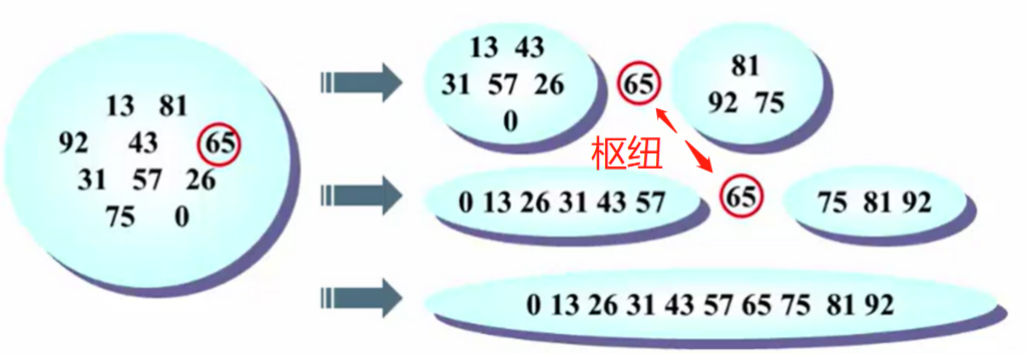
数据结构第十周笔记——排序(下)(慕课浙大版本--XiaoYu)

10.1 快速排序

10.1.1 算法概述

快速排序的算法跟归并函数的算法差不多，策略都是分而治之的策略

1. 分而治之
    1. 主元(pivot)=>中枢枢纽的意思



伪码描述

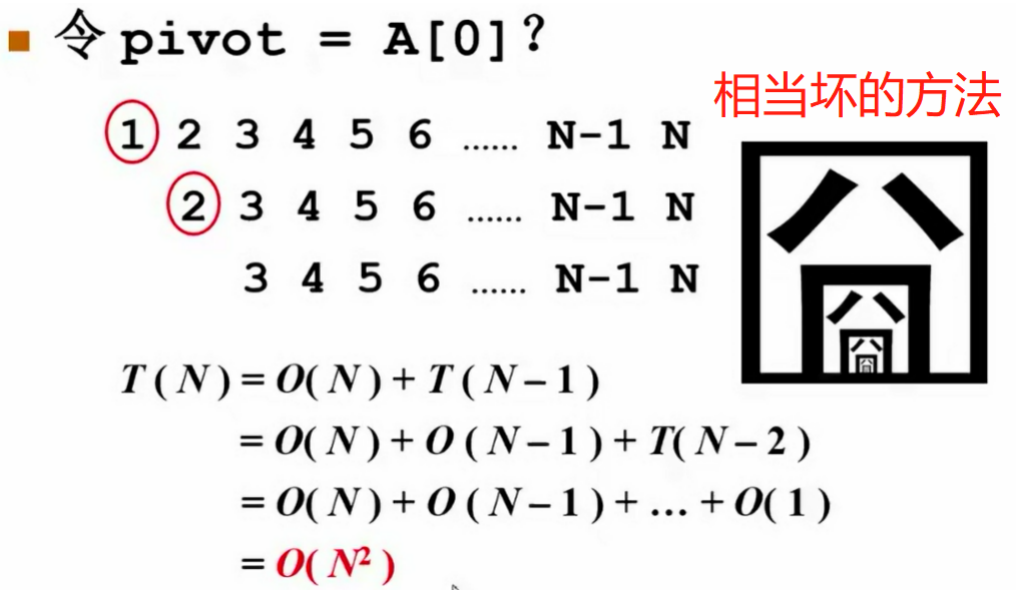
```
void Quicksort( ElementType A[],int N )
{
    if( N < 2 ) return;
    pivot = 从A[]中选一个主元;//主元的选择决定了快速排序到底快不快
    将S = { A[] \ pivot }将除了主元以外的元素分成两个独立子集://怎么分
        A1 = {a属于S | a <= pivot }和A2 = {a属于S | a >= pivot };//一部分由小于等于
pivot元素来组成的，另一部分由大于等于pivot元素组成
    A[] = Quicksort(A1,N1) U {pivot} U Quicksort(A2,N2);
}
```

什么是快速排序算法的最好情况？每次正好中分

什么是快速排序算法的最好情况？

每次正好中分 ⟶ $T(N) = O(N\log N)$

10.1.2 选主元

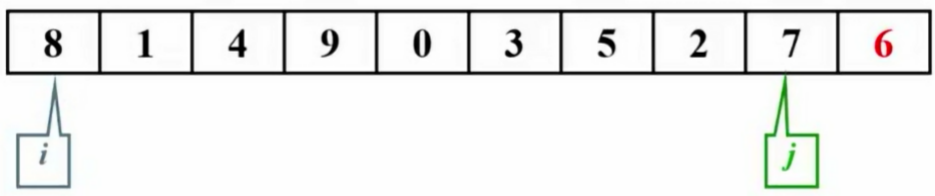


1. 随机取pivot?rand()函数不便宜=相当花费时间
2. 取头、中、尾的中位数
    1. 例如8、12、3的中位数就是8
    2. 测试一下pivot()不同的取法和堆运行速度有多大影响?

```
ElementType Median3( ElementType A[],int Left,int Right )
{
    int Center = ( Left + Right ) / 2;
    if( A[ Left ] > A[ Center ] )//三步的比较跟交换(保证从左到右的大小顺序)。左边比中间大
        Swap( &A[ Left ],&A[ Center ] );
    if( A[ Left ] > A[ Right ] )//左边比右边大
        Swap( &A[ Left ],&A[ Right ] );
    if( A[ Center ] > A[ Right ] )//中间比右边大
        Swap( &A[ Center ],&A[ Right ] );
    //这样三步交换下来，左边一定是最小的那个
    Swap( &A[ Center ],&A[ Right-1 ] );//将pivot藏到右边(为了之后方便，先将Center放到现在需要考虑的子列的最右边)，然后就只需要考虑A[Left + 1]....A[ Right - 2]
    return A[ Right - 1];//返回pivot
}
```

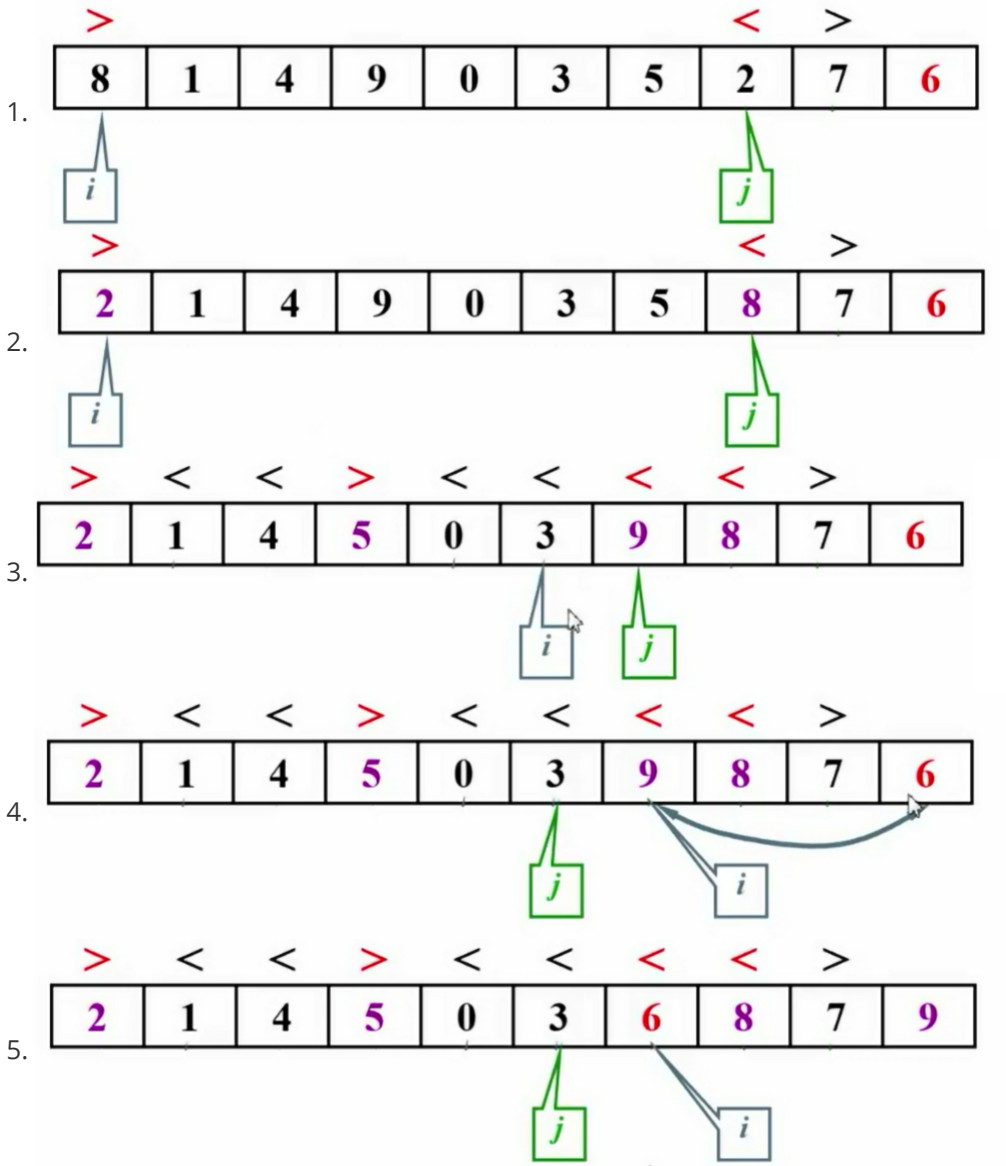
> 字符串交换(swap)swap操作实现交换两个容器内所有元素的功能。要交换的容器的类型必须匹配： 必须是相同类型的容器，而且所存储的元素类型也必须相同。调用了swap函数后，右操作数原来存储的元素被存放在左操作数中，反之亦然。

10.1.3 子集划分



1. i和j不是C语言的指针意义，而是指向存放位置的意思
2. 6是主元，被藏到了最右边的位置

3. 将主元和i与j进行比较(比较完指针i与j当前的位置后向里靠)，发现大小符号相反的之后i与j当前的数值对调，如下图



以上就是快速排序为什么快的原因：

1. 每次他选定主元之后，这个主元在子集划分完成以后，他就被一次性的放在他最终的正确位置上
2. 不像插入排序，每次做了元素交换以后，这个元素所待的位置只是临时的，当下一张新的扑克牌插进来的时候，这些牌所有的都要往后错，一张牌的插入就可能要牵扯到多次的移动

上述快速排序需要考虑的问题：

1. 如果有元素正好等于pivot怎么办?
    1. 停下来做交换？ 当所有元素都相等的时候会做很多很多次完全没有用处的交换。但做了很多次无用的交换后最终我们的主元会被换到中间的位置(好处：每次递归的时候这个原始的序列都被基本上等分成两个等长的序列)复杂度NlogN
    2. 不理他，继续移动指针？ 避免了很多次无用的交换，但每次子集划分的时候，主元都是被放在某一个端点的(就复杂度又变成了N^2了)
    3. 结论：还是选择停下来交换划算点

小规模数据的处理

1. 快速排序的问题
    1. 用递归....
    2. 对小规模的数据(例如N不到100)可能还不如插入排序快

2. 解决方案

    1. 当递归的数据规模充分小，则停止递归，直接调用简单排序(例如插入排序)
    2. 在程序中定义一个Cutoof的阈值(决定什么时候不递归)

10.1.4 算法实现

```
void Quicksort( ElementType A[],int Left,int Right )
{
    if( Cutoff <= Right - Left ){
    Pivot = Median3( A,Left,Right );//pivot是主元的意思，在这里返回的不仅仅只是一个主元的值
    //这里的Left参数是最小值，Right参数是最大值。真正的主元被藏在了Right-1的地方
    i = Left; j = Right - 1;
    for(;;){
        while(A[ ++i ] < Pivot ){}
        while(A[ --j ] < Pivot ){}
        if( i < j)
            Swap( &A[i],&A[j] );//i < j则证明中间还有其他元素，这时候就可以调换
        //如果i > j则这个子集划分应该结束了
        else break;
    }
    Swap( &A[i],&A[Right-1]);//把藏在right-1这个位置的主元换到A[i]的位置上面去
    Quicksort(A,Left,i-1);//递归的左半部分
    Quicksort(A,i+1,Right);//递归的右半部分
    }
    else
        Insertion_Sort(A+Left,Right-Left+1);//Right-Left+1：待排序列的总个数；A+Left：开始的地方
}
```

快速排序的标准接口应该怎么写?

```
void Quick_Sort(ElementType A[], int N)
{
   /* 这里写什么？如下*/
    Quicksort( A, 0, N-1 );
}
```

10.2 表排序

10.2.1 算法概述

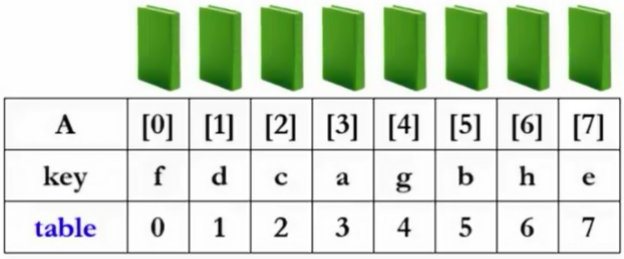
什么时候会用到表排序：

1. 待排元素不是一个简单的整数，而是一个庞大的结构(比如说是一本书)
2. 表排序在实际上是不需要移动原始数据的，移动的是指向他们位置的指针

间接排序：

1. 不移动元素本身，只移动指针

2. 定义一个指针数组作为"表"(table)



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

3. 交换的只是table的整数(指针)，得到

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

10.2.2 物理排序

N个数字的排列由若干个独立的环组成，意思如下：

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

table值的3跳到1再跳到5最后跳到0形成一个环。而环与环之间是独立互不交集(不相干)的

如何判断一个环的结束？每访问一个空位i后，就令table[i]=i。当发现table[i]==i时，环就结束了。

复杂度分析：

1. 最好情况：初始即有序
2. 物理排序过程的最坏情况是：有N/2个环，每个环包含2个元素。需要3N/2次元素移动
3. T = O(mN),m是每个A元素的复制时间

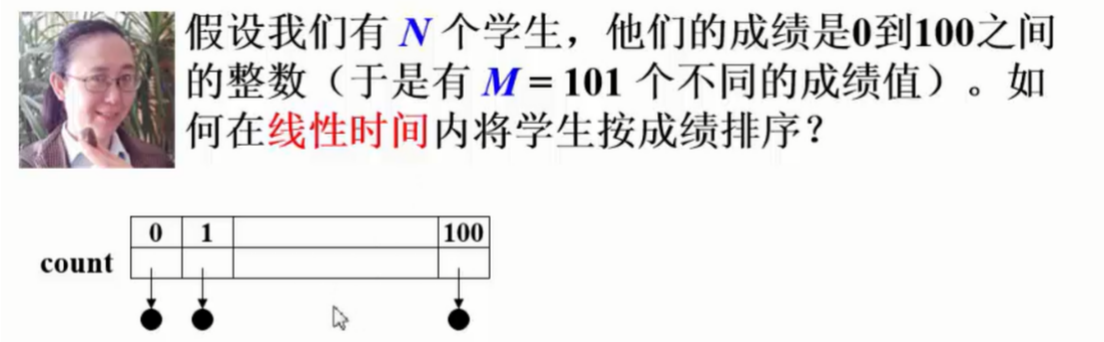
10.3 基数排序

仅仅基于比较进行的排序所有的这些算法他的最坏时间复杂度下界是O(NlogN),也就是说不管有多快我们总能制造出一个最坏的情况让他用最快的算法跑他也只能跑到NlogN。

这个时候除了比较之外有没有更快的排序呢？有，那就是基数排序

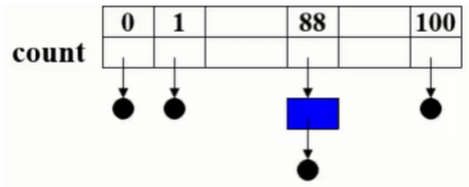
10.3.1 桶排序

基数排序是桶排序的升级版



count是数组，这个数组的每一个元素都是一个指针，一开始被初始化为空链表的头指针，所以一开始有101个空链表(对应了101个空的桶)

假设一个学生考88分：先找到88这个桶，然后把学生信息插到这个链表的表头里




```
伪码描述
void Bucket_Sort(ElementType A[],int N)
{
    count[]初始化;
    while(读入一个学生成绩grade)
        将该生插入count[grade]链表;
    for( i=0;i<M;i++){
        if( count[i] )//桶不为空
            输出整个count[i]链表;
    }
}
桶排序算法的时间复杂度T(M, N)是多少？   T(N,M) = O(M+N)
    当M非常小的时候(例如4w个学生只有101个不同成绩值，那这个时候其实就相当于一种线性的算法)
```

10.3.2 基数排序

如果M>>N的话怎么办?

假设我们有 $N=10$ 个整数，每个整数的值在0到999之间（于是有 $M=1000$ 个不同的值）。还有可能在线性时间内排序吗？

值是0-999之间，最多一共就三位数。我们在考虑三位数的时候每一位数只有十种可能。

基数就是这个进制的基数，二进制的基数就是2，八进制基数就是8，所以十进制的基数自然就是10

```
输入序列：64，8，216，512，27，729，0，1，343，125
    用"次位优先"(Least Significant Digit)=>简称LSD算法
    //什么是次位优先？假设目前手里是216，这个时候6 个位数是最次位，2 百位数是主位(有一种算法是主位优先)
    //比较先从个位数开始
```

第一步：建立十个桶 Bucket

0	1	2	3	4	5	6	7	8	9

第二步：根据**个位数**把他们放入相应的桶里面

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729

第三步：根据十位数放入相应的桶里面：

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							

最后一步：根据百位数放入相应的桶里面：

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

设元素个数为N，整数进制为B，LSD的趟数为P，则最坏时间复杂度是：T=O(P(N+B))

次位优先LSD算法

对于给定范围在0-999之间的10个关键字{64，8，216，512，27，729，0，1，343，125}
① 先为最次位关键字建立桶（10个），将关键字按最次位分别放到10个桶中
② 然后将①中得到的序列按十位放到相应的桶里
③ 做一次收集，扫描每一个桶，收集到一个链表中串起来
④ 将③中得到的序列按最主位放到桶中
⑤ 最后做一次收集，这样就得到一个有序的序列了

10.3.3 多关键字的排序

基数排序不仅仅用于处理整数的基数，还可以用于处理有多关键字的排序

一副扑克牌是按2种关键字排序的

K^0[花色] ♣ < ♦ < ♥ < ♠

K^1[面值] 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

有序结果：2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

采用"主位优先"(Most Significant Digit)排序：为花色建4个桶
在每个桶内分别排序，最后合并结果
=>其实我们还是需要在每个桶内调用相应的排序算法来解决，但是总体的时间复杂度会比我们把它完整的看成一个待排数组来排稍微好一点

采用"次位优先"(Least Significant Digit)排序：为面值建13个桶
这个方便就不需要再排序了，直接将结果合并，然后再为花色建4个桶就ok了

从上面两个例子来看，我们可以看出次位优先比主位优先要聪明得多，时间复杂度也快不少

10.4 排序算法的比较

前三个是简单排序，时间复杂度都是比较差的。优点在于程序非常好写，很短且不需要额外的空间

冒泡排序跟直接插入排序因为每次都交换两个相邻的元素，虽然慢，但是稳定

简单选择排序是跳着交换的，导致它是不稳定的

希尔排序是最先打破下界的算法，d最坏情况下是2，这个还是取决于增量排序，因为这是跳着排，所以也不稳定

堆排序跟归并排序理论来说，他们的时间复杂度都是最好的为NlogN

归并排序的缺点：需要一个额外的空间。当我们要排的数据量非常大的时候，归并排序会导致我们只能排一半的数据，本来可以排下的数据因为空间的问题而排不下。但好处在于稳定

排序方法	平均时间复杂度	最坏情况下时间复杂度	额外空间复杂度	稳定性
简单选择排序	$O(N^2)$	$O(N^2)$	O(1)	不稳定
冒泡排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
直接插入排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
希尔排序	$O(N^d)$	$O(N^2)$	O(1)	不稳定
堆排序	O(NlogN)	O(NlogN)	O(1)	不稳定
快速排序	O(NlogN)	$O(N^2)$	O(logN)	不稳定
归并排序	O(NlogN)	O(NlogN)	O(N)	稳定
基数排序	O(P(N+B))	O(P(N+B))	O(N+B)	稳定