## 第2章 算法基础

#### 练习 2.1-1



**问题分析**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 59 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 58 | 59 |

#### 练习 2.1-2



**问题分析**

INSERTION-SORT-NON-ASCENDING(*A*)

**for** *j* = 2 🡪 *A*.*length*

*key* = *A*[*j*]

*i* = *j*-1

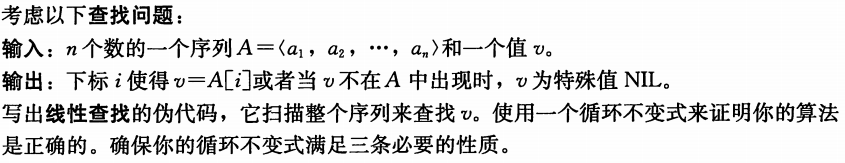
**while** *i* > 0 && *A*[*i*] < *key*

*A*[*i*+1] = *A*[*i*]

*i*--

*A*[*i*+1] = *key*

#### 练习 2.1-3



**问题分析**

LINEAR-SEARCH (A)

**for** *i* = 1 🡪 *n*

**if** *A*[*i*] == *v*

**return** *i*

**return** NIL

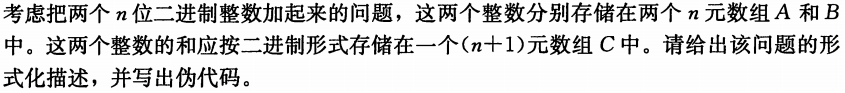
以上代码存在一个循环不变式：每次迭代开始之前，*A*[1 .. *i*-1]中没有等于*v*的元素。下面详细说明：

初始化：在第一次迭代之前，*i* = 1。*A*[1 .. 0]是一个空数组，显然没有等于*v*的元素，因此循环不变式成立。

保持：假设在第*i*-1次迭代之前循环不变式成立，所以*A*[1 .. *i*-2]中没有等于*v*的元素。在第*i*-1次迭代时，判断*A*[*i*-1] ≠ *v*，否则循环会退出。所以在第*i*次迭代之前，*A*[1 .. *i*-1]没有等于*v*的元素，故循环不变式成立。

终止：导致循环终止的条件有两个：1) *A*[*i*] = *v*，这种情况下，*A*[1 .. *i*-1]没有等于v的元素，而元素*A*[*i*]等于*v*，故在数组*A*中可以找到下标*i*使得*A*[*i*] = *v*；2) *i* = *n*+1，这种情况下，根据循环不变式，*A*[1 .. *n*]中没有等于v的元素，故*v*不在*A*中出现。

#### 练习 2.1-4



**问题分析**

该问题的形式化描述如下。

**输入**：用数组表示的两个*n*位二进制整数<*a*1, *a*2, …, *an*>和<*b*1, *b*2, …, *bn*>。

**输出**：用数组表示的一个(*n*+1)位二进制整数，表示输入的两个整数的和。

下面给出伪代码。

BINARY\_INTEGER\_ADD(*A*, *B*)

*n* = *A*.*length*

let *C* be a new array with the length of (*n*+1)

*carry* = 0

**for** *i* = 1 🡪 *n*

**if** *carry* == 0

**if** *A*[*i*] == 0 && *B*[*i*] == 0

*C*[*i*] = 0

*carry* = 0

**elseif** *A*[*i*] == 1 && *B*[*i*] == 1

*C*[*i*] = 0

*carry* = 1

**else**  // (*A*[*i*] == 0 && *B*[*i*] == 1) || (*A*[*i*] == 1 && *B*[*i*] == 0)

*C*[*i*] = 1

**else**  // *carry* == 1

**if** *A*[*i*] == 0 && *B*[*i*] == 0

*C*[*i*] = 1

*carry* = 0

**elseif** *A*[*i*] == 1 && *B*[*i*] == 1

*C*[*i*] = 1

*carry* = 1

**else**  // (*A*[*i*] == 0 && *B*[*i*] == 1) || (*A*[*i*] == 1 && *B*[*i*] == 0)

*C*[*i*] = 0

*carry* = 1

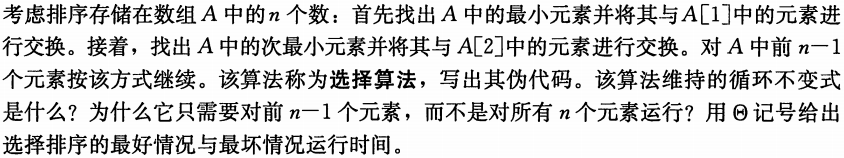
#### 练习 2.2-1



**问题分析**

Θ(*n*3)

#### 练习 2.2-2



**伪代码**

SELECTION-SORT(*A*)

**for** *i* = 1 🡪 *A*.*length*-1

*min* = *A*[*i*]

*min\_id* = *i*

**for** *j* = *i*+1 🡪 *A*.*length*

**if** *A*[*j*] < *min*

*min* = *A*[*j*]

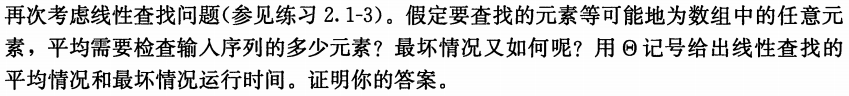
*min\_id* = *j*

**exchange** *A*[*i*] and *A*[*min\_id*]

**分析**

无论对于什么情况的输入，内层**for**循环的次数都是一样的，所以该算法没有最好情况与最坏情况之分，运行时间都为Θ(*n*2)。

#### 练习 2.2-3



**问题分析**

线性查找问题，最坏情况是要查找的元素位于数组的最后一个位置，或要查找的元素在数组中不存在，这种情况下算法需要比较数组中的每一个元素，因此运行时间为Θ(*n*)。

平均来看，算法需要比较数组中一半的元素，因此运行时间也为Θ(*n*)。

#### 练习 2.2-4



**问题分析**

针对最好情况做特殊处理。在算法中，对输入是否为最好情况做一个判断。如果输入是最好情况，那么直接输出一个事先计算好的结果；如果不是，那么再进行计算。这样做的前提是，判断最好情况的运行时间必须足够小。

#### 练习 2.3-1



**问题分析**

3

41

52

26

38

57

9

49

3 41

26 52

38 57

9 49

3 26 41 52

9 38 49 57

3 9 26 38 41 49 52 57

#### 练习 2.3-2



**伪代码**

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

*n*1 = *q* – *p* + 1

*n*2 = *r* - *q*

let *L*[1 .. *n*1] and *R*[1 .. *n*2] be new arrays

**for** *i* = 1 🡪 *n*1

*L*[*i*] = *A*[*p* + *i* - 1]

**for** j = 1 🡪 *n*2

*R*[*j*] = *A*[*q* + *j*]

*i* = *j* = 1

*k* = *p*

**while** *i* <= *n*1 && *j* <= *n*2

**if** *L*[*i*] <= *R*[*j*]

*A*[*k*++] = *L*[*i*++]

**else**

*A*[*k*++] = *R*[*j*++]

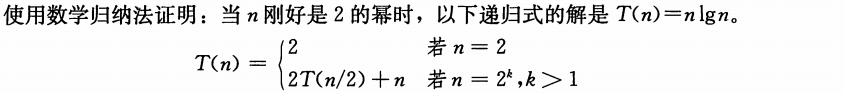
**while** *i* <= *n*1

*A*[*k*++] = *L*[*i*++]

**while** *j* <= *n*2

*A*[*k*++] = *R*[*j*++]

#### 练习 2.3-3

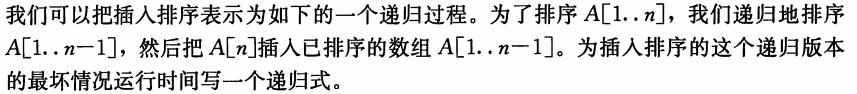


**问题分析**

初始情况*n* = 2，*T*(2) = 2 = 2 ∙ lg2，等式*T*(*n*) = *n* lg*n*成立。

现在假设等式对*n*/2成立，即*T*(*n*/2) = (*n*/2) lg(*n*/2)。根据递归式，*T*(*n*) = 2*T*(*n*/2) + *n* = 2 ∙ (*n*/2) lg(*n*/2) + *n* = *n*(lg*n* - lg2) + n = *n*(lg*n* - 1) + n = *n* lg*n*。

#### 练习 2.3-4



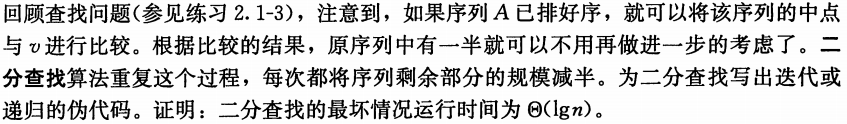
**问题分析**

排序*A*[1 .. *n*]的时间分为两部分：1) 排序*A*[1 .. *n*-1]的时间*T*(*n*-1)；2) 将*A*[*n*]插入到已排序数组*A*[1 .. *n*-1]的时间*c*(*n*)。

将*A*[*n*]插入到已排序数组*A*[1 .. *n*-1]，在最坏情况下需要做*n*-1次比较，故最坏时间为*c*(*n*) = Θ(*n*)。

将两部分时间相加得到*T*(*n*) = *T*(*n*-1) + Θ(*n*)。

#### 练习 2.3-5



**伪代码**

BINARY-SEARCH (*A*, *p*, *q*, *v*)

*m* = 

**if** *p* > *q*

**return** NIL

**else if** *A*[*m*] > *v*

BINARY-SEARCH (*A*, *p*, *m*-1, *v*)

**else if** *A*[*m*] < *v*

BINARY-SEARCH (*A*, *m*+1, *q*, *v*)

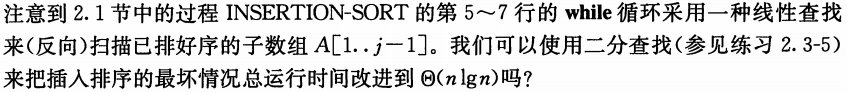
**else**

**return** *m*

**时间分析**

简单分析，对于长度为*n*的查找问题，其运行时间*T*(*n*)分为两部分：1) 计算中值*m*并将*A*[*m*]与*v*比较的时间，这部分时间为常数时间Θ(1)；2) 查找左半边或右半边的时间*T*(*n*/2)。故*T*(*n*) = *T*(*n*/2) + Θ(1)。很容易证明，*T*(*n*) = Θ(lg*n*)。

#### 练习 2.3-6



**问题分析**

用二分查找，可以在Θ(lg *j*)时间之内找到*A*[*j*]要插入的位置。但是要将*A*[*j*]插入到合适的位置*k* (1 ≤ *k* ≤ *j*)，需要将*A*[*k .. j*-1]中每个元素都向后移一个位置，在最坏情况下，这个过程仍然需要花费Θ(*j*)时间。因此，引入二分查找不能把插入排序的时间降到Θ(*n* lg *n*)。

#### 练习 2.3-7\*



**问题分析**

该问题可分以下几步来解决：

1. 先对数组*S*中的*n*个整数进行排序，得到排序后的数组*S*sorted[1 .. *n*]。采用归并排序方法，时间为Θ(*n* lg *n*)。
2. 然后，对已排序的数组*S*sorted中的每个元素*S*sorted[*i*]，计算其与*x*的差值*D*[*i*] = *x* – *S*sorted[*i*]。这一步的时间为Θ(*n*)。
3. 由于*S*sorted中元素按非降序排列，因此*D*中的元素必然按非升序排列。将*D*中的元素反向排列，使其变成非降序，得到序列*D*sorted[1 .. *n*]。这一步的时间也为Θ(*n*)。
4. 下面寻找*S*sorted和*D*sorted中是否存在相同的元素，如果存在相同的元素，说明原来的集合*S*中存在两个和刚好为*x*的元素。寻找相同元素可按以下步骤进行：① 令*i*代表*S*sorted的*index*，令*j*代表*D*sorted的*index*，初始时让*i* = 1且*j* = 1。② 进入迭代过程，每次迭代比较*S*sorted[*i*]和*D*sorted[*j*]。如果*S*sorted[*i*] == *D*sorted[*j*]，说明*S*sorted和*D*sorted中存在相同的元素，立即退出循环；如果*S*sorted[*i*] > *D*sorted[*j*]，让*j* = *j*+1，继续迭代；如果*S*sorted[*i*] < *D*sorted[*j*]，让*i* = *i*+1，继续迭代；如果*i* > *n*或*j* > *n*，说明*S*sorted和*D*sorted中不存在相同的元素，退出循环。这一步的迭代次数最大为2*n*，因此时间为Θ(*n*)。

下面用循环不变式来证明算法第4)步的正确。首先给出循环不变式：每次循环开始之前，*S*sorted[1 .. *i*-1]中没有与*D*sorted[1 .. *n*]相等的元素，并且*D*sorted[1 .. *j*-1]中也没有与*S*sorted[1 .. *n*]相等的元素。

① 初始情况：开始时，*i* = 1，*j* = 1，*S*sorted[1 .. 0]和*D*sorted[1 .. 0]都为空，循环不变式显然成立。

② 循环保持：如果*S*sorted[*i*] < *D*sorted[*j*]，说明*S*sorted[*i*] < *D*sorted[*j* .. *n*]，又由于*D*sorted[1 .. *j*-1]中也没有与*S*sorted[1 .. *n*]相等的元素，所以*S*sorted[*i*] ≠ *D*sorted[1 .. *j*-1]，故*S*sorted[*i*]与*D*sorted[1 .. *n*]中所有元素都不相等，故*S*sorted[1 .. *i*] 中没有与*D*sorted[1 .. *n*]相等的元素，让*i* = *i*+1能保证下一次迭代之前循环不变式正确。同理，如果*S*sorted[*i*] > *D*sorted[*j*]，让*j* = *j*+1同样能保证下一次迭代之前循环不变式正确。

③ 循环终止：如果*S*sorted[*i*] == *D*sorted[*j*]，循环终止，并且找到了*S*sorted和*D*sorted中存在的相等的元素。如果迭代进行到*i* = *n*+1或*j* = *n*+1时，也没有出现*S*sorted[*i*] == *D*sorted[*j*]的情况，循环也终止，说明*S*sorted和*D*sorted中没有找到相等的元素。

还需要考虑一种特殊情况：如果在*S*sorted和*D*sorted中找到相等的元素*S*sorted[*i*] == *D*sorted[*j*]，并且*S*sorted[*i*] == *D*sorted[*j*] == *x* – *S*sorted[*i*]，那么有可能*D*sorted[*j*]对应到*S*sorted中的元素*S*sorted[*k*]与*S*sorted[*i*]是同一个元素，这种情况虽然*S*sorted[*i*] + *S*sorted[*k*] == *S*，但与题目要求“两个不同整数之和为S”不符，需要排除这种情况。如果*D*sorted[*j*]对应到*S*sorted中的元素*S*sorted[*k*]与*S*sorted[*i*]是同一个元素，由于*D*sorted是由*x*减去*S*sorted中的每一个元素后反序得到的，如果*D*sorted[*j*]必定是*D*sorted中的倒数第*i*个元素，对应于*S*sorted中的正数第*i*个元素。这时*j* = *n*-*i*+1，所以有*i* + *j* = *i*+*n*-*i*+1 = *n*+1。因此如果迭代过程中遇到*S*sorted[*i*] == *D*sorted[*j*]并且*S*sorted[*i*] == *D*sorted[*j*] == *x* – *S*sorted[*i*]，那么再检查*i* + *j*是否等于*n*+1。如果相等，则继续迭代；如果不等，说明找到符合题目要求的两个元素，迭代停止。

实际上，循环不必进行到*i* = *n*+1或*j* = *n*+1才终止，进行到*i* + *j* >= *n*+1就可以终止了。根据循环不变式：每次循环开始之前，*S*sorted[1 .. *i*-1]中没有与*D*sorted[1 .. *n*]相等的元素，并且*D*sorted[1 .. *j*-1]中也没有与*S*sorted[1 .. *n*]相等的元素。*S*sorted[1 .. *i*-1]中没有与*D*sorted[1 .. *n*]相等的元素意味着和为*x*的两个整数不会出现在*S*sorted[1 .. *i*-1]中。而*D*sorted[1 .. *j*-1]实际上对应于*S*sorted[*n*-*j*+2 .. *n*]，*D*sorted[1 .. *j*-1]中也没有与*S*sorted[1 .. *n*]相等的元素意味着和为*x*的两个整数不会出现在*S*sorted[*n*-*j*+2 .. *n*]中。因此，和为*x*的两个整数只可能出现在*S*sorted[*i* .. *n*-*j*+1]中。如果*i* + *j* == *n*+1，那么*S*sorted[*i* .. *n*-*j*+1]包含的元素个数为*n*-*j*+1-*i*+1 = 1，如果可选元素只有1个，显然不可能找到两个不同整数之和为*x*。循环进行到*i* + *j* >= *n*+1就终止，刚好排除了上面所描述的特殊情况，因此循环过程中不必对这种特殊情况做处理。

将以上4步的时间加起来，得到该算法的运行时间为Θ(*n* lg *n*)。

以上算法可以得到正确结果，然而没有必要这么麻烦。可以采用下面的简化过程。

1. 先对*n*个整数进行排序，采用归并排序方法，时间为Θ(*n* lg *n*)。
2. 进入循环：① 初始时让*i* = 1且*j* = *n*。② 进入迭代过程，每次迭代将*S*sorted[*i*]和*S*sorted[*j*]相加，如果*S*sorted[*i*] + *S*sorted[*j*] < *x*，让*i* = *i*+1，继续迭代；如果*S*sorted[*i*] + *S*sorted[*j*] > *x*，让*j* = *j*-1，继续迭代；如果*S*sorted[*i*] + *S*sorted[*j*] == *x*并且*i* < *j*，能够找到“两个不同整数之和为*x*”，退出循环。以上的循环过程中始终假设*i* < *j*，如果迭代进行到*i* >= *j*，说明*S*sorted中不可能找到“两个不同整数之和为*x*”，也退出循环。

可以证明，这个简化的算法与前面那个算法是等价的，并且其运行时间依然为Θ(*n* lg *n*)。

**伪代码**

FIND\_SUM (*S*, *x*)

*n* = *S*.*length*

**if** *n* < 2

**return** NIL

MERGE\_SORT(*S*, 1, *n*)

*i* = 1

*j* = *n*

**while** *i* < *j*

**if** *S*[*i*] + *S*[*j*] > *x*

*j*--

**else if** *S*[*i*] + *S*[*j*] < *x*

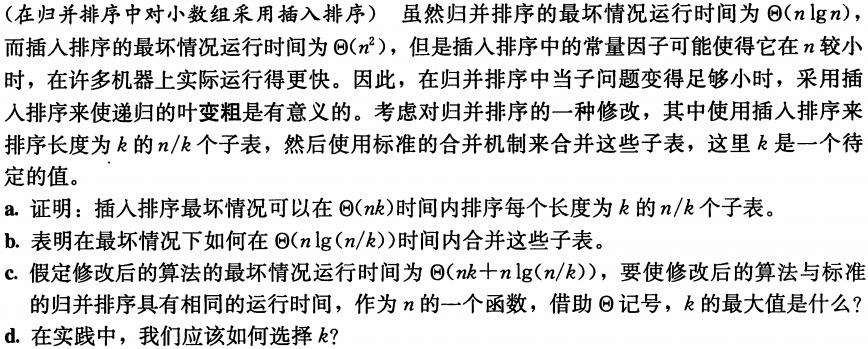
*i*++

**else**

**return** *i* and *j*

**return** NIL

#### 思考题 2-1



**a-问题分析**

对于单个长度为k的子表，插入排序的时间为Θ(*k*2)。一共有*n*/*k*个子表，总的插入排序时间为(*n*/*k*)Θ(*k*2) = Θ(*nk*)。

**b-问题分析**

类似归并排序算法，要合并*n*/*k*个子表，一共需要进行Θ(lg(*n*/*k*))层合并操作，每层合并操作的时间为Θ(*n*)。因此，合并过程的总时间为Θ(*n*lg(*n*/*k*))。

**c-问题分析**

使两种算法具有相同的运行时间，即Θ(*nk* + *n*lg(*n*/*k*)) = Θ(*n*lg*n*)。两边展开，可得

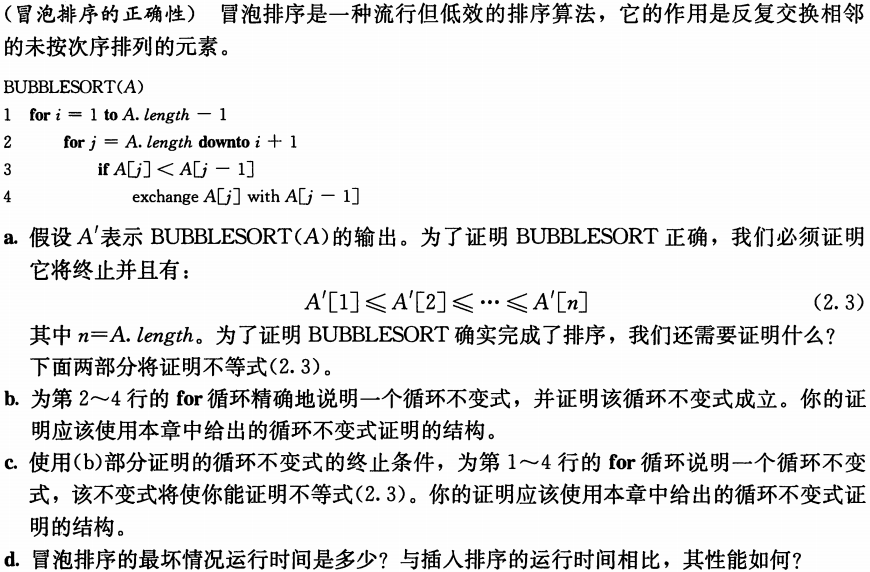
Θ(*nk* + *n*lg*n* – *n*lg*k*)) = Θ(*n*lg*n*)

于是得到*k* = lg*k*。由于*k* ≤ *n*，所以k的最大值为Θ(lg *n*)。

**d-问题分析**

现实中，*k*应当为使插入排序比归并排序快的最大值，一般可通过实验确定。

#### 思考题 2-2



**a-问题分析**

还必须证明*A’*中的元素与*A*中的元素是一一对应的，即*A’*包含了*A*中的所元素，并且不包含*A*以外的元素。

**b-问题分析**

略。

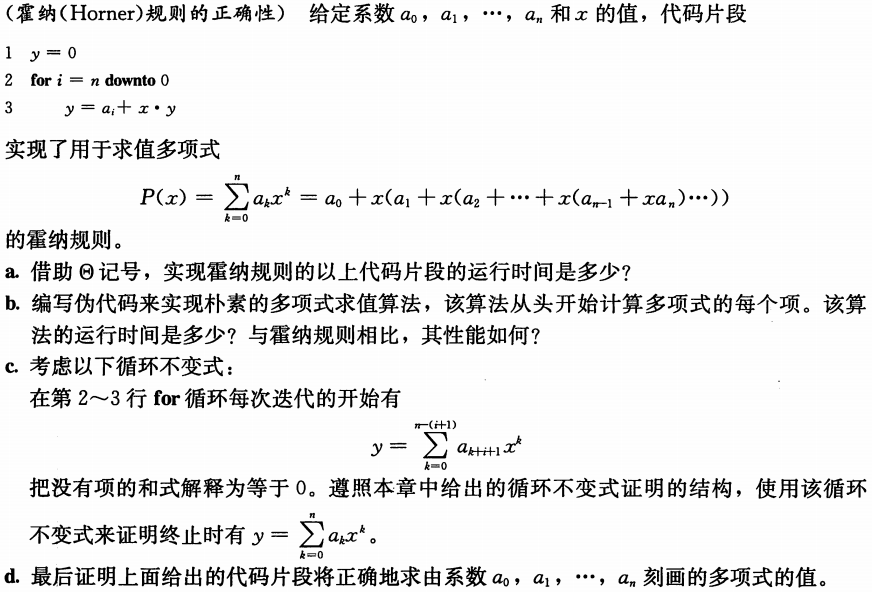
**c-问题分析**

略。

**d-问题分析**

冒泡排序对所有情况的输入其运行时间都为Θ(*n*2)。

#### 思考题 2-3



**a-问题分析**

在霍纳规则代码中，一共有Θ(*n*)次迭代，每次迭代花费常数时间Θ(1)，因此总时间为Θ(*n*)。

**b-问题分析**

朴素的多项式求值算法的伪代码如下：

*y* = 0

for *i* = 0 🡪 *n*

*t* = 1

for *j* = 1 🡪 *i*

*t* \*= *x*

*y* += (*ai* \* *t*)

该算法的运行时间为Θ(*n*2)。

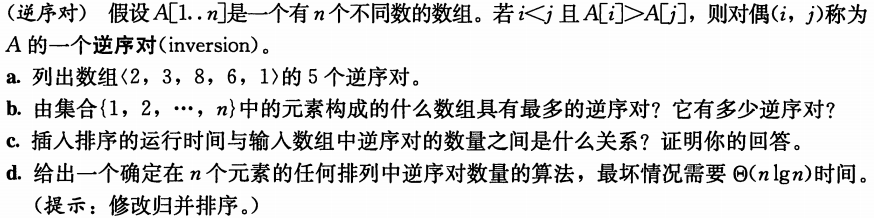
**c-问题分析**

略

**d-问题分析**

略

#### 思考题 2-4



**a-问题分析**

5个逆序对：(8, 6) (6, 1) (8, 1) (3, 1) (2, 1)

**b-问题分析**

逆序对最多的情况出现在元素按降序排列的时候，即{*n*, *n*-1, … , 2, 1}。这种情况，任意两个元素都构成逆序对，一共有组逆序对。

**c-问题分析**

插入排序的运行时间*T*(*n*)与数组中的逆序对数量*R*(*n*)成线性关系*T*(*n*) = Θ(*R*(*n*))。下面用数学归纳法证明。

初始情况*n* = 2，线性关系显然成立。

假设对于*n*-1个元素的情况，插入排序的运行时间*T*(*n*-1)与逆序对的数量*R*(*n*-1)成线性关系*T*(*n*-1) = Θ(*R*(*n*-1))。对于*n*个元素的情况，其逆序对数目*R*(*n*)分为两部分：①前n-1个元素的逆序对数目*R*(*n*-1)；②前n-1个元素与第*n*个元素构成的逆序对数目*Rp*(*n*)。故*R*(*n*) = *R*(*n*-1) + *Rp*(*n*)。*n*个元素的插入排序可分为两步进行：①首先将前*n*-1个元素排序，其时间*T*(*n*-1) = Θ(*R*(*n*-1))；②然后将第*n*个元素插入到已排序好的前*n*-1个元素中，这一步的时间*Tp*(*n*) = Θ(*Rp*(*n*))。故排序*n*个元素的运行时间*T*(*n*) = *T*(*n*-1) + *Tp*(*n*) = Θ(*R*(*n*-1)) + Θ(*Rp*(*n*)) = Θ(*R*(*n*-1) + *Rp*(*n*)) = Θ(*R*(*n*))。

综上所述，插入排序的运行时间*T*(*n*)与数组中的逆序对数量*R*(*n*)成线性关系*T*(*n*) = Θ(*R*(*n*))。

**d-问题分析**

考虑分治法求解。对于*A*[1 .. *n*]，将其对半分为*AL* = *A*[1 .. *n*/2]和*AR* = *A*[*n*/2+1 .. *n*]。*A*[1 .. *n*]的逆序对数目*R*(*A*)分为3部分：

① 左半边*AL*的逆序对数目*R*(*AL*)

② 右半边*AR*的逆序对数目*R*(*AR*)

③ 左半边元素与右半边元素构与的逆序对数目*R*(*AL, AR*)。

故*R*(*A*) = *R*(*AL*) + *R*(*AR*) + *R*(*AL, AR*)。

①和②实际上是原问题的子问题，可以递归地解决。

下面考虑③。一种简单的方法是，对*AL*中的每个元素*AL*[*i*]，遍历*AR*中的每个元素*AR*[*j*]，寻找与*AL*[*i*]构成逆序对的元素。这种方法的时间为(*n*/2)2 = Θ(*n*2)，时间成本太高。然而，如果*AL*和*AR*分别已经排好序，可以把时间缩减到Θ(*n*)。并且，分别排序*AL*和*AR*并不影响*AL*和*AR*之间的逆序对数目*R*(*AL, AR*)。具体方法如下：

下面的伪代码假设数组*A*分为两部分*A*[*p* .. *m*]和*A*[*m*+1 .. *q*]。

CROSS\_REVERSE\_NUM(*A*, *p*, *m*, *q*)

*nL* = *m* – *p* + 1

*nR* = *q* – *m*

*i = p*

*j = m*+1

*num* = 0

**while** *i* <= *m* && *j* <= *q*

**if** *A*[*i*] > *A*[*j*] // 说明*A*[*i* .. *m*]都比*A*[*j*]大，*A*[*i* .. *m*]与*A*[*j*]都构成逆序对

*num* += (*m* - *i* + 1)

*j*++

**else** // 说明*A*[*p* .. *i*]都没有*A*[*j*]大，不构成逆序对，因此考察下一个元素*A*[*i*+1]

*i*++

**return** *num*

上述方法可以和归并排序揉合在一起，先执行CROSS\_REVERSE\_NUM(*A*, *p*, *m*, *q*)，然后执行归并排序。在递归的每一层，执行CROSS\_REVERSE\_NUM(*A*, *p*, *m*, *q*)和归并排序各花费Θ(*n*)时间。递归一共有Θ(lg*n*)层，因此该算法的执行时间为Θ(*n*lg*n*)。

**d-伪代码**

REVERSE\_NUM(*A*, *p*, *r*)

*num* = 0

**if** *p* < *r*

*m* = 

*num* += REVERSE\_NUM(*A*, *p*, *m*)

*num* += REVERSE\_NUM(*A*, *m*+1, *q*)

*num* += CROSS\_REVERSE\_NUM(*A*, *p*, *m*, *q*)

MERGE(*A*, *p*, *m*, *q*)

**return** *num*