不算，后缀ABABABA不算，那么，在所有相等的<前缀，后缀>里，长度最大的就是ABABA，则前缀后缀最大值就是5！

前缀后缀最大值

一个字符串S，长度为N.

找出它的N+1个前缀（包括空前缀）

找出它的N+1个后缀（包括空后缀）

按照长度划分，得到N+1对序偶<前缀，后缀>

删除前缀、后缀等于S的<前缀，后缀>，得到N对<前缀，后缀>。

在这N对中，找到一对满足：

1. 前缀 = 后缀　　2. 前缀后缀的长度最大

该长度就是S的前缀后缀最大值！

前缀后缀最大值

下面我们拿暴力匹配算法中的模板串

S=AAABAAAD 来具体阐述！

我们定义一个数组：int next[N];

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

接下来，我们分析字符串S的每一个前缀的next值，即每一个前缀的前缀后缀最大值。

然后，我们再介绍如果使用这个next数组！

重要的话要多讲几遍！

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

请务必牢记这个定义！KMP核心部分就是这个next数组。对next数组的理解透彻与否决定你能否快速、准确地解决KMP相关的题目！注意，是S[0...i-1] 不是S[0...i]！

同时务必正确理解前缀后缀最大值的含义！

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1								

S=AAABAAAD，第0个前缀：空前缀

空前缀的next值我们直接定义为 -1

即next[0]=-1. 记得之前的“删除前缀、后缀等于S的<前缀，后缀>”的操作吗?

删除后找不到<前缀，后缀>，next值为-1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0							

S=AAABAAAD，第1个前缀：A

next[1]=0. 表示S[0...0]的前缀后缀最大值是0

一个前缀：空

一个后缀：空

显然，next[1]=|空|=0

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1						

S=AAABAAAD，第2个前缀：AA

next[2]=1. 表示S[0...1]的前缀后缀最大值是1

两个前缀：空，A

两个后缀：空，A

显然，next[2]=|A|=1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2					

S=AAABAAAD，第3个前缀：AAA

next[3]=2. 表示S[0...2]的前缀后缀最大值是2

三个前缀：空，A，AA

三个后缀：空，A，AA

显然，next[3]=|AA|=2

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0				

S=AAABAAAD，第4个前缀：AAAB

next[4]=0. 表示S[0...3]的前缀后缀最大值是0

四个前缀：空，A，AA，AAA

四个后缀：空，B，AB，AAB

显然，next[4]=|空|=0

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1			

S=AAABAAAD，第5个前缀：AAABA

next[5]=1. 表示S[0...4]的前缀后缀最大值是1

五个前缀：空，A，AA，AAA，AAAB

五个后缀：空，A，BA，ABA，AABA

显然，next[5]=|A|=1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2		

S=AAABAAAD，第6个前缀：AAABAA

next[6]=2. 表示S[0...5]的前缀后缀最大值是2

空, A, AA, AAA, AAAB, AAABA

空, A, AA, BAA, ABAA, AABAA

显然，next[6]=|AA|=2

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	

S=AAABAAAD，第7个前缀：AAABAAA

next[7]=3. 表示S[0...6]的前缀后缀最大值是3

空, A, AA, AAA, AAAB, AAABA, AAABAA

空, A, AA, AAA, BAAA, ABAAA, AABAAA

显然，next[7]=|AAA|=3

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	0

S=AAABAAAD，第8个前缀：AAABAAAD

next[8]=0. 表示S[0...7]的前缀后缀最大值是0

空,A,AA, AAA, AAAB, AAABA, AAABAA, AAABAAA

空,D,AD,AAD, AAAD, BAAAD, ABAAAD, AABAAAD

显然，next[8]=|空|=0

得到next数组

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	0

我们得到S串的next数组

再次回忆，next[i]表示S[0...i-1]这个前缀的前缀后缀最大值。

那么，有什么用呢？！

它与子串匹配有什么关系呢?

回顾与分析

给两个字符串S1和S2，问S2是否是S1的子串?

S1=AAABAAABAAABAAAD

S2=AAABAAAD

在这个问题中，我们得到文本串S1和模板串S2.

现在想判断S2(单词)是否是S1(文章)的子串。

我们先对模板串S2，构建next数组，得到S2每一个前缀的前缀后缀的最大值。

接下来，开始KMP匹配过程!

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

设两个变量 int t1，t2；t1表示当前扫到S1串的位置，t2表示当前扫到S2串的位置。起初，t1=t2=0；

S1[t1]=S2[t2]=A，t1++，t2++；//t1=1，t2=1

S1[t1]=S2[t2]=A，t1++，t2++；//t1=2，t2=2

S1[t1]=S2[t2]=A，t1++，t2++；//t1=3，t2=3

S1[t1]=S2[t2]=B，……

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

此时t1=t2=3;

S1[t1]=S2[t2]=B，t1++，t2++；//t1=t2=4

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=5

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=6

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=7

此时发现，S1[t1] != S2[t2]

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

此时，t1=7, t2=7, S1[t1]=B, S2[t2]=D 出现不相等！

在之前的暴力匹配算法中，这说明了什么？

说明了从 t1=0 这个起始位置开始，往后长度为|S2|的子串不与S2匹配，那么，在暴力算法中，接下来应该枚举下一个起始位置 t1=1, 再往下判断, 是吧？但是！在KMP中有所不同！……

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

在KMP中，S1[7] !=S2[7]，此时出现了失配!

怎么办呢？我们查询7号位置的next值，即next[7]=3.

然后, 我们直接令t2=next[t2]=next[7] (next[7]=3);相当于把S2右移了4格，下面，我们先来看下这个神奇的变化！再分析下这么做的理由和优势。

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2`	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

S1未动，S2整体往右移动了 |S2| - next[7] 个格子

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

此时，我们只需从 t1=7，t2=3 这个匹配对开始往下继续匹配即可。而S1[4...6]与S[0...2]相当于已经匹配成功了，不需要再匹配。

想想看，为什么可以这么做?

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

我们在匹配时，是不是已经得到S1[0...6]=S2[0...6]了？

然后，通过next数组，next[7]=3，是不是我们就知道S2[0...2]=S2[4...6]这个性质？于是我们就可以推得:S2[0...2]=S1[4...6],又由于next[7]表示S2[0..6]这个前缀的前缀后缀最大值！所以，这样挪动是正确的！

正确可行的匹配一定会延续到S1的位置7，这样挪动和将i退回到1是等价的，退回到1，新的匹配也是S1的后缀和S2的前缀匹配。

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

换句话说，我们在S1[7]与S2[7]处出现了失配，此时t1不需要返回，只需改变t2。在7处失配，则只需查询S2[0...6]处的前缀后缀最大值，表示某一段前缀等于后缀，而又是长度最大的！那么移动后，失配点的前段一定还是匹配的，而只需在从失配点往下匹配即可，若失配点还是失配，再继续改变t2

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

失配点是 i=7,j=3,失配点的前一段S1[4...6]=S2[0..2]仍然匹配，现在只需从失配点，t1=7，t2=3，继续往下匹配即可...

当next[t2]==-1,表示S2的首个字符和S1的第i个字符都不匹配，那么i就加1

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

好！我们继续往下匹配，发现t1=11, t2=7的时候，又出现了失配！与刚才同样的步骤，令t2=next[t2]=next[7].

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2`					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

S1未动，S2整体往右移动了 |S2| - next[7] 个格子

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

从 i=11，j=3这个匹配对，继续往下匹配...

前一段S1[8...10]与S2[0...2]已经匹配成功！

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

继续往下匹配，都能匹配成功！

发现S2[0...7]与S1[8...15]完全匹配成功！

S2是S1的子串。

代码：构建next数组

```
void get_next(int *next,char *s2,int lens){
  //用于构建s2的next数组，kmp的前奏
   int t1=0,t2;
   next[0]=t2=-1;
   while(t1<lens){
      if(t2==-1||s2[t1]==s2[t2]){
         next[t1+1]=t2+1;
         t1++;
         t2++;
      }
      else t2=next[t2];
   }
}
```

代码：KMP匹配

```
bool kmp(int *next,char *s1,int lens1,char *s2,int lens2){ //用于判断S2是否是S1的子串
    int t1=0,t2=0;
    while(t1<lens1&&t2<lens2){
        if(t2==-1||s1[t1]==s2[t2]){
            t1++;
            t2++;
        }
        else t2=next[t2];
    }
    if(t2==lens2) return true;//S2是S1子串
    else return false;//S2不是S1子串
}
```

代码：KMP匹配2

```
int kmp(int *next,char *s1,int lens1,char *s2,int lens2){
    int t1=0,t2=0;
    int times=0;
    while(t1<lens1){
        if(t2==-1||s1[t1]==s2[t2]){
            t1++;
            t2++;
        }
        else t2=next[t2];
        if(t2==lens2){
            times+=1;
            t2=next[t2];
        }
    } //求S2在S1中出现了多少次
    return times;
}
```

KMP的用途

1. 判断一个串是否是另一个串的子串。
2. 判断一个串在另一个串出现了多少次。
3. 求一个字符串的最小循环节.
4. 进行各式各样的字符串匹配，模糊匹配等

（kmp匹配只是最经典的匹配一种，很多时候是需要在失配函数上做文章，完成另类的匹配）

5. …… 等等