

## 第 2 章 算法基础

### 2.1 插入排序

INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

$$T(n) = \Theta(n^2)$$

**Ex2.1-2** 重写过程 INSERTION-SORT, 使之按非升序排序.

INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] < key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

**Ex2.1-3** 线性查找问题.

LINEAR-FIND( $A, v$ )

```
1  for  $j = 1$  to  $A.length$ 
2      if  $A[j] == v$ 
3          return  $j$ 
4  return NIL
```

**Ex2.1-4** 二进制数加法

BINARY-NUMBER-ADDITION( $A, B, n$ )

```
1  Let  $C[1 \dots n + 1]$  be a new array.  
2   $r = 0$   
3  for  $i = n$  downto 1  
4       $C[i] = (A[i] + B[i] + r) \% 2$   
5       $r = (A[i] + B[i] + r) / 2$   
6   $C[1] = r$   
7  return  $C$ 
```

## 2.2 分析算法

**Ex2.2-2: 选择排序** 首先找到  $A$  中最小元素并与  $A[1]$  进行交换, 然后找出次最小元素并与  $A[2]$  进行交换...

SELECTION-SORT( $A$ )

```
for  $i = 1$  to  $A.length - 1$   
     $min = i$   
    for  $j = i + 1$  to  $A.length$   
        if  $A[j] < A[min]$   
             $min = j$   
    Exchange  $A[min]$  and  $A[i]$ 
```

## 2.3 设计算法

MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays
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  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else
17          $A[k] = R[j]$ 
18          $j = j + 1$ 
```

MERGE-SORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

利用递归树, 可以得知 MERGE-SORT 的算法复杂度为  $\Theta(n \lg n)$

**Ex2.3-2** 重写过程 MERGE-SORT, 使之不使用哨兵, 而是一旦数组 L 或 R 的所有元素均被复制回 A 就立刻停止, 然后把另一个数组的剩余部分复制回 A.

MERGE-SORT-WITHOUT-GUARD( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  Let  $L[1 \dots n_1]$  and  $R[1 \dots n_2]$  be new arrays.
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   $i = 1$ 
9   $j = 1$ 
10  $k = p$ 
11 while  $i \leq n_1$  and  $j \leq n_2$ 
12     if  $L[i] \leq R[j]$ 
13          $A[k] = L[i]$ 
14          $i = i + 1$ 
15     else
16          $A[k] = R[j]$ 
17          $j = j + 1$ 
18      $k = k + 1$ 
19 while  $i \leq n_1$ 
20      $A[k] = L[i]$ 
21      $i = i + 1$ 
22 while  $j \leq n_2$ 
23      $A[k] = R[j]$ 
24      $j = j + 1$ 
```

**Ex2.3-4** 我们可以把插入排序表示为如下的一个递归过程。为了排序  $A[1 \dots n]$ , 我们递归地排序  $A[1 \dots n-1]$ , 然后把  $A[n]$  插入已排序的数组  $A[1 \dots n-1]$ 。为插入排序的这个版本的最坏情况运行时间写一个递归式。

INSERTION( $A, p$ )

```
1   $key = A[p]$ 
2   $i = p - 1$ 
3  while  $i > 0$  and  $A[i] < key$ 
4       $A[i + 1] = A[i]$ 
5       $i = i - 1$ 
6   $A[i + 1] = key$ 
```

INSERTION-SORT-RECURSIVE( $A, p$ )

```
1  if  $p > 1$ 
2      INSERTION-SORT-RECURSIVE( $A, p - 1$ )
3      INSERTION( $A, p$ )
```

其递归式如下：

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n = 2 \end{cases}$$

**Ex2.3-5** 假设  $A$  已排好序，为二分查找写出迭代式或递归的伪代码。证明二分查找的最坏情况运行时间为  $\Theta(\lg n)$

BINARY-SEARCH-RECURSIVE( $A, key, low, high$ )

```
1  if  $low \leq high$ 
2       $mid = \lfloor (low + high)/2 \rfloor$ 
3      if  $key < A[mid]$ 
4          return BINARY-SEARCH-RECURSIVE( $A, key, low, mid - 1$ )
5      elseif  $key > A[mid]$ 
6          return BINARY-SEARCH-RECURSIVE( $A, key, mid + 1, high$ )
7      else
8          return  $mid$ 
9  else
10     return NIL
```

该算法的递归式为  $T(n) = T(n/2) + \Theta(1)$ ，由递归树可知，其算法复杂度为： $\Theta(\lg(n))$

**Ex2.3-6** 注意到过程 INSERTION-SORT 的第 5-7 行的 while 循环采用一种线性查找来（反向）扫描已排好序的子数组  $A[1..j-1]$ 。我们可以使用二分查找来把插入排序的最坏情况总运行时间改进到  $\Theta(n \lg n)$  吗？

先将上面的 BINARY-SEARCH-RECURSIVE 算法修改为如下形式：

BINARY-SEARCH-RECURSIVE( $A, key, low, high$ )

```
1  if  $low \leq high$ 
2       $mid = \lfloor (low + high)/2 \rfloor$ 
3      if  $key < A[mid]$ 
4          return BINARY-SEARCH-RECURSIVE( $A, key, low, mid - 1$ )
5      elseif  $key > A[mid]$ 
6          return BINARY-SEARCH-RECURSIVE( $A, key, mid + 1, high$ )
7      else
8          return  $mid$ 
9  else
10     return  $low$ 
```

INSERTION-SORT-USING-BINARY-SEARCH( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4       $index = \text{BINARY-SEARCH-RECURSIVE}(A, key, 1, i)$ 
5      if  $A[index] < key$ 
6           $index = index + 1$ 
7      while  $i \geq index$ 
8           $A[i + 1] = A[i]$ 
9           $i = i - 1$ 
10      $A[i + 1] = key$ 
```

可见，在最坏情况下，就算使用了 BINARY-SEARCH-RECURSIVE 算法，也不能将其运行时间降低到  $\Theta(n \lg n)$ 。

**2.3-7** 描述一个运行时间为  $\Theta(n \lg n)$  的算法，给定  $n$  个整数的集合  $S$  和另一个整数  $x$ ，该算法能确定  $S$  中是否存在两个其和刚好为  $x$  的元素。

先用归并排序对  $S$  进行排序，其运行时间为  $\Theta(n \lg n)$ ，再使用二分查找算法查找  $x - e$ ，其运行时间为  $\Theta(n \lg n)$ ，总运行时间为  $\Theta(n \lg n)$ 。

## 思考题

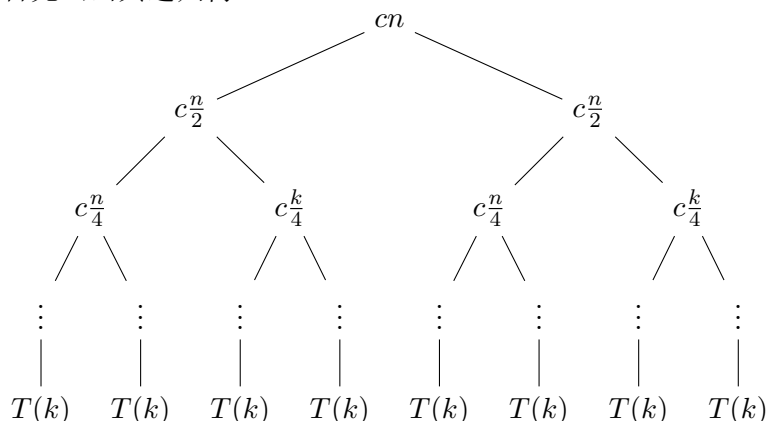
**2-1** 在归并排序中对小数组采用插入排序，虽然归并排序在最坏情况运行下时间为  $\Theta(n \lg n)$ ，而插入排序的最坏情况运行时间为  $\Theta(n^2)$ ，但是在插入排序中的常量因子可能使得它在  $n$  较小时，在许多机器上实际运行得更快。因此，在归并排序中当  $n$  较小时，采用插入排序来使递归的叶变粗是有意义的。考虑对归并排序的一种修改，其中使用插入排序来排序长度为  $k$  的  $n/k$  个子表，然后使用标准的合并机制来合并这些子序列，这里  $k$  是一个待定的值。

a. 证明：插入排序最坏情况可以在  $\Theta(nk)$  时间内排序每个长度为  $k$  的  $n/k$  个子表。

根据插入排序的特点，排序长度为  $k$  的序列在最坏情况下的运行时间为  $\Theta(k^2)$ ，那么排序  $n/k$  个这样的子序列则需要  $\frac{n}{k}\Theta(k^2) = \Theta(nk)$ 。

b. 表明在最坏情况下如何在  $\Theta(n\lg(\frac{n}{k}))$  时间内合并这些子表。

首先画出其递归树：



可知，该递归树总共有  $\lg(\frac{n}{k}) + 1$  层，除了最后一层的代价为  $\Theta(nk)$  外，其他各层的代价都是  $cn$ ，从最后一层向上都是在合并，所以总代价为： $cn\lg(\frac{n}{k}) = \Theta(n\lg(\frac{n}{k}))$

c. 假定修改后算法的最坏情况运行时间为  $\Theta(nk + n\lg(\frac{n}{k}))$ ，要使修改后的算法与标准的归并排序具有相同的运行时间，作为  $n$  的一个函数，借助  $\Theta$  记号， $k$  的最大值是什么？

$\Theta(nk + n\lg(\frac{n}{k})) = \Theta(nk + n\lg n - n\lg k)$ ，此时  $\lg k$  是一个常量项可以略去，所以  $\Theta(nk + n\lg(\frac{n}{k})) = \Theta(nk + n\lg n)$ ，并且当  $k = \lg(n)$  时， $\Theta(nk + n\lg n) = \Theta(n\lg n)$ ，所以  $k$  的最大值为  $\lg n$ 。

d. 在实践中，我们应该如何选择  $k$ ？

$$k = \lceil \lg n \rceil$$

**2-2** (冒泡排序的正确性) 冒泡排序是一种流行但低效的排序算法，它的作用是反复交换相邻的未按次序排列的元素。

BUBBLE-SORT( $A$ )

```

1  for  $i = 1$  to  $A.length - 1$ 
2      for  $j = A.length$  downto  $i + 1$ 
3          if  $A[j] < A[j - 1]$ 
4              Exchange  $A[j]$  with  $A[j - 1]$ 
```

显然，冒泡排序在最坏情况下的运行时间是  $\Theta(n^2)$ ，虽然插入排序的最坏情况下运行时间也是  $\Theta(n^2)$ ，但冒泡排序的内循环中交换两个元素的步骤要比插入排序中移动元素的步骤多。因此，插入排序要比冒泡排序的性能要好一些。

**2-3** (霍纳 (Horner) 规则的正确性) 给定系数  $a_0, a_1, \dots, a_n$  和  $x$  的值，代码片段：

```

1   $y = 0$ 
2  for  $i = n$  downto 0
3       $y = a_i + xy$ 
```

实现了用于求值多项式:

$$p(x) = \sum_{k=0}^n a_k x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n) \dots))$$

a. 借助  $\Theta$  符号, 实现霍纳规则的以上代码的运行时间是多少?

$\Theta(n)$

b. 编写伪代码来实现朴素的多项式求值算法, 该算法从头开始计算多项式的每个项。该算法的运行时间是多少? 与霍纳规则相比, 其性能如何?

```
1  y = a0
2  for i = 2 to n
3      h = x
4      for j = 2 to i
5          h = h * x
6      y = y + ai * h
```

上面的代码具有嵌套的 *for* 循环, 因此其算法的运行时间为  $\Theta(n^2)$ 。

**2-4 (逆序对)** 假设  $A[1 \dots n]$  是一个有  $n$  个不同数的数组, 若  $i < j$  且  $A[i] > A[j]$ , 则对偶  $(i, j)$  称为  $A$  的一个逆序对 (*inversion*)。

a. 列出数组  $(2, 3, 8, 6, 1)$  的 5 个逆序对。

$(2, 1), (3, 1), (8, 6), (6, 1), (8, 1)$

b. 由集合  $\{1, 2, 3, \dots, n\}$  中的元素构成的什么数组具有最多的逆序对? 它有多少个逆序对? 当集合中的元素按照由大到小的顺序排列时, 它具有最多的逆序对。

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

c. 插入排序中的运行时间与输入数组中逆序对的数量之间有什么关系? 证明你的回答。

$$T(n) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$$

d. 给出一个确定在  $n$  个元素的任何排列中逆序对数量的算法, 最坏情况需要  $\Theta(n \lg n)$  时间。

在归并排序算法中, 当合并  $L, R$  数组时, 两者都已按照从小到大的排好了序。我们发现在合并的 *for* 循环每次迭代时, 只要  $L[i] > R[j]$ , 那么  $L$  中剩余的其他元素都会大于  $R[j]$ , 因为  $R$  数组中的元素在原数组中总会在  $L$  数组的右边, 这就相当于找到了  $q - p + 1 - i$  个逆序对, 所以只要在每次迭代求和  $q - p + 1 - i$  就可以得知其逆序对的个数。如下伪代码所示:



MERGE( $A, p, q, r$ )

...

**for**  $k = p$  **to**  $r$

**if**  $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i + 1$

**else**

$A[k] = R[j]$

$\text{cnt} += q - p - i + 1$

$j = j + 1$

...