Chapter 2 插入排序以及归并排序

台运鹏

July 9th 2021

1 插入排序

首先是排序问题的定义：
输入：n 个数的序列 $<a_1,a_2,...,a_n>$
输出：输入序列的升序排列 $<a_1',a_2',...,a_n'>$

INSERTION-SORT(A)
for $j = 2$ **to** $A.length$ // 从第二个值开始
    $key = A[j]$
    // 将 $A[j]$ 插入已排序的序列 $A[1..j-1]$.
    $i = j - 1$ // 从该位置的前一个开始
    // 如果还在序列内且第 i 个大于 key
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$ // 往前推
    $A[i+1] = key$

2.1 问题解答：

LINEAR-SEARCH(A, v)
i = NIL
for $j = 1$ **to** $A.length$
    if $A[j] = v$
        $i = j$

```
ADD-TWO-BINARY-NUMBERS(A,B)
n = A.length
C[1, n+1]
for j = n to 1
    tem = A[j] + B[j]
    tem = tem + C[j+1]
    if tem ≥ 2
        C[j] = C[j] + 1
        tem = tem - 2
    C[j+1] = tem
```

2 算法分析

对于 for 循环而言，虽然循环体是运行了 $n-1$ 次，但是对于是否满足循环条件也会判定一次，因而是 $n-1+1=n$ 次，$while$ 亦同理。t_j 表示当 j 取一个值时，运行 $while$ 循环所需的次数，$while$ 循环内的语句因为并没有多加一次判断，因而是 t_j-1。默认注释语句是不会有代价的，即 $c=0$。

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] = key$	c_8	$n-1$

因而所需时间 $T(n)$ 可被以下式子表示：

$$T(n) = c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

最好的情况：

即输入的序列已经是有序的，那么，for 循环依旧进行，而 $while$ 体因不满足循环条件不会运行，但是 $while$ 循环判断依旧正常运行。由公式可知，$T(n)$ 是关于 n 的线性函数，因而记为 $\Theta(n)$，$\Theta(n)$ 是对函数的抽象表达，表示的是这个函数 n 的最高次是一次，也就是说这个函数的增长速度是由 n 决定的，做算法分析时，我们通常更在乎增长率或者增长量级。

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\
&= an + b
\end{aligned}
$$

最差的情况：

即输入的序列是降序排列的。那么，对于 $A[j]$ 来说，在 $while$ 循环时，就不免与前面 $A[1,\ldots,j-1]$ 的每一个数进行比较，加上最后一次的判断，$t_j = j$（计算 T_n 时为了排版美观，将 c_8 加到了上面）。由公式可知，$T(n)$ 是关于 n 的二次函数，因而即为 $\Theta(n^2)$。

$$
\begin{aligned}
\sum_{j=2}^{n} j &= \frac{(n+2)(n-1)}{2} \\
\sum_{j=2}^{n} j-1 &= \frac{n(n-1)}{2}
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1) + c_8(n-1) \\
&+ c_5\frac{(n+2)(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} \\
&= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n \\
&- (c_2 + c_4 + c_5 + c_8) \\
&= an^2 + bn + c
\end{aligned}
$$

2.1 问题解答

思路采取双指针，首先默认 j 是最小值索引，将 j 指着的序列向后拖一个就是第二个指针 i 指向的序列，然后如果发现有最小的，则刷新最小值索引。后面交换用代码时要注意要把一个值先储存起来，否则当替换发生一个时，两个值是一样的，无法完成第二个的替换。复杂度为 $\Theta(n^2)$。

经过上面的分析，不难发现如果真的想减少时间复杂度，减少循环语句是不错的选择。

```
SELECTION-SORT(A)
n = A.length
for j = 1 to n − 1
    smallest = j
    for i = j + 1 to n
        if A[i] < A[smallest]
            smallest = i
    将 A[j] 与 A[smallest] 交换
```

3 设计算法

上述提到的插入排序采用的是增量法，对已排好序的 $A[1\ldots j-1]$，再排一次，那么已排好的序列即为 $A[1\ldots j]$。接下来介绍算法常用的设计技巧——**分治**（$Divide\,And\,Conquer$）。它包括以下步骤：

- 拆分：将原问题拆解成与自身相似的若干子问题
- 递归：递归地解决各个子问题
- 合并：将各个子问题的答案合并起来

合并排序（$MERGE$）便是这样设计出来。

- 拆分：将原数组 $A[p,r]$ 拆解成两个子数组 $A[p,q]$ 和 $A[q+1,r]$
- 递归：利用合并排序递归地解决子问题
- 合并：最终将两个数组，排序结束

因为不断递归，最终所有的子问题都会是长度为 1 的序列，因而不会有时间损失，主要的时间复杂度则在于合并的过程中。假设我们现在有两堆朝上的扑克牌，分别排好序，首先比较两个堆顶的牌的大小，如果一个较小，则将其拿出，放到新的一堆底下。不断重复此过程。可以看出每一个元素都会比一次，但是当哪一堆为空时，与之比较的元素省去比较，但还会判断是否为空，因而时间复杂度为 $\Theta(n)$，n 为序列长度。

下面是关于合并的伪代码，不难发现两个数组 L, R 的长度都比应该的长度加了 1，多的位置用以存储**哨兵**，值为 ∞，继续上面的例子，当某一堆为空时，也就是到了某个子数组的最后，即为 ∞，那么，较小的肯定为与之比较的值，直到最后两堆均为空，受限于最后一个 for 循环，这时已经结束了。第 5 行，若索引从 $p+i$ 开始，那么第一个值对应的索引则为 $p+1$ 而不是 p，所以需要减 1，R 的初始赋值同理。

伪代码中有三个 for 循环，易得前两个时间复杂度分别为 $\Theta(n_1), \Theta(n_2)$，而 $\Theta(n_1+n_2) = \Theta(n)$，最后一个一共迭代 n 次，每次均为常量时间，因而 $MERGE$ 的复杂度为 $\Theta(n)$。

MERGE(A,p,q,r)

1 $n_1 = q - p + 1$
2 $n_2 = r - q$
3 让 $L[1, n_1+1]$ 和 $R[1, n_2+1]$ 是 A 的两个子数组
4 **for** $i = 1$ **to** n_1
5 　　$L[i] = A[p+i-1]$
6 **for** $j = 1$ **to** n_2
7 　　$R[j] = A[q+j]$
8 $L[n_1+1] = \infty$
9 $R[n_2+1] = \infty$
10 $i = 1$
11 $j = 1$
12 **for** $k = p$ **to** r
13 　　**if** $L[i] <= R[j]$
14 　　　　$A[k] = L[i]$
15 　　　　$i = i + 1$
16 　　**else** $A[k] = R[j]$
17 　　　　$j = j + 1$

接下来开始对整个数组的排序，当 $p \geq r$ 时，说明该数组只有一个数，则不需要排序，否则就递归地进行排序。

MERGE-SORT (A,p,r)

if $p < r$

  $q = \lfloor (p+r)/2 \rfloor$

  MERGE-SORT(A, p, q)

  MERGE-SORT$(A, q+1, r)$

  MERGE(A, p, q, r)

归并排序的主要步骤如下图所示，先不断拆分，直至只有一个元素，然后开始不断向上合并。

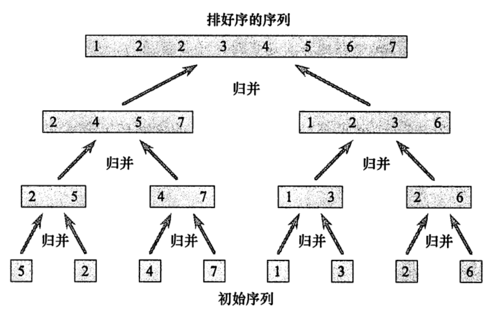

图 2-4 归并排序在数组 $A=\langle 5, 2, 4, 7, 1, 3, 2, 6\rangle$上的操作。随着算法自底向上地推进，待合并的已排好序的各序列的长度不断增加

在实际将伪代码变成代码时，需要注意的点：

- 很多时候不必像伪代码中那样写，伪代码的目的是为了让人能明白每一步发生了什么，实际应用时，应结合不同的编程语言的特性进行修改，而非抄写
- 伪代码为了便于理解，所有下标均从 1 开始，如 $A[1, n+1]$ 则表示 A 有 $n+1$ 个元素，而代码的索引从 0 开始，上述 1 两个伪代码中的 p, q, r 到了代码中自然变成了索引，具体查看 2_3.py。

3.1 时间分析

一般的分治算法分为**分解，递归求解，合并**，假设问题规模为 n，被分解为 a 个子问题（注：并不是分解到最底层的子问题，只是分解一次），每个子问题的规模为原问题的 $1/b$。假设子问题的规模小于等于某个常数 c 时，求解子问题所需的时间为 $\Theta(1)$，分解所需的时间记为 $D(n)$，合并所需的时间为 $C(n)$，那么，分治算法可被下列式子表示：

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ aT(n/b) + D(n) + C(n) & Other \end{cases}$$

假设输入序列的规模是以 2 的幂出现，如 2,4,8 等，在第四章将会证明这个算法的时间与规模无关，但取定合适的值的确有助于找规律。因为总长度为 2 的幂，所以 $a = b = 2$，但其他情况下，一般两者不相等，不过对于题解并无影响。计算分解位置只需要 $\Theta(1)$ 时间，而合并是 $\Theta(n)$，两者相加可以忽略 $\Theta(1)$。

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

为了便于理解，将上式中 $\Theta(1)$ 重写为 c，那么 $\Theta(n)$ 可被表示为 cn。

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

图 (b) 和图 (c) 中表现了分解的过程。不妨将此递归过程用树来表示，图 (c) 中节点上记录每次分解的代价，即为 cn，那么一开始是 cn，分解为 2 个规模为 $n/2$ 的子问题，那么，这两个子问题分别分解节点上就该是 $cn/2$，依次类推，分解到最后一层时，每个子问题的规模为 1，那么代价均为 c，具体见下图。

那么，该树一共有多少层呢？$lgn + 1$ 层，当然是以 2 为底的。用归纳法证明一下：

- $n = 1$，不用分解，即为 1 层，$lg1 + 1 = 1$
- $n \ge 2$，因为我们假设输入规模是 2 的幂的形式，所以假设 $n = 2^i$ 时满足层数为 $lg2^i + 1 = i + 1$，那么，$n = 2^{i+1}$ 时，比 $n = 2^i$ 时多了一层，即为 $i + 2$，而 $lg2^{i+1} + 1 = i + 2$，假设成立。

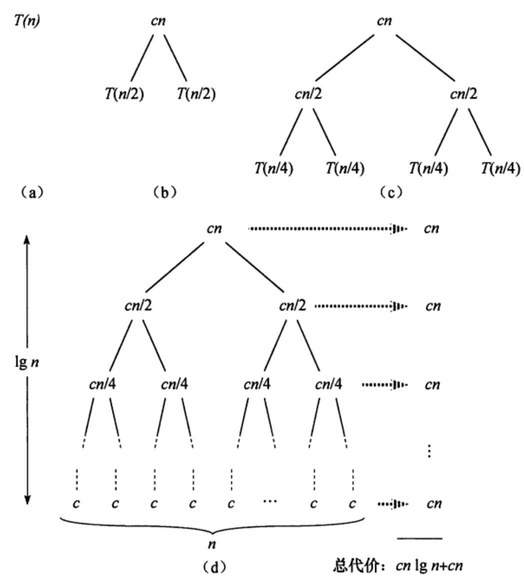

图 2-5 对递归式 $T(n)=2T(n/2)+cn$，如何构造一棵递归树。(a)部分图示 $T(n)$，它在(b)～(d)部分被逐步扩展以形成递归树。在(d)部分，完全扩展了的递归树具有 $\lg n+1$ 层(即如图所示，其高度为 $\lg n$)，每层将贡献总代价 cn。所以，总代价为 $cn\lg n+cn$，它就是 $\Theta(n\lg n)$

每一层所有节点的代价之和均为 cn，一共 $lgn+1$ 层，所以总代价即为 $cn(lgn+1)$，这里 $nlgn$ 比 n 的增长速度更快，因而后面的 cn 可被忽略，即总代价为 $\Theta(nlgn)$。

3.2 问题解答

2.3-3：

- $n=2$ 时，$T(2)=2=2lg2$
- $n\geq 4$ 时，假设 $n=2^k$ 满足 $T(2^k)=2^klg2^k=k2^k$，那么 $n=2^{k+1}$ 时，$T(2^{k+1})=2T(2^k)+2^{k+1}=k2^{k+1}+2^{k+1}=(k+1)2^{k+1}=2^{k+1}lg2^{k+1}$

2.3-4：

若要排 $A[1\ldots n]$，需要先把 $A[1\ldots n-1]$ 排好，然后把 $A[n]$ 插入，因为 $A[n]$ 具有随机性，所以最糟糕的情况是所有都找一遍，代价为 $\Theta(n)$，当只有一个值时，已排好，所以是 $\Theta(1)$。递归式如下：

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

最后一次分解是 $T(2) = T(1) + \Theta(2) = \Theta(1) + \Theta(n-(n-2))$，把所有代价相加，下面的式子是近似，当某个式子对于整体代价不起决定作用时，就丢掉不看。

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(n-(n-2)) \\ &= n\Theta(n) = \Theta(n^2) \end{aligned}$$

2.3-2

```
n1 = q - p + 1
n2 = r - q
让 L[1, n1], R[1, n2] 成为 A 的子数组
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
i = 0
j = 0
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    if L[i] > R[j]
        A[k] = R[j]
        j = j + 1
    if i = n1 + 1
        A[k + 1, r] = R[j, n2]
    if j = n2 + 1
        A[k + 1, r] = L[i, n1]
```

2.3-5:

提供两种思路：迭代和递归，通过双指针（分别指向数组的开头和结尾），不断夹逼，最终得到值。两者都是将数组一分为二进行比较，然后继续上述过程，比较的代价为 $\Theta(1)$，因而，表达式为 $T(n) = T(n/2) + \Theta(1)$，而树的高度为 $lgn + 1$，因而总代价为 $\Theta(lgn)$。

```
BINARY-SEARCH-ITERATIVE(A,v,low,high)
while low ≤ high
    mid = ⌊(low + high)/2⌋
    if A[mid] == v
        return mid
    elseif A[mid] < v
        low = mid + 1
    else
        high = mid − 1
return NIL
```

```
BINARY-SEARCH-RECURSIVE(A,v,low,high)
if low > high
    return NIL
mid = ⌊(low + high)/2⌋
if A[mid] == v
    return mid
elseif A[mid] < v
    return BINARY-SEARCH-RECURSIVE(A,v,mid+1,high)
else
    return BINARY-SEARCH-RECURSIVE(A,v,low,mid-1)
```

2.3-6:

当看到带有 lgn 时，就应该想到分治算法，这表示了多少次二分数组，首先我们用合并排序来使整个数组变成良序，这个代价为 $nlgn$，接下来再运用二分搜索，这个代价为 lgn，接下来外面套个 for 循环，求 lgn 求和 n 次，即为 $nlgn$，所以总和变为 $nlgn$。

```
INSERTION-SORT-BINARY-SEARCH(A,x)
n = A.length
MERGE-SORT(A,1,n)
for j = 1 to n
    index = BINARY-SEARCH(A, x − A[j])
    if index ≠ NIL and index ≠ j
        return True
return False
```

其实数组类还有一种更常见的思路，就是双指针，在二分搜索时也有这种做法，一般这种方法通过移动两端的指针最终得出答案。在本题中，前面用归并排序跟前面一样，指针的表达式是 $T(n) = T(n-1) + \Theta(1)$，损失易得为 lgn，而前面套了个 $while$ 循环，当 $low = high = n$ 时循环才结束，所以加起来损失为 $nlgn$，故总损失也是 $nlgn$。

```
INSERTION-SORT-TWO-WAY(A,x)
n = A.length
MERGE-SORT(A,1,n)
low = 1
high = n
while low < high
    if A[low] + A[high] == x
        return True
    elseif A[low] + A[high] > x
        high = high − 1
    else
        low = low + 1
return False
```