我从未编写过的最漂亮的代码

报告人：樊玉伟
学号：00746121

作者：Jon Bentley

北京大学

outline

- 排序算法综述
- 作者思想与看法
- 快速排序的分析及其中的美妙算法
- 总结与反思

常见排序算法

- 插入排序：直接插入排序，Shell排序
- 选择排序：直接选择排序，堆排序
- 交换排序：冒泡排序，快速排序
- 归并排序

- 基数排序，桶排序

排序算法评判标准

- 时间代价：比较次数
  移动次数
- 空间代价：额外空间
  堆栈深度
- 稳定性：存在多个具有相同排序码的记录
  排序后这些记录的相对次序保持不变

算法比较

算法	最大时间	平均时间	最小时间	辅助空间	稳定性
直接插入排序	0(n2)	0(n2)	0(n)	0(1)	稳定
Shell排序	0(n3/2)	0(n3/2)	0(n3/2)	0(1)	不稳定
直接选择排序	0(n2)	0(n2)	0(n2)	0(1)	不稳定
堆排序	0(nlogn)	0(nlogn)	0(nlogn)	0(1)	不稳定
冒泡排序	0(n2)	0(n2)	0(n)	0(1)	稳定
快速排序	0(n2)	0(nlog n)	0(nlog n)	0(logn)	不稳定
归并排序	0(nlog n)	0(nlog n)	0(nlog n)	0(n)	稳定
堆排序	0(nlog n)	0(nlog n)	0(nlog n)	0(1)	不稳定

测试数据

数组规模	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
直接插入排序	0.00000047	0.00002	0.001782	0.1752	17.917	____	0.00011	0.35094
选择排序	0.0000011	0.000041	0.002922	0.2778	36.5	____	0.27781	0.29109
冒泡排序	0.0000016	0.000156	0.01562	1.5617	207.69	____	0.00006	2.4484
Shell排序	0.00000078	0.000016	0.000281	0.0038	0.0579	0.8204	0.00125	0.00687
堆排序	0.00000204	0.000027	0.000344	0.0042	0.0532	0.6891	0.00406	0.00375
快速排序	0.00000169	0.000021	0.000266	0.003	0.0375	0.4782	0.0019	0.00199
归并排序	0.00000219	0.000028	0.000375	0.0045	0.0532	0.5969	0.00364	0.0036

- 你最喜欢哪个算法？
- 为什么？

作者思想

- 通过删除代码来实现功能的提升
- 只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完善
- 良好的写作风格即为简练。省略不必要的字词
- 尽量将每一件事情变得简单
- 如果我有足够的时间，那么我的给你写的信就会越短

QuickSort算法

```
void quicksort(int l, int u)
{   int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

这句话有什么用？

QuickSort 评价及改进

- 优点：速度快，空间代价低
- 缺点：不稳定。处理有序情况效率较低
- 改进：

  随机化快速排序

  平衡化快速排序

  设置阀值—混合排序

QuickSort算法分析

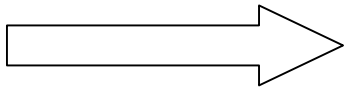
- 最好情况：每次选择的值均为中间值
  O(nlog n)次比较，
- 最坏情况：每次选择的值均为最值
  O(n^2)次

平均值呢？

算法内部比较次数分析

```
for (i = l+1; i <= u; i++)
 {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

comps++没必要



```
comps += u-l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

由于x元素师随机的，m的取值也是随机的

算法内部比较次数分析(2)

```
void quickcount(int l, int u)
{   int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

l，u对于排序是重要的，但此处只有u-

程序复杂度明显降低，运算复杂度为O(n)，存储为O(log n)

算法内部比较次数分析(3)

```
void qc(int n)
{   int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

转化为函数

```
int cc(int n)
{   int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-
      1) + cc(n-m);
}
```

平均比较次数

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

计算每一种情况下的比较次数

问题：算法复杂度

平均比较次数

- 运算复杂度为O(3^n)

原因：重复运算较多

解决办法：动态编程-->存储中间结果

平均比较次数

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

降低了复杂度，但增加了存储量

i从1到n时，t[i-1]+t[n-i]出现了两次

可将n-1拿到外面去

平均比较次数

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += 2*t[i]
    t[n] = n-1 +  sum/n
```

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

t[i]的前k项出现重复运算

数据

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

可以看出，数据增长速度不是很快

最终版本

如果不考虑中间数据
只求n对应平均比较次数

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

算法比较

示例	代码行数	答案类型	答案数量	运行时间	空间
2	13	Sample	1	n lg n	N
3	13				
4	8			n	lg N
5	8				
6	6				
7	6	Exact		3^N	N
8	6		N	N^2	N
9	6				
10	6				
11	4			N	
12	4	Exact	N	N	

理论分析

$$C_0 = 0; S_0 = 0;$$

$$S_n = S_{n-1} + 2C_{n-1};$$

$$C_n = n - 1 + S_n/n$$

$$C_n = (n+1)(2H_{n+1} - 2) - 2n \approx 1.386 n \lg n$$

总结

- 用最简洁的代码表示尽量多的功能
- 关注最重要的问题，暂时忽略不太重要的部分
- 尽量将每件事情变得简单，并且直到不可能再简单为止

思考

前面所列的几种排序哪个最快，哪个时空代价比最低

评判几种排序

如何类似作者的思路分析算法复杂度？