

# 数据结构第十二周笔记——综合习题选讲 (慕课浙大版本--XiaoYu)

## 习题选讲 - Insert or Merge

### 习题-IOM.1 插入排序的判断

#### 题意理解

如何区分简单插入和非递归的归并排序

1. 插入排序：前面有序，后面没有变化
2. 归并排序：分段有序

Sample Input:

```
10
3 1 2 8 7 5 9 4 6 0
1 2 3 7 8 5 9 4 6 0

10
3 1 2 8 7 5 9 4 0 6
1 3 2 8 5 7 4 9 0 6
```

#### 捏软柿子算法

ps：在插入和归并两种算法里，哪种算法比较容易判断？**插入排序**

判断是否插入排序

1. 从左向右扫描，直到发现顺序不对，跳出循环
2. 从跳出地点继续向右扫描，与原始序列比对，发现不同则判断为"非"(如果是插入排序的话，所有的元素都是跟原始序列一模一样的)

如果在**对比**的过程中发现不同，则跳出循环

3. 循环自然结束，则判断为"是"，返回跳出地点

1. 解析：(我们可以返回一个布尔值，例如非为0、是为1，但如果我返回的是"是"的话，插入排序要往下进行一步，简单返回一个1在我进行插入排序下一步的时候，还得从左向右扫描去找那个要执行下一步的那个点。所以我们返回"是"的同时返回一个跳出的地点

如果是插入排序，则从**跳出地点**开始进行一趟插入

### 习题-IOM.2 归并段的判断

#### 判断归并段的长度

错误的想法：

1. 从头开始连续有序的子列长度？  

2	1	8	9	6	5	3	4
1	2	8	9	5	6	3	4
2. 所有连续有序子列的最短长度？  

4	2	1	3	13	14	12	11	8	9	7	6	10	5
1	2	3	4	11	12	13	14	6	7	8	9	5	10

1. 这个其实是四个一段的，但前8个刚好都是有序的

2. 保险正确的判断方法：从原始序列出发，真的在做归并排序，每归并一趟就把归并的中间结果跟这个结果的序列做一个比对。什么时候每一个数都对上了就再把当前的归并多执行一次然后输出结果

3. `for (l=2;l<=N;l*=2)`  
 //在保证l是4的情况下，要检查看能不能是8，我们要重复前面的步骤看两段之间的衔接点是不是有序

4 2 1 3 13 14 12 11 8 9 7 6 10 5  
 1 2  $\leq$  3 4 11 12  $\leq$  13 14 6 7  $\leq$  8 9 5 10

红色位置没有有序跳出循环(此时l为4，我们直接以4为归并段继续执行下一趟的归并就可以了)

4 2 1 3 13 14 12 11 8 9 7 6 10 5  
 1 2 3 4  $\leq$  11 12 13 14 6 7 8 9  $\leq$  5 10

## 其他数据测试

最小N(应该是多大?)

ps: 边界测试是每道题里面测试非常重要的一个组成部分

N会是1吗? N等于1会出现什么情形?

N等于1就意味着整个序列里面只有一个数字，在排序前它是一个数字，在排序之后他仍然是同一个数字，在这种情况下我不管是使用插入排序还是归并排序得到的都会是同样的结果，这样解就不是唯一的。我们题目输出的要求是插入排序或者归并排序的其中一个，所以N=1是绝对不可以的

保证可以区分两种算法的最小N应该是：4(区分插入排序与归并排序最小要求)

1. 插入排序第一步，什么都没变
2. 归并排序第一步，什么都变了

尾部子列无变化，但前面变了(归并)

最大N

## 习题选讲 - Sort with Swap(0,\*)


### 习题-SWS.1 环的分类

#### 题意理解

1. 给定N个数字的排列，如何仅利用与0交换达到排序目的?

0在里面扮演了空位的问题

□ N个数字的排列由若干个独立的环组成



A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	a	b	c	d	g	f	h	e
table	3	5	2	1	7	0	4	6

Temp = f



## ■ 环分3种

1. 只有1个元素：不需要交换
2. 环里 $n_0$ 个元素，包括0：需要 $n_0-1$ 次交换
3. 第 $i$ 个环里有 $n_i$ 个元素，不包括0：先把0换到环里，再进行 $(n_i+1)-1$ 次交换 —— 一共是 $n_i+1$ 次交换

■ 若 $N$ 个元素的序列中包含 $S$ 个单元环、 $K$ 个多元环，则交换次数为：

$$n_0 - 1 + \sum_{i=1}^{K-1} (n_i + 1)$$

$$= \sum_{i=0}^{K-1} n_i + K - 2 = N - S + K - 2$$

## 习题-SWS.2 算法示例

下标	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A[]	3	5	7	2	6	4	9	0	8	1

T[]	7	9	3	0	5	1	4	2	8	6
-----	---	---	---	---	---	---	---	---	---	---

$T[A[i]] = i$ ; 元素 $i$ 在 $A[T[i]]$ 中存放

下标	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A[]	0	1	2	3	4	5	6	7	8	9

T[]	7	9	3	0	5	1	4	2	8	6
-----	---	---	---	---	---	---	---	---	---	---

对于不包含0的swap操作次数为 $n+1$ ，包含0则是 $n-1$ 次

$$3 + 6$$

$$N - S + K - 2 = 10 - 1 + 2 - 2 = 9$$

## 习题选讲 - Hashing - Hard Version

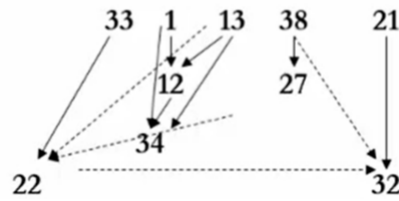
### 习题-HHV 算法思路概述

这是哈希问题的逆问题

#### 题意理解

1. 已知 $H(x) = x \% N$ 以及用线性探测解决冲突问题，模大小取决于目的有多少个下标
2. 先给出散列映射的结果，反求输入顺序
  1. 当元素 $x$ 被映射到 $H(x)$ 位置，发现这个位置已经有 $y$ 了，则 $y$ 一定是在 $x$ 之前被输入的

下标	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
H[ ]	33	1	13	12	34	38	27	22	32		21



限制：为了保证解是唯一的，当有几个元素都有可能是同时被插入的时候，我们是从小到大去插入的

因为12模11，余数为1，所以跟1冲突，放在12下面。后面都是类型的操作

依次输入顺序为 1 13 12 21 33 34 38 27 22 32

## 串的模式匹配(KMP算法)

### KMP-1. 问题及简单解决方案

#### 什么是串

1. 线性存储的一组数据(默认是字符)
2. 特殊操作集

1. 求串的长度
2. 比较两串是否相等
3. 两串相接
4. 求子串
5. 插入子串
6. 匹配子串(有难度)
7. 删除子串

#### 什么是串的模式匹配

目标：给定一段文本，从中找出某个指定的关键字

例如从一本Thomas Love Peacock写于十九世纪的小说《Headlong Hall》中找到那个最长的单词：

osseocarnisanguineoviscericartilaginonervomedullary

或者从古希腊喜剧《Assemblywomen》中找到一道菜的名字：

Lopadotemachoselachogaleokraniroleipsanodrimhypotrimmatosilphioparaomelitokatakechymenokichlepikossyphophattoperisteralektryonoptekhephallioikigklopeleiolagoiosiraobaphetraganopterygon

当我们文本是很长的时候，而指定的关键字也是一个很长的字符串的时候，模式匹配就不再是一件简单的事情了

给定一段文本：  $string = s_0s_1 \dots s_{n-1}$

给定一个模式：  $pattern = p_0p_1 \dots p_{m-1}$

求  $pattern$  在  $string$  中出现的位置

```
Position PatterMatch(char *string,char *pattern)//position指位置
//模式匹配就是给定一段文本(string)，给定一个模式(*pattern)，我们要通过PatterMatch函数
来返回这个pattern里string第一次出现的位置
```

## 简单实现

方法1: C语言的库函数strstr

```
接口: char *strstr(char *string,char *pattern)
//返回的是char *这个类型的变量(指向某个字符的指针)，变量里面存的是pattern这个字符串第一个
字母在string出现的时候那个字母所在的位置
//一个小Demo
#include <stdio.h>
#include <string.h>//库要记得包含进来

typedef char* Position;//给char*重新起个名字，让不懂的人也可以知道返回的是一个位置

int main()
{
    char string[] = "This is a simple example.";
    char pattern[] = "simple";
    Position p = strstr(string,pattern);
    if( p == NotFound ) printf("Not Found.\n");//能不能找到进行一个判断
    else printf("%s\n",p);
    return 0;
}
//输出: simple example.
//如果输入的找不到，就会输出一个空指针(#define NotFound NULL)
```

strstr的复杂度怎么样? 要想知道这个问题我们就得了解一下strstr是怎么运行的

```
string = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
pattern = "aab"
```

如上图，是两个指针指向两个变量的内容开头进行比对，第一个对上了对下一个，直到全部对上或者中途失败的时候将pattern的a与string下一个字符继续比对，一直循环下去，直到比对完都没成功或者中途成功了就退出循环

若给定文本长度为  $n$ ，模式长度为  $m$ ，则库函数 strstr 的最坏时间复杂度是： $T = O(n*m)$

当我们的pattern比较小的时候，我们这个strstr库函数还是很好用的，当两者都不小的时候就得慎重了

## 简单改进

方法2: 从末尾开始比

```
String [blue bar with cursor] ?
pattern [red bar]
```

时间复杂度:  $T = O(n)$ //仅仅是根据上方的例子进行的改动，如果pattern = "aab"换成"baa"一样要比Q

所以这个改进是没啥作用的

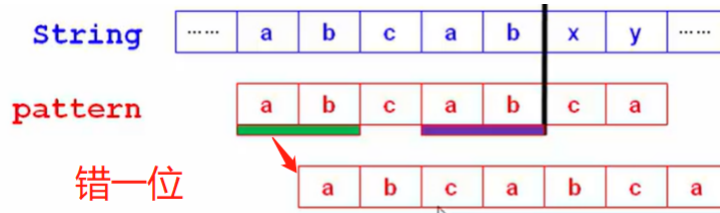
## KMP-2. KMP 算法思路

### 大师改进

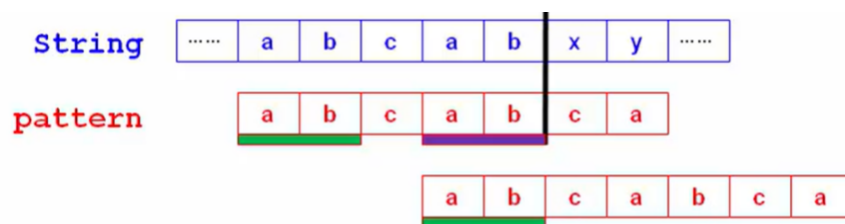
方法3: KMP(Knuth、Morris、Pratt)算法

$T = O(n+m)$

简单的往前错一位的比较是完全没有必要的没有意义的，如下图



KMP算法的想法:



指针指向x不会回退(回溯)了，直接继续从x开始，继续往前比较

$$\text{match}(j) = \begin{cases} \text{满足 } p_0 \cdots p_i = p_{j-i} \cdots p_j \text{ 的最大 } i (< j) \\ -1 & \text{如果这样的 } i \text{ 不存在} \end{cases}$$

match的具体例子

pattern	a	b	c	a	b	c	a	c	a	b
j	0	1	2	3	4	5	6	7	8	9
match	-1	-1	-1	0	1	2	3	-1	0	1

下标从0到9

第0个字符对应的是一个长度为1的子串，所以他不可能产生匹配，match就永远是-1

从0到1: a跟b是配不上的，match也为-1

0-2: a和c配不上，ab和bc也配不上，所以match还是为-1

0-3: ab和ca是配不上的，abc跟bca也配不上，a对应的j为0，所以match也为0

//此时限制条件是最大i是小于j的，如果i=j的话那就相当于自己等于自己就没有意义了( $p_0 \dots p_j = p_0 \dots p_j$ )

//所以我们考虑他的真子串

0-4: a跟b配不上，abc跟cab配不上，ab跟ab能配上，match值为1...

对于 pattern = abcabcacab，最后 3 个字符的 match 值是多少? -1, 0, 1

在早期的教科书上被叫做failure(失败的意思)

match值的含义:

例子: 从0到6的子串，首跟尾能配上的小串，从0开始他的尾部下标为3，abca跟abca能配上。这就是match的含义

此代码块内容来自百度百科：

**KMP**算法是一种改进的字符串匹配算法，由**D.E.Knuth**，**J.H.Morris**和**V.R.Pratt**提出的，因此人们称它为克努特-莫里斯-普拉特操作（简称**KMP**算法）。**KMP**算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。具体实现就是通过一个**next()**函数实现，函数本身包含了模式串的局部匹配信息。**KMP**算法的时间复杂度 $O(m+n)$

**KMP**算法是三位学者在 **Brute-Force**算法的基础上同时提出的模式匹配的改进算法。**Brute-Force**算法在模式串中有多个字符和主串中的若干个连续字符比较都相等,但最后一个字符比较不相等时,主串的比较位置需要回退。**KMP**算法在上述情况下,主串位置不需要回退,从而可以大大提高效率

## 模式匹配类型

### (1)精确匹配

如果在目标**T**中至少一处存在模式**P**，则称匹配成功，否则即使目标与模式只有一个字符不同也不能称为匹配成功，即匹配失败。给定一个字符或符号组成的字符串目标对象**T**和一个字符串模式**P**，模式匹配的目的是在目标**T**中搜索与模式**P**完全相同的子串，返回**T**和**P**匹配的第一个字符串的首字母位置。

### (2)近似匹配

如果模式**P**与目标**T**(或其子串)存在某种程度的相似，则认为匹配成功。常用的衡量字符串相似度的方法是根据一个串转换成另一个串所需的基本操作数目来确定。基本操作由字符串的插入、删除和替换来组成

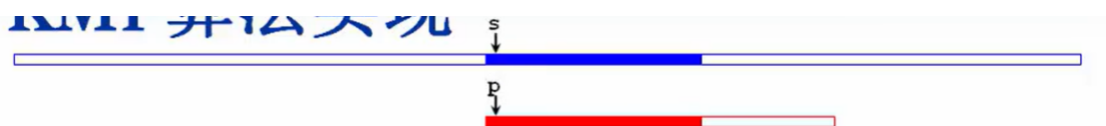
## KMP-3. KMP 算法实现

```
#include<stdio.h>
#include<string.h>

typedef int Position;//Position对应的是一个整型变量，指的是数组的下标
#define NotFound -1//NotFound就应该定义成一个不可能是数组下标的东西

int main()
{
    char string[] = "This is a simple example.";//默认指字符串，但这个串可以是任何类型的
    char pattern[] = "simple";
    Position p = KMP(string,pattern);//返回的是一个字符指针的话就只能处理字符串，如果返回的是数组下标的话那可以处理任何字符的串
    if (p == NotFound ) printf("Not Found.\n");
    else printf("%s\n",string+p);//因为这里返回的是整数，这个整数就没办法被当作字符串的头指针了。如果我们要打印整个字符串的话，我们这里就只能写成string+p这样的形式
    return 0;
}
```

## KMP算法实现



一直走到指针不匹配





```

Position KMP(char *string,char *pattern )
{
    int n = strlen(string);//strlen得到string的长度，下方也是一样 复杂度：O(n)
    int m = strlen(pattern);//复杂度：O(m)
    int s,p, *match;//声明两个指针
    if(n < m) return NotFound;//找的n不可能比m短
    match = (int *)malloc(sizeof(int) *m);
    BuildMatch(pattern,match);//Tm = O(?)
    s = p =0;
    while( s<n &&p<m){ //当这两个指针一起往前飞，任何一个指针先指到自己指的串的末尾的时候结束，复杂度O(n)
        if(string[s] == pattern[p]){ s++;p++; }//p有时候加有时候回退，但s永远加
        else if (p>0) p = match[p-1]+1;//为了防止得到错误，这里加上条件p>0
        //如果p = 0的话，意味着pattern从第一个字符就不匹配，这个时候p不动，s向前走一格
        else s++;//当string[s] == pattern[p]不匹配，我们s++，继续下一轮匹配
    }
    //在我们跳出while循环的时候，p指针已经碰到pattern的末尾(p==m)，那就是完全的匹配上了
    //反之p还没有到结尾，而string已经到p的结尾了，就意味着我们找不到这个模式
    return (p == m) ? (s-m) : NotFound;
}

```



KMP的整体时间复杂度：T = O(n+m+Tm)

## KMP-4. BuildMatch 的实现原理



$$1 + 2 + \dots + \frac{j+1}{2}$$

for ( j=0; j<m; j++ )



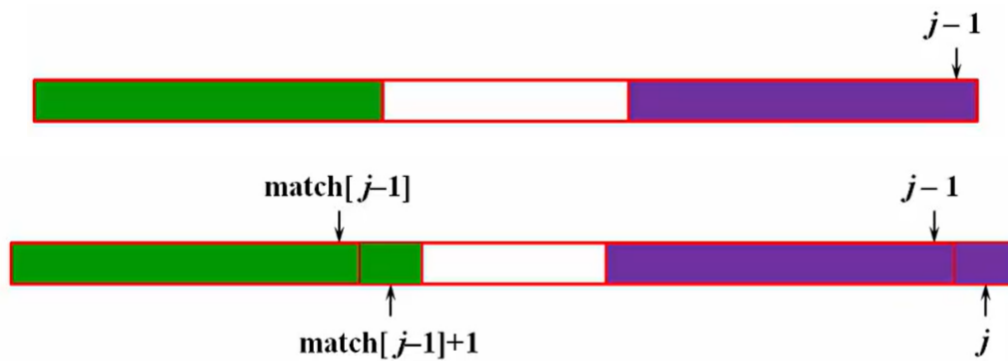
$$1 + 2 + \dots + \frac{j+1}{2} + \dots + j = O(j^2)$$

如果采用这种方法实现的话，时间复杂度将会达到Tm = O(m³)

新想法：如果我们要算j的match值的话先考虑他跟j-1的match值有什么关系

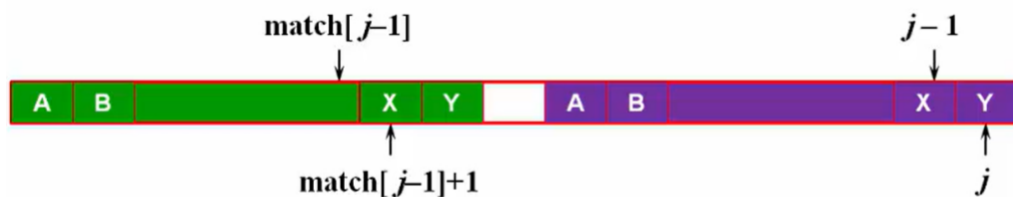
假如我们这是从0到j-1的字段





`match[j] >= match[j-1] + 1` (是否正确?)

如果 `match[j-1]+1` 这个位置上的字符与 `j` 位置上的字符相等, `match[j]` 会有可能比 `match[j-1]+1` 更大吗? **不可能**



`match[j] = match[j-1] + 1` (最多持平啦, 利用反证法证明)

且能得到这个结果的前提是运行很好 `if (pattern[match[j-1]+1] == pattern[j])`

当 `pattern[match[j-1]+1] != pattern[j]` 时, 下一个待与 `pattern[j]` 比较的元素下标是: `match[match[j-1]+1]+1`



## KMP-5. BuildMatch的编程实现

```
void BuildMatch(char *pattern, int *match)
{
    int i, j;
    int m = strlen(pattern); // 复杂度 O(m)
    match[0] = -1;
    for (j = 1; j < m; j++) { // 复杂度 O(m)
        i = match[j-1]; // 把这个值存入 i 里面, 后面就可以直接用 i 来表示, 不至于显得很长很啰嗦, 怎么感觉像数学的换元法啊哈哈
        while ((i >= 0) && (pattern[i+1] != pattern[j])) // 每次都考虑最坏情况的话复杂度可能为 O(j) 了, 每次都退到底的话
            i = match[i]; // 让 i 做了一个回退, 回退到 while 条件两者有其中之一不发生的时候
        if (pattern[i+1] == pattern[j])
            match[j] = i+1; // i 回退总次数不会超过 i 增加的总次数
        else match[j] = -1;
    }
}
```

整个算法复杂度:  $T_m = O(m)$