

管中窥豹

——从插入排序的改进
和优化看算法的艺术



朴素的插入排序

- ◆ 最简单的插入排序,我们可以用如下简洁的代码完成:
- ◆ `template <class Item>`
- ◆ `void sort(Item a[], int l, int r){`
- ◆ `for(int i=l+1;i<r;i++)`
- ◆ `for(int j=i;j>l;j--)`
- ◆ `if(a[j-1]<a[j])`
- ◆ `exchange(a[j-1],a[j]);`
- ◆ `}`

简单的优化

- ◆ 上面的程序是很直观，很容易理解的。但是显然这里面有很多低效率的部分。
- ◆ 下面分析上面程序3个明显的冗余。





1、循环体中冗余的判断

- ◆ `for(int j=i;j>l;j--)`
- ◆ `if(a[j-1]<a[j])`
- ◆ `exchange(a[j-1],a[j]);`
- ◆ 这段代码中循环的终止条件是`j=1`，在每一次循环中，程序无条件的进行了`if(a[j-1]<a[j])`的判断。事实上，一旦这个判断为假之后，我们就没有必要再去做任何的判断了，因为`a[i]`已经位于了它正确的位置了。



2、循环条件冗余的判断

- ◆ 每一次循环我们都需要判断 $j>1$ 的条件。然而事实上，很少有情况我们的程序会运行到 $j=1$ （会执行到此处的情况是当前待插入的值应该被插入到数组头位置）。这条判断显得很冗余。
- ◆ 回忆我们在数据结构与算法课程中顺序查找时使用的监视哨方法，我们可以采用下面的做法：在程序开始时把最小的数放在 $a[0]$ 位置。




3、过多的移动

- ◆ 原始的算法里每一次交换进行了3次移动。我们假定这样的交换进行了 k 次。然而，我们可以发现，经过 $3k$ 次的移动之后，原来的数组发生的变化是已排好序的部分有 k 个数向后移动了1为， $a[i]$ 放在了空出来的位置。这样的结果我们完全可以通过 $k+1$ 次移动完成。

稍加优化后的伪代码

- 1、找出最先出现的最小元素，交换到待排序数组的首位置。
- 2、对 $i = 2, 3, \dots, n$ 进行循环，施以步骤3-6。
- 3、置 $j = i - 1, K = K_i, R = R_i$ 。
- 4、如果 $K \geq K_j$ 转向步骤6。
- 5、置 $R_{j+1} = R_j, j = j - 1$ 。返回步骤4。
- 6、 $R_{j+1} = R$

下面将精确分析这个算法的时间代价。



这个程序的运行时间是 $9B+10N-3A-9$ (Knuth)。

其中 A 是步骤 5 中 j 减少到 0 的次数； B 是移动的次数，也就是序列中逆序对的个数。对于 A, B 可以给出如下数学估计：

A ：最小值 0；最大值 $N-1$ ；数学期望是 H_n-1 ；标准差为

$$\sqrt{H_n - H_n^{(2)}} \quad (H_n = \sum_{i=1}^n \frac{1}{i} = \Theta(\lg n), \quad H_n^{(2)} = \sum_{i=1}^n \frac{1}{i^2} = \Theta(1))$$

B ：最小值 0；最大值 $\frac{N^2-N}{2}$ ；数学期望： $\frac{N^2-N}{4}$ 标准差为

$$\frac{\sqrt{n(n-1)(n+2.5)}}{6}$$

这些式子的证明涉及高深的数学知识，这里从略。有兴趣的同学可以参见 TAOCP 1.2.10 5.1.1 节相关内容。

进一步可以得到该算法的平均时间： $2.25n^2 + 7.75n - 3H_n - 6$

历尽艰辛，我们得到了这个算法精确的时间代价。



二分插入

- ◆ 目前为止，我们在寻找插入位置和实际的移动过程中都没有从数量级上予以降低。通过二分插入的方法，我们可以有效的在 $O(n \log n)$ 的时间内完成所有的比较。
- ◆ 这种做法看似非常的优秀。但是事实上他只解决了一半的问题。因为在移动的时候我们的代价依然是 $O(n^2)$
- ◆ 其次的一点是，这种考虑从纯数学的角度来看是对的，但是如果考虑到硬件的情况，他不仅可能意义不大，甚至不如朴素的做法。H.Nagler指出，当记录长80字符，在IBM705上对与超过128个记录的排序不应该推荐这种做法。

二路插入

同样移动的代价也是可以降低的。下面的做法（早在上世纪50年代就被提出）虽然没有从量级给予降低，但是他的效率还是比之前的提高了1倍。

Table 2
TWO-WAY INSERTION

					503				
				087	503				
			087	503	512				
		061	087	503	512				
		061	087	503	512	908			
	061	087	170	503	512	908			
	061	087	170	503	512	897	908		
061	087	170	275	503	512	897	908		

表中的第一项被放置在一个区域的中心，通过向右或者向左来腾出空间（就看哪种方便）。

似乎这种做法需要1倍的额外空间。但是通过循环表的方式，可以把额外的空间代价压缩掉。

Shell排序

- ◆ Shell排序首先有Donald.L.Shell提出。
- ◆ 如果在插入排序中一次只把一个元素移动1个位置，不可能把算法的时间改造成平方级别以下。

设 $a_1a_2\cdots a_n$ 是 $1\sim n$ 的一个全排列

$\frac{1}{n} \sum_{i=1}^n |a_i - i|$ 就是排序过程中某个记录所要走过的平均纯距离

计算一下上式的期望 $E\left(\frac{1}{n} \sum_{i=1}^n |a_i - i|\right) = \frac{n-1/n}{3}$

Shell排序的思想是，通过非相邻的数据项进行交换来提高效率。实际上，Shell排序只不过是插入排序的步长上进行了修改。



Table 3

SHELLSORT WITH INCREMENTS 8, 4, 2, 1

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
8-sort:																
	503	087	154	061	612	170	765	275	653	426	512	509	908	677	897	703
4-sort:																
	503	087	154	061	612	170	512	275	653	426	765	509	908	677	897	703
2-sort:																
	154	061	503	087	512	170	612	275	653	426	765	509	897	677	908	703
1-sort:																
	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Table 4

SHELLSORT WITH INCREMENTS 7, 5, 3, 1

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
7-sort:																
	275	087	426	061	509	170	677	503	653	512	154	908	612	897	765	703
5-sort:																
	154	087	426	061	509	170	677	503	653	512	275	908	612	897	765	703
3-sort:																
	061	087	170	154	275	426	512	503	653	612	509	765	677	897	908	703
1-sort:																
	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908


Shell排序的代码

```
template <class Item>
void shellsort(Item a[],int l,int r) {
    int h;
    for(h=1;h<=(r-1)/9;h=3*h+1);
    for(;h>0;h/=3) {
        for(int i=l+h;i<=r;i++) {
            int j=1;
            Item v=a[i];
            while(j>=l+h && v<a[j-h]) {
                a[j]=a[j-h];
                j-=h;
            }
            a[j]=v;
        }
    }
}
```



Shell排序的效率分析

- ◆ Shell排序的效率分析至今是一个难题，对于不同的增量序列，时间代价可能相差甚远。
- ◆ 各种效率分析的论文无不像一个数学迷宫。显然这里并不打算进行各种深奥的数学推演。这里列出的是有趣的且比较常用的一些结果。
- ◆ （对于关心证明的读者，建议去看Knuth的TAOCP，这里面比较全也比较难，在M.A.Weiss的著作里给出了几个比较简单的增量序列的时间代价分析）



Knuth 1969 年提出了一个简单的增量序列 $h_0 = 1, h_{s+1} = 3h_s + 1$, Knuth 自称似乎和任何其他增量一样好（当规模不是很大的情况下）。

他的学生 Sedgewick 评价说，这个算法“效率相对还可以”。可以证明使用这个增量序列的算法时间复杂度以 $O(n^{3/2})$ 为上界（不一定紧）。

Sedgewick 指出使用 1 8 23 77 281 ……可以有 $O(n^{4/3})$ 的上界。

Hibbard增量序列

Hibbard 提出了一个如下形式的增量序列: $\{1, 3, 7, \dots, 2^k - 1\}$ 。

目前得到的上界是 $O(n^{1.5})$ 。

试验模拟表明甚至可以达到 $O(n^{1.25})$ ，这个结果尚未得到证明。

(M.A.Weiss "Empirical Results on the Running Time of Shellsort", Computer Journal, 34 1991)



并不是所有的增量序列都是好的

- ◆ 1, 2, 4, 8, 32.....
- ◆ 这就是Shell当年提出Shell排序时所使用的增量序列。
- ◆ 这个算法在最坏的情形下情况非常糟糕，它退化成了一个二次的运行时间。
- ◆ 比如当奇数位保存的均为较大的数，而较小的数全部位于偶数位上时。



Pratt方法:

也不是只有两两互素的序列才好

1				
2	3			
4	6	9		
8	12	18	27	
16	24	36	54	81

以1, 2, 3, 4, 6,
9, 8, 12.....为增
量序列的Shell排序
可以做到:

$$O(n \log^2 n)$$

每一个数是它又上方数的两倍



没有任何增量序列可以给出 $O(n \log n)$ 的最坏运行时间。

这个结果属于 *C. G. Plaxton, B. Poonen, and T. Suel*. Improved lower bounds for Shellsort. In. Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science 1992

所以有人说“这些序列的结果表明 Shell 排序的时间代价已经很接近线性了”，这是值得商榷的。严谨的说法是，很接近 $n \log n$ 了。

n	O	K	G	S	P	I
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

O 1 2 4 8 16 32 64.....

K 1 4 13 40 121 364.....

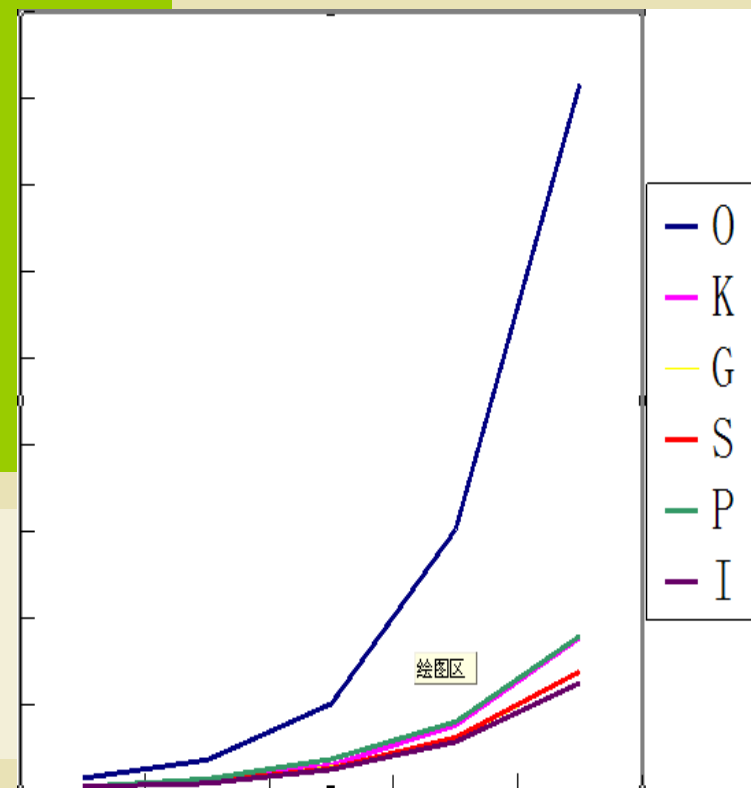
G 1 2 4 10 23 51..... $[\alpha^{i+1}]$


S 1 8 23 77 281..... $4^{i+1} + 3 \times 2^i + 1$

P 1 7 8 49 56 64一种变形的 pratt 序列

I 1 5 19 41 109.....两种不同的序列结合的结果

当规模为12500时，使用朴素的插入排序的时间大约是1100；稍加优化后的约是500。






◆ Shell排序是一个典型的简单算法呈现出复杂特性的例子。对Shell排序效率的研究要求高超的数学分析技术，这方面的研究绵延了数十年，至今依然是令人困惑的数学问题，没有人能回答对于很大的 N ，哪个增量序列是最好的。R.Sedgewick "Anlysis of Shellsort and related algorithms" 里有Shell排序最近的进展。

◆ 上面我们简要的列出了一些有意思的结果，更多的结果可以参见Knuth系列第三卷中的内容。最后给出的表格告诉我们对步长序列进行设计的一系列方法的实际运用中确实大大提高了我们的效率。

◆ 即使对于大的文件，Shell排序也具有较高的效率。更重要的是，他的实现非常简单。至今，很多排序的程序都在使用Shell排序。相对于一般我们常提到的一些 $n\log n$ 的算法，除非 n 特别大，否则他们并不会比Shell优秀太多。
(*Algorithm in C++ Part 3*)

- 
- ◆ 对于Shell排序的介绍就此为止。
 - ◆ 下面，我们将回到最初的直接插入排序。下面的方法通过改进数据结构提高了插入的效率。



表插入

- ◆ 我们知道，直接插入排序主要在进行如下两个步骤：
 - ◆ （1）找出一个恰当的插入位置
 - ◆ （2）为即将插入的元素准备插入空间，并将其插入。
- ◆ 之前的算法使用的是顺序表的结构，在期望意义下，需要进行一半元素的移动。但是如果改成链式结构，可以在常数时间内完成插入。

效率分析

表插入算法的时间代价是 $7B+14N-3A-6$

注意到之前的算法的时间复杂度是 $9B+10N-3A-9$

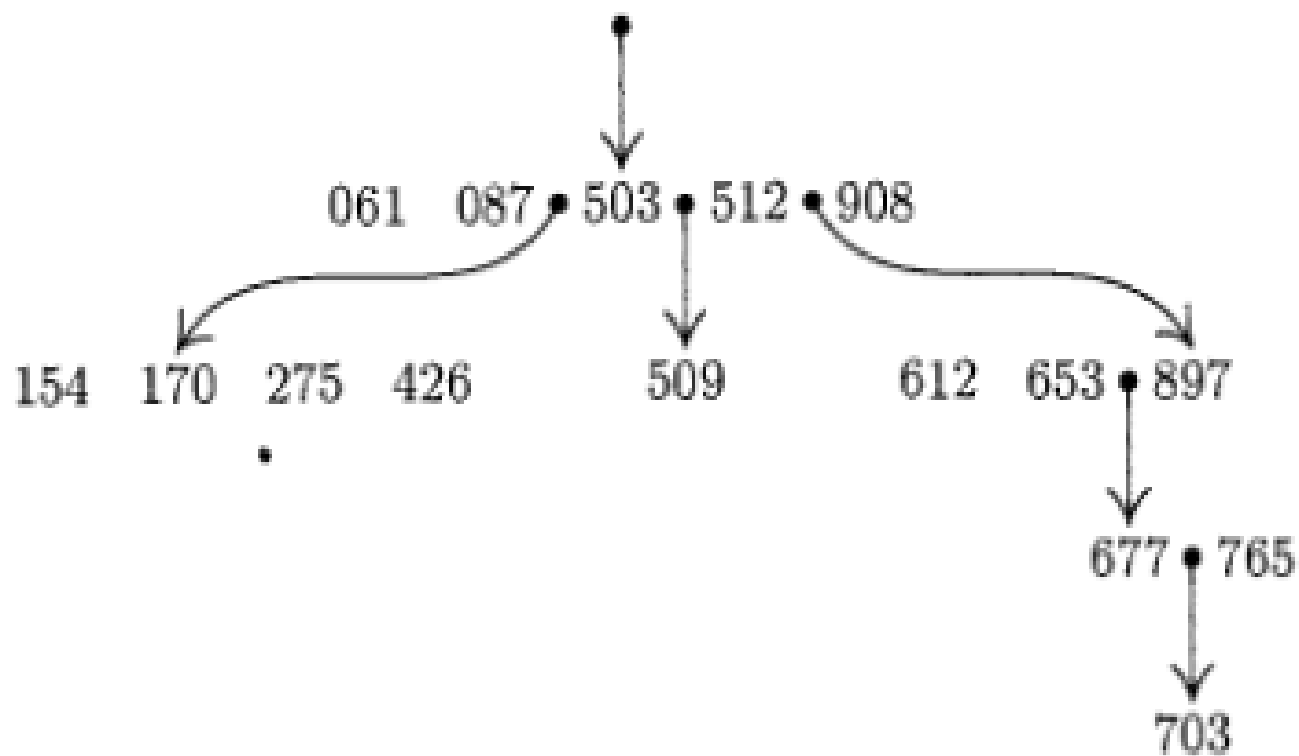
注意到 B 的期望是 $\frac{N^2-N}{4}$, 可见这个算法比之前的算法有了很大的改进, 实践表明这可以节省 22% 的时间。

还可以证明 (参见 TAOCP) B 前面的系数可以进一步降低到 5。这样我们几乎可以节省一半的时间。



- ◆ 和二叉插入同样的，这依然只解决了一半的问题（移动的问题），链式的结构将不支持二分查找。
- ◆ 一个自然的问题就是能不能把两者的优点进行整合。
- ◆ 答案是肯定的。这方面最先给出回答的是D.J.Wheeler(1957)。

Wheeler树插入方案





Wheeler树插入方案效率分析

使用二路插入,当有必要移动元素的时候通过开辟新的内存空间和使用指针代替数据的移动;同样的方法递归应用于新伸长出来的树枝。

可以看到,这样的结构最坏的情况是会退化成一棵每个节点上有两个数的独枝树(一个线性表),在这种情况下每一次插入的平均代价依然是 $\Theta(n)$ 的,但是前面的系数减小了一半。类似于二叉树的随机插入分析,可以得到平均情形下每一次操作都是 $O(n \log n)$ 的。或者使用类似AVL和Splay Tree的技术可以把最坏的情形下控制 $O(n \log n)$ 。当然,这样的操作绝非易事。

C.M.Berners-Lee 1958 提出了一个更加简单的改进。




分批处理

- ◆ 考虑这样的一个问题：假定现在需要对1000个元素排序，且前998个已经排好序，只需要将最后两个数插入到适当位置。是否仍然需要对整个序列进行两遍扫描？

- ◆ 如果事先比较好他们的大小，我们就可以通过1次扫描完成对他们的插入。

- ◆ 计算表明这样的做法可以比原始做法节省约33%次的比较和移动次数。

- ◆ 分批处理：多个任务一起进行。

The image is a vertical collage on the left side of the slide. It features a compass rose at the bottom, a sword hilt in the middle, and a circular medal or seal at the top. The background of the slide is a solid light blue color.


下面对目前所做的工作做一个总结：

我们开始于一个最简单自然的插入排序，然后提出了 3 个明显的改进方案。改进过后的算法依然在期望意义上有 $\frac{n^2}{4}$ 次的比较和移动。

然后我们根据两条线索对这个算法进行了优化

(1) 不改变插入排序的步长，从插入排序的比较和移动两个方面进行优化。提出了二叉插入（比较的次数降低到 $n \log n$ ），二路插入（移动的次数降低到 $\frac{n^2}{8}$ ）。采用链式结构可以把移动次数（改变链接）降低到 $2n$ 。最后我们结合了二叉插入和二路插入的优点，介绍了 Wheeler 树方法以及分批处理。

(2) 改变插入排序的步长，从而我们从量级上改变了插入排序的代价，也就是 Shell 排序。Shell 排序的分析至今是个难题，所有的代价分析都非易事。这里介绍了相关的结果，并给出了一些实践中的效果。



People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

The problem of efficient sorting remains just as fascinating today as it ever as.

-----Knuth



- ◆ 关于排序的主要参考书是Knuth系列的第三卷，里面几乎包含了上面讨论的每一个问题，除了内容丰富外，该书的一个显著特点就是所有结论都有完整的数学上的分析做为依托，当然不一定每个人都喜欢这种风格。
- ◆ 由Knuth的学生同时也是著名的算法大师R.Sedgewick撰写的Algorithm in C++ Part3中也详细的介绍了各种排序的算法。这本书的特点是难度适中，舍弃了过于繁复的数学推导。
- ◆ Baeza-Yates,Gonnet的《算法与数据结构手册》中有更新的关于排序的参考资料。



参考文献

- ◆ CLRS *Introduction to Algorithm* 2nd ed.
- ◆ D.E.Knuth *The Art of Computer Programming* Vol1,3
- ◆ R.Sedgewick *Algorithm in C++* Part3
- ◆ Mark Allen Weiss *Data Structures & Algorithm analysis in Java*
- ◆ 张铭 王腾蛟 赵海燕 《数据结构与算法》



Thanks for attention!