今天是数据结构与算法的串。

一、基本概念

串(string)是由零个或多个字符组成的有限序列，又名叫字符串。 一般记为：$S =' a_1a_2a_3'$，其中，S是串名，单引号括起来的字符序列是串的值；a_n可以是字母、数字或者其他字符；串中字符的个数n称为串的长度。

- 空串：n = 0 时的串称为空串。
- 空格串：是只包含空格的串。注意它与空串的区别，空格串是有内容有长度的，而且可以不止一个空格。
- 子串与主串：
  - **子串**：串中任意个数的**连续字符组成的子序列。**
  - **主串**：包含子串的串称为主串，子串在主串中的位置就是子串的第一个字符在主串中的序号。

串的逻辑结构和线性表极为相似，区别仅在于串的数据对象限定为字符集。

在基本操作上,串和线性表有很大差别。线性表的基本操作主要以单个元素作为操作对象，如查找、插入或删除某个元素等；而串的基本操作**通常以子串作为操作对象**，如查找、插入或删除一个子串等。

二、串的存储结构

(1) 定长顺序存储表示

类似于线性表的存储结构，用一组地址连续的存储单元存储串值的字符序列。在串的定长顺序存储结构中，为每个串变量分配一个固定长度的存储区，即**定长数组**。

```
#define MAXLEN 255    //预定义最大串长为255
typedef struct{
    char ch[MAXLEN];    //每个分量存储一个字符
    int length;    //串的实际长度
}SString;
```

串的实际长度只能小于等于`MAXLEN`，超过预定义长度的串值会被舍去，称为截断。

串长有两种表示方法:

- 一是如上述定义描述的那样，用一个额外的变量`len`来存放串的长度；

- 二是在串值后面加一一个不计入串长的结束标记字符“\0”，此时的串长为隐含值。

在一些串的操作(如插入、联接等)中，若串值序列的长度超过上界MAXLEN,约定用“截断”法处理，要克服这种弊端，只能不限定串长的最大长度，即采用动态分配的方式。

（2）堆分配存储表示

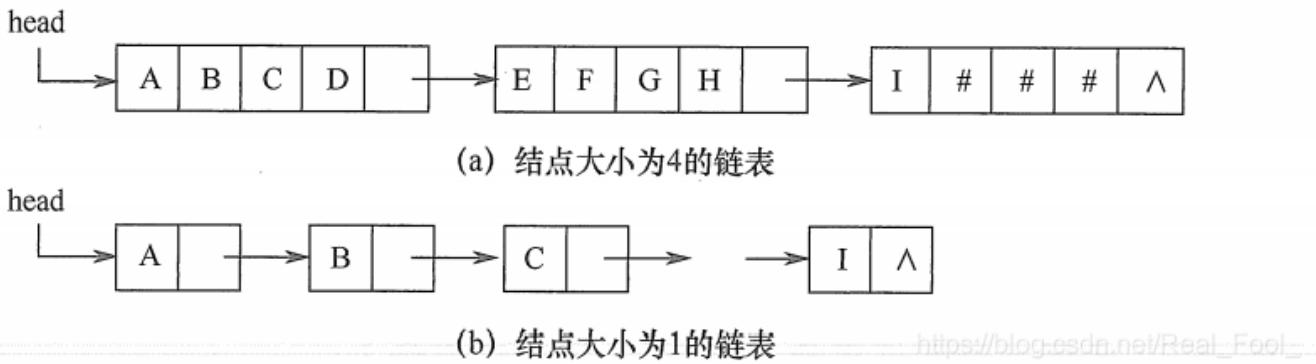
堆分配存储表示仍然以一组地址连续的存储单元存放串值的字符序列，但它们的存储空间是在程序执行过程中动态分配得到的。在C语言中，存在一一个称之为“堆”的自由存储区，并用`malloc()`和`free()`函数来完成动则返回一个指向起始地址的指针，作为串的基地址，这个串由ch指针来指示;若分配失败，则返回NULL。已分配的空间可用`free()`释放掉。

```
typedef struct{
    char *ch;     //按串长分配存储区，ch指向串的基地址
    int length;     //串的长度
}HString;
```

上述两种存储表示通常为高级程序设计语言所采用。

（3）块链存储表示

类似于线性表的链式存储结构，也可采用链表方式存储串值。由于串的特殊性(每个元素只有一个字符)，在具体实现时，每个结点既可以存放一个字符， 也可以存放多个字符。每个结点称为块，整个链表称为块链结构。图(a)是结点大小为4 (即每个结点存放4个字符)的链表,最后一个结点占不满时通常用“#”补上;图(b)是结点大小为1的链表。



(a) 结点大小为4的链表

(b) 结点大小为1的链表

块链存储表示仅做简单介绍。

三、字符串的模式匹配

(1) 简单的模式匹配

直接暴力枚举

```
int Index(SString S, SString T){
    int i = 1, j = 1;
    while(i <= S.length && j <= T.length){
        if(S.ch[i] == T.ch[j]){
            ++i; ++j;    //继续比较后继字符
        }else{
            //指针后退重新开始匹配
            i = i-j+2;
            j = 1;
        }
    }
    if(j > T.length){
        return i - T.length;
    }else{
        return 0;
    }
}
```

(2) KMP算法

在上面的简单匹配中，每趟匹配失败都是模式后移一位再从头开始比较。

而某趟已匹配相等的字符序列是模式的某个前缀，这种频繁的重复比较相当于模式串在不断地进行自我比较，这就是其低效率的根源。

因此，可以从分析模式本身的结构着手，如果已匹配相等的前缀序列中有某个后缀正好是模式的前缀，那么就可以将模式向后滑动到与这些相等字符对齐的位置，主串i指针无须回溯，并继续从该位置开始进行比较。而模式向后滑动位数的计算仅与模式本身的结构有关，与主串无关。KMP算法的特点就是：仅仅后移模式串，比较指针不回溯。

首先是要了解子串的结构具体包括：前缀、后缀和部分匹配值。

- 前缀指除最后一个字符以外,字符串的所有头部子串;
- 后缀指除第一个字符外,字符串的所有尾部子串;
- 部分匹配值则为字符串的前缀和后缀的最大公共前后缀长度。

下面以' a b a b a '为例：

- ' a ' 的前缀和后缀都为空集最大公共前后缀长度长度为0。
- ' a b ' 的前缀为{ a }，后缀为{ b }， { a } ∩ { b } = N U L L，最大公共前后缀长度长度为0。
- ' a b a ' 的前缀为{ a , a b }，后缀为{ a , b a }, { a , a b } ∩ { a , b a } = { a }，最大公共前后缀长度长度为1

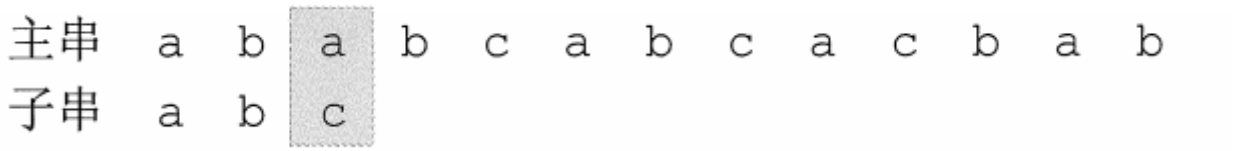
- ' a b a b ' 的前缀∩后缀，{ a , a b , a b a } ∩ { b , a b , b a b } = { a b } ，最大公共前后缀长度长度为2。
- ' a b a b a ' 的前缀∩后缀， { a , a b , a b a , a b a b } ∩ { a , b a , a b a , b a b a } = { a , a b a } ，公共元素有两个,最大公共前后缀长度长度为3。

故字符串' a b a b a '的最大公共前后缀的长度为00123。具体如何利用最大公共前后缀，我们来看主串为 ' a b a c a b c a c b a b ' ，子串为 ' a b c a c '的模式匹配上 。

首先计算字串的最大公共前后缀 (Partial match,PM)：

编号	1	2	3	4	5
S	a	b	c	a	C
PM	0	0	0	1	0

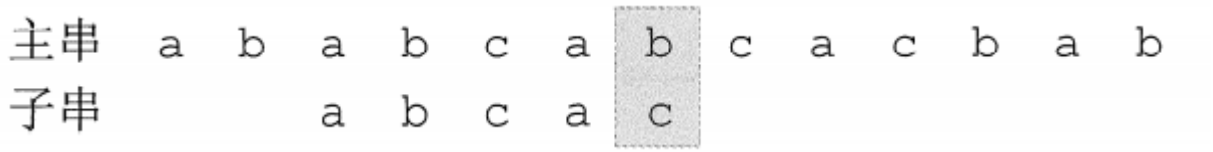
下面利用PM进行匹配：



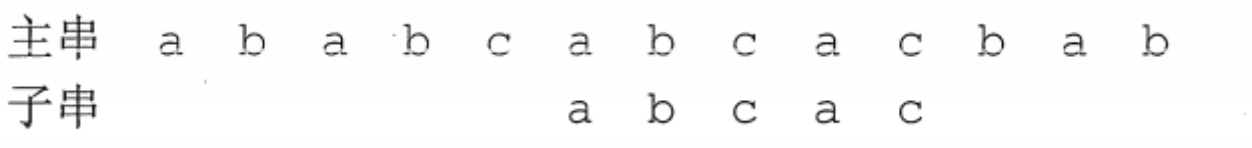
当我们发现a和c不匹配时，前面的ab是匹配的。此时我们发现前一位b的PM值为0，我们可以计算出字串需要向后移动的位数：

$$移动位数 = 已匹配的字符数 - 对应最大公共前后缀$$

因为2 - 0 = 2，所以将子串向后移动2位：



发现b和c不匹配是，前一位a的PM值为1，计算需要位移 4 - 1 = 3 位



子串全部比较完成，匹配成功。整个匹配过程中，**主串始终没有回退**，故KMP算法可以在`O ( n + m )`的时间数量级上完成串的模式匹配操作，大大提高了匹配效率。

对于移动位数，我们每次都需要利用PM的值重新计算，但其实对于每一位每次移动的位数都是固定的，因此我们可以直接利用一个next数组记录需要移动的位置。

编号	1	2	3	4	5
S	a	b	c	a	c
next	0	1	1	1	2

最终得到子串指针变化公式`j = next[ j ]`。

`next[ j ]`的含义是：在子串的第j个字符与主串发生失配时,则跳到子串的next[j]位置重新与主串当前位置进行比较。通过分析，可以知道，除第一个字符外，模式串中其余的字符对应的next数组的值等于其最大公共前后缀长度加上1。

$$next[j] = 最大公共前后缀长度 + 1$$

公式可以推到为：

$$next[j]=\begin{cases}0, & j=1\\ max\{k \mid 1<k<j且'p_1\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\}, & 当此集合不空时\\ 1, & 其他情况\end{cases}$$

因此代码可以写成（我的代码是匹配和被匹配的字符串都是**从0下标开始**，要是都是从1下标开始，再在每个next的值上加1就好了）。

```
void GetNext(string s1, vector<int> &next){
    next[0] = -1;
    int j = -1; // i是指最大后缀的最后一个字母下标，j是指最大前缀的最后一个字母下标
    for(int i = 1; i < s1.size(); i++){
        //每次和下一个字母匹配（j + 1）
        while(j != -1 && s1[i] != s1[j + 1]) j = next[j];
        if(s1[i] == s1[j + 1]) j++;
        next[i] = j;
    }
}
```

与next数组的求解相比，KMP算法就简单许多，和简单模式匹配算法很相似：

```cpp
int main(){
    int m, n;
    string s1, s2;
    cin >> n >> s1 >> m >> s2;
    vector<int> next(n + 1);
    GetNext(s1, next);
    int j = -1;
    for(int i = 0; i < m; i++){
        while(j != -1 && s2[i] != s1[j + 1]) j = next[j];
        if(s2[i] == s1[j + 1]) j++;
        if(j == n - 1) {
            cout << i - n + 1 << " "; // 输出每次匹配到的下标
            j = next[j]; // 返回重新匹配
        }
    }
    return 0;
}
```