第二章 算法入门

由于时间问题有些问题没有写的很仔细，而且估计这里会存在不少不恰当之处。另，思考题 2-3 关于霍纳规则，有些部分没有完成，故没把解答写上去，我对其 c 问题有疑问，请有解答方法者提供个意见。

给出的代码目前也仅仅为解决问题，没有做优化，请见谅，等有时间了我再好好修改。

插入排序算法伪代码

```
INSERTION-SORT(A)
1   for j ← 2 to length[A]
2       do 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              do A[i+1]←A[i]
7                 i ← i − 1
8          A[i+1]←key
```

C#对插入排序算法的实现:

```
public static void InsertionSort<T>(T[] Input) where T:IComparable<T>
 {
     T key;
     int i;
     for (int j = 1; j < Input.Length; j++)
     {
         key = Input[j];
         i = j - 1;
         for (; i >= 0 && Input[i].CompareTo(key)>0;i-- )
             Input[i + 1] = Input[i];
         Input[i+1]=key;
     }
 }
```

插入算法的设计使用的是增量（incremental）方法：在排好子数组A[1..j-1]后，将元素A[j]插入，形成排好序的子数组A[1..j]

这里需要注意的是由于大部分编程语言的数组都是从0开始算起，这个与伪代码认为的数组的数是第1个有所不同，一般要注意有几个关键值要比伪代码的小1.

如果按照大部分计算机编程语言的思路，修改为：

```
INSERTION-SORT(A)
1   for j ← 1 to length[A]
2       do key ←A[j]
3          i ←j-1
```

```
4        while i ≥ 0 and A[i] > key
5            do A[i+1]←A[i]
6                i ← i − 1
7        A[i+1]←key
```

循环不变式(Loop Invariant)是证明算法正确性的一个重要工具。对于循环不变式，必须证明它的三个性质：

初始化(Initialization)：它在循环的第一轮迭代开始之前，应该是正确的。

保持(Maintenance)：如果在循环的某一次迭代开始之前它是正确的，那么，在下一次迭代开始之前，它也是正确的。

终止(Termination)：当循环结束时，不变式给了我们一个有用的性质，它有助于表明算法是正确的。

运用循环不变式对插入排序算法的正确性进行证明：

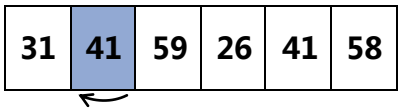
初始化：j=2，子数组 A[1..j-1]只包含一个元素 A[1],显然它是已排序的。

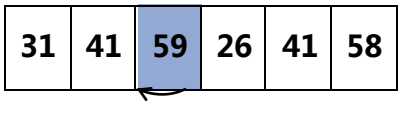
保持 若A[1..j-1]是已排序的 则按照大小确定了插入元素A[j]位置之后的数组A[1..j]显然也是已排序的。

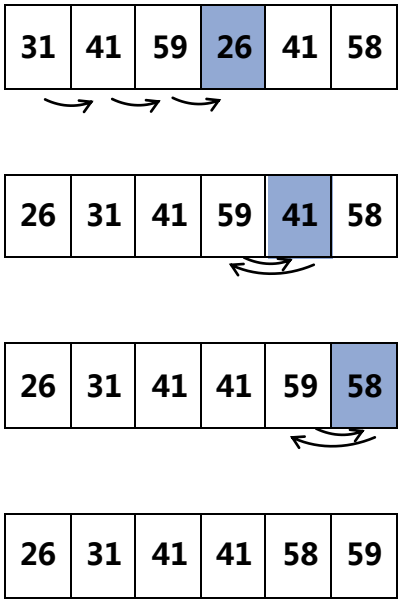
终止：当 j=n+1 时，退出循环，此时已排序的数组是由 A[1],A[2],A[3]…A[n]组成的A[1..n]，此即原始数组 A。

练习

2.1-1：以图 2-2 为模型，说明 INSERTION-SORT 在数组 A=<31,41,59,26,41,58>上的执行过程。







2.1-2：重写过程 INSERTION-SORT，使之按非升序（而不是按非降序）排序。

```
INSERTION-SORT(A)
1   for j ← 2 to length[A]
2       do 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              do A[i+1]←A[i]
7                 i ← i − 1
7           A[i+1]←key
```

2.1-3：考虑下面的查找问题：

输入：一列数 A=<a_1,a_2,...,a_n>和一个值 v

输出：下标 i，使得 v=A[i]，或者当 v 不在 A 中出现时为 NIL。

写出针对这个问题的现行查找的伪代码，它顺序地扫描整个序列以查找 v。利用循环不变式证明算法的正确性。确保所给出的循环不变式满足三个必要的性质。

```
LINEAR-SEARCH(A,v)

1   for i ← 1 to length[A]

2       if v=A[i]

3          return i
```

4 return NIL

现行查找算法正确性的证明。

初始化： i=1，子数组为 A[1..i]，只有一个元素 A[1],如果 v=A[1]就返回 1,否则返回 NIL，算法显然是正确的。

保持：若算法对数组 A[1..i]正确，则在数组增加一个元素 A[i+1]时，只需要多作一次比较，因此显然对 A[1..i+1]也正确。

终止：算法如果在非最坏情况下定能返回一个值此时查找成功，如果 n 次查找（遍历了所有的数）都没有成功，则返回 NIL。算法在有限次查找后肯定能够给出一个返回值，要么说明查找成功并给出下标，要么说明无此值。因此算法正确。

该算法用 C#实现的代码：

```
public static int LinearSearch<T>(T[] Input, T v) where T:IComparable<T>
{
    for (int i = 0; i < Input.Length;i++ )
        if (Input[i].Equals(v))
            return i;
    return -1;
}
```

2.1-4：有两个各存放在数组 A 和 B 中的 n 位二进制整数，考虑它们的相加问题。两个整数的和以二进制形式存放在具有(n+1)个元素的数组 C 中。请给出这个问题的形式化描述，并写出伪代码。

A 存放了一个二进制 n 位整数的各位数值，B 存放了另一个同样是二进制 n 位整数的各位上的数值，现在通过二进制的加法对这两个数进行计算，结果以二进制形式把各位上的数值存放在数组 C（n+1 位）中。

BINARY-ADD(A,B,C)

1 flag← 0

2 **for** j ← 1 **to** n

```
3       do key←A[j]+B[j]+flag
4       C[j]←key mod 2
5        if key>1
6            flag←1
7    if flag=1
8       C[n+1]←1
```

1.**RAM**(Random-Access Machine)模型分析通常能够很好地预测实际计算机上的性能，RAM 计算模型中，指令一条接一条地执行，没有并发操作。RAM 模型中包含了真实计算机中常见的指令：算术指令（加法、剑法、乘法、出发、取余、向下取整、向上取整指令）、数据移动指令（装入、存储、复制指令）和控制指令（条件和非条件转移、子程序调用和返回指令）。其中每天指令所需时间都为常量。

RAM 模型中的数据类型有整数类型和浮点实数类型。

2.算法的运行时间是指在特定输入时，所执行的基本操作数（或步数）。

插入算法的分析比较简单，但是不是很有用，所以略过。（在解思考题 2-1 时有具体的实例分析，请参看）

3.一般考察算法的最坏情况运行时间。这样做的理由有三点：

A．一个算法的最坏情况运行时间是在任何输入下运行时间的一个上界。

B．对于某些算法，最坏情况出现的是相当频繁的。

C．大致上来看，“平均情况“通常与最坏情况一样差。

4.如果一个算法的最坏情况运行时间要比另一个算法的低，我们常常就认为它的效率更高。

练习

2.2-1：用 Θ 形式表示表示函数$n^3/1000-100n^2-100n+3$

$$\Theta(n^3)$$

2.2-2:考虑对数组 A 中的 n 个数进行排序的问题 :首先找出 A 中的最小元素 ,并将其与 A[1]中的元素进行交换。接着，找出 A 中的次最小元素，并将其与 A[2]中的元素进行交换。对 A 中头 n-1 个元素继续这一过程。写出这个算法的伪代码 ,该算法称为选择排序(selection sort)。对这个算法来说，循环不变式是什么?为什么它仅需要在头 n-1 个元素上运行，而不是在所有 n 个元素上运行？以Θ形式写出选择排序的最佳和最坏情况下的运行时间。

假设函数 MIN(A,i,n)从子数组 A[i..n]中找出最小值并返回最小值的下标。

SELECTION-SORT(A)

1 **for** i←1 **to** n-1

2 j←MIN(A,i,n)

3 exchange A[i]↔A[j]

选择排序算法正确性的证明

初始化：i=1，从子数组 A[1..n]里找到最小值 A[j]，并与 A[i]互换，此时子数组 A[1..i]只有一个元素 A[1]，显然是已排序的。

保持：若 A[1..i]是已排序子数组。这里显然 A[1]≤A[2]≤A[3]≤…≤A[i]，而 A[i+1..n]里最小值也必大于 A[i] ,找出此最小值与 A[i+1]互换并将 A[i+1]插入 A[1..i]得到子数组 A[1..i+1]。A[1..i+1]显然也是已排序的。

终止：当 i=n 时终止，此时已得到已排序数组 A[1..n-1]，而 A[n]是经过 n-1 次比较后剩下的元素，因此 A[n]大于 A[1..n-1]中任意元素，故数组 A[1..n]也即是原数组此时已是已排序的。所以，算法正确。

仅需要在头 n-1 个元素上运行是因为经过 n-1 次比较后剩下的是最大元素 ,其理应排在最后一个位置上，因此可以不必对此元素进行交换位置操作。

由于 MIN()函数和 SWAP()函数对于任意情况运行时间都相等，故这里最佳和最坏情况下运行时间是一样的。

$$n+\sum_{i=1}^{n-1}(n-i)+\Theta(n)=\Theta(n^2)$$

选择算法的的 C#实现：

```
private static int Min<T>(T[] Input,int start,int end) where T:IComparable<T>
{
    int flag=start;
    for (int i = start; i < end; i++)
        if (Input[flag].CompareTo(Input[i]) > 0)
            flag = i;
    return flag;
}
private static void Swap<T>(ref T a,ref T b) where T : IComparable<T>
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
public static T[] SelectionSort<T>(T[] Input) where T:IComparable<T>
{
    for (int i = 0; i < Input.Length - 1; i++)
        Swap(ref Input[Min(Input, i, Input.Length)],ref Input[i]);
    return Input;
}
```

2.2-3：再次考虑线性查找问题（见练习 2.1-3）。在平均情况下，需要检查输入序列中的多少个元素？假定查找的元素是数组中任何一个元素的可能性都是相等的。在最坏情况下又怎么样呢？用Θ相似表示的话，线性查找的平均情况和最坏情况运行时间怎么样？对你的答案加以说明。

平均：n/2 次。因为任意一个元素大于、小于查找数的概率一样。

最坏：n 次。最后一个元素才是要查找的元素。

用Θ表示都是：**Θ(n)**

2.2-4：应如何修改一个算法，才能使之具有较好的最佳情况运行时间？

要使算法具有较好的最佳情况运行时间就一定要对输入进行控制，使之偏向能够使得算法具有最佳运行情况的排列。

5.分治法（divide-and-conquer）:有很多算法在结构上是递归的：为了解决一个给定的问题，算法要一次或多次地递归调用其自身来解决相关的问题。这些算法通常采用分治策略：将原问题划分成 n 个规模较小而结构与原问题相似的子问题；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

容易确定运行时间，是分治算法的有点之一。

6.分治模式在每一层递归上都有三个步骤：

分解（Divide）: 将原问题分解成一系列子问题；

解决（Conquer）: 递归地解各子问题。若子问题足够小，则直接求解；

合并（Combine）: 将子问题的结果合并成原问题的解。

7.合并排序（Merge Sort）算法完全依照了分治模式。

分解：将 n 个元素分成各含 n/2 个元素的子序列；

解决：用合并排序法对两个子序列递归地排序；

合并：合并两个已排序的子序列以得到排序结果。

在对子序列排序时，其长度为 1 时递归结束。单个元素被视为是已排好序的。

合并排序的关键步骤在于合并步骤中的合并两个已排序子序列。为做合并，引入一个辅助过程 MERGE(A,p,q,r)，其中 A 是个数组，p、q 和 r 是下标，满足$p \le q < r$。该过程假设子数组 A[p..q]和 A[q+1..r]都已排好序，并将他们合并成一个已排好序的子数组代替当前子数组 A[p..r]。

MERGE 过程的时间代价为**Θ(n)**，其中 n=r-p+1 是待合并的元素个数。

MERGE 过程:

MERGE(A,p,q,r)

1 $n_1 \leftarrow q - p + 1$

2 $n_2 \leftarrow r - q$

3 ▷create arrays L[1..$n_1 + 1$] and R[1..$n_2 + 1$]

4 **for** $i \leftarrow 1$ **to** n_1

5 **do** L[i]←A[p+i-1]

6 **for** j←1 **to** n_2

7 **do** R[j]←A[q+j]

8 L[$n_1 + 1$]← ∞

9 R[$n_2 + 1$]← ∞

10 $i \leftarrow 1$

11 $j \leftarrow 1$

12 for $k \leftarrow p$ **to** r

13 do if L[i]≤R[j]

14 then A[k]←L[i]

15 $i \leftarrow i + 1$

16 else A[k]←R[j]

17 $j \leftarrow j + 1$

MERGE 过程正确性的证明

初始化:第一轮循环,k=p,i=1,j=1,已排序数组 L、R,比较两数组中最小元素 L[i]、R[j],取较小的置于 A[p],此时子数组 A[p..p]不仅是已排序的(仅有一个元素),而且是所有待排

序元素中最小的。若最小元素是 L[i]，取 i=i+1，即 i 指向 L 中未排入 A 的所有元素中最小的一个；同理，j 之于 R 数组也是如此。

保持：若 A[p..k]是已排序的，由计算方法知，L 中 i 所指、R 中 j 所指及其后任意元素均大于等于 A[p..k]中最大元素 A[k],当 k=k+1，A[k+1]中存入的是 L[i]、R[j]中较小的一个，但是仍有 A[k]≤A[k+1]，而此时，子数组 A[p..k+1]也必是有序的，i、j 仍是分别指向 L、R 中未排入 A 的所有元素中最小的一个。

终止： k=r+1 时终止跳出循环，此时，A[p..r]是已排序的，且显有 A[p]≤A[p+1]≤..≤A[r]。此即原待排序子数组，故算法正确。

MERGE-SORT(A,p,r)

1 **if** p<r

2 **then** q←⌊(p+r)/2⌋

3 MERGE-SORT(A,p,r)

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

5 MERGE-SORT(A,p,q,r)

算法与二叉树的后序遍历算法（先左子树，然后右子树，最后根）相似。

（第三行、第四行顺序可以互换）

合并排序算法的 C#实现代码：

```
public static void MergeSort<T>(T[] Input,int p,int r) where T:IComparable<T>
{
    int q;
    if (p < r)
    {
        q = (p + r) / 2;
        MergeSort(Input, p, q);
        MergeSort(Input, q + 1, r);
        Merge(Input,p,q,r);
    }
```

```
}
private static void Merge<T>(T[] Input,int p,int q,int r) where T:IComparable<T>
{
    int n1 = q - p + 1;
    int n2 = r - q;
    T[] L = new T[n1];
    T[] R = new T[n2];
    for (int i = 0; i < n1; i++)
        L[i] = Input[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = Input[q + 1 + j];

    for (int i = 0, j = 0, k = p; k <= r; k++)
    {
        if(i<n1&&j<n2)
            if (L[i].CompareTo(R[j]) < 0||L[i].Equals(R[j]))
            {
                Input[k] = L[i];
                ++i;
                continue;
            }
            else
            {
                Input[k] = R[j];
                ++j;
                continue;
            }
        if (i >= n1 && j < n2)
        {
            Input[k] = R[j];
            ++j;
            continue;
        }
        if (i < n1 && j >= n2)
        {
            Input[k] = L[i];
            ++i;
            continue;
        }
    }
}
```

8.当一个算法中含有对其自身的递归调用时，其运行时间可以用一个递归方程（或递归式）来表示。

合并算法的递归式：

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

$D(n)$是分解该问题所用时间，$C(n)$是合并解的时间；对于合并排序算法，a 和 b 都是 2

T(n)在最坏的情况下合并排序 n 个数的运行时间分析:

当 n>1 时，将运行时间如下分解：

分解：这一步仅仅算出子数组的中间位置，需要常量时间，因而$\mathbf{D(n)=\Theta(1)}$

解决：递归地解为两个规模为 n/2 的子问题，时间为$\mathbf{2T(n/2)}$

合并：含有 n 个元素的子数组上，MERGE 过程的运行时间为$\mathbf{C(n)=\Theta(n)}$

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

将上式改写：

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

在所构造的递归树中，顶层总代价为$\mathbf{cn}$（n 个点的集合）。往下每层总代价不变，第 i 层的任一节点代价为$\mathbf{c(n/2^i)}$（共$\mathbf{2^i}$个节点总代价仍然是$\mathbf{cn}$）。最底层有 n 个节点（$n\times 1$），每个点代价为 c。此树共有$\mathbf{\lg n+1}$层，深度为$\mathbf{\lg n}$。

因此 n 层的总代价为：

$$\mathbf{cn\times(\lg n+1)=cn\lg n+cn=\Theta(n\lg n)}$$

练习

2.3-1：2-4 为模型，说明合并排序在输入数组 A=<3,41,52,26,38,57,9,49>上的执行过程。

以文字代替图示

1.(3)(41)→(3,41);(52)(26) →(26,52);(38)(57) →(38,57);(9)(49) →(9,49)

2.(3,41)(26,52) →(3,26,41,52);(38,57)(9,49) →(9,38,49,57)

3.(3,26,41,52)(9,38,49,57) →(3,9,26,38,41,49,52,57)

2.3-2：MERGE 过程，使之不适用哨兵元素，而是在一旦数组 L 或 R 中的所有元素都被复制回数组 A 后，就立即停止，再将另一个数组中余下的元素复制回数组 A 中

MERGE(A,p,q,r)

1 $n_1 \leftarrow q - p + 1$

2 $n_2 \leftarrow r - q$

3 ▷create arrays L[1..n_1] and R[1..n_2]

4 **for** $i \leftarrow 1$ **to** n_1

5 **do** L[i]←A[p+i-1]

6 **for** j←1 **to** n_2

7 **do** R[j]←A[q+j]

8 $i \leftarrow 1$

9 $j \leftarrow 1$

10 **for** $k \leftarrow p$ **to** r

11 **do if** $i < n_1$ **and** $j < n_2$

12 **if** L[i]≤R[j]

13 A[k]←L[i]

14 $i \leftarrow i + 1$

15 **continue**

16 **else** A[k]←R[j]

17 j←j+1

18 **continue**

19 **do if** i≥ n_1 **and** j< n_2

20 A[k]←R[j]

21 j←j+1

22 **continue**

23 **do if** i< n_1 **and** j≥ n_2

24 A[k]←L[i]

25 i ← i + 1

26 **continue**

2.3-3：利用数学归纳法证明：当 n 是 2 的整数次幂时，递归式

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

的解为$T(n) = n \lg n$。

1° $n = 2$（可看做$k = 1$）时，$T(n) = 2 = 2 \lg 2^1$

$n = 2^2$时，$k = 2$，$T(n) = 2T(2^2/2) + 2^2 = 2T(2) + 4 = 2 \times 2 + 4 = 8 = 2^2 \lg 2^2$

2° 当$k \geqslant 2$，$n = 2^k$时

$$T(2^k) = 2T(2^{k-1} + 2^k = 2^k \lg 2^k = k2^k)$$

则当$k = k + 1$，即$n = 2^{k+1}$时：

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2^{k+1} \\ &= 2 \cdot k \cdot 2^k + 2^{k+1} \\ &= (k+1)2^{k+1} \\ &= (k+1) \lg 2^{k+1} \end{aligned}$$

故当$k \geqslant 1$，即 n 是 2 的整数倍幂时均有$T(n) = n \lg n$

2.3-4：插入排序可以如下改写成一个递归过程：为排序 A[1..n]，先递归地排序 A[1..n-1]，然后再将 A[n]插入到已排序的数组 A[1..n-1]中去。对于插入排序的这一递归版本，为它的

运行时间写一个递归式。

首先是 INSERTION 过程

INSERTION (A,p,r)

```
1    for j ← p to r
2       do key ←A[j]
3            i ←j-1
4            while i > 0 and A[i] > key
5                 do A[i+1]←A[i]
6                      i ← i − 1
7            A[i+1]←key
```

插入排序的递归调用算法：

RECURSION-INSERTION-SORT(A,p,r)

```
1    if p<r
2        r←r-1
3        RECURSION-INSERTION-SORT(A,p,r)
4        INSERTION(A,p,r)
```

该算法的 C#实现代码：

```
public static void RecursionInsertionSort<T>(T[] Input,int p,int r) where T:IComparable<T>
{
    if (p < r)
    {
        --r;
        RecursionInsertionSort(Input, p, r);
        Insertion(Input,p,r);
    }
}
private static void Insertion<T>(T[] Input, int p, int r) where T : IComparable<T>
{
    T key;
    int i;
    for (int j = 1; j < r; j++)
    {
        key = Input[j];
        i = j - 1;
        for (; i >= 0 && Input[i].CompareTo(key) > 0; i--)
```

```
            Input[i + 1] = Input[i];
        Input[i + 1] = key;
    }
}
```

$$T(n) = \begin{cases} \Theta(1) & n \leqslant c \\ \frac{n-1}{n}T(n-1) + \Theta(n^2) \end{cases}$$

2.3-5：回顾一下练习 2.1-3 中提出的查找问题，注意如果序列 A 是已排序的，就可以将该序列的中点与 v 进行比较。根据比较的结果，原序列中有一半就可以不用再做进一步的考虑了。二分查找（binary search）就是一个不断重复这一查找过程的算法，它每次都将序列余下的部分分成两半，并只对其中的一半做进一步的查找。写出二分查找算法的伪代码，可以是迭代的，也可以是递归的。说明二分查找的最坏情况运行时间为什么是$\Theta(\lg n)$。

使用递归，先确定一个过程 BINARY(A,p,r,v)

BINARY(A,p,r,v)

1 **for** j←p **to** r

2 **if** A[j]=v

3 **return** j

4 **return NIL**

然后是二分查找的递归过程

BINARY-SEARCH(A,p,r,v)

1 **if** p=0 and r=0 and A[0]=v

2 return 0

3 **if** p<r

4 $q \leftarrow \lfloor (p+r)/2 \rfloor$

5 **if** A[q]>v

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

7 **return** BINARY(A,p,q,v)

8 **else** BINARY-SEARCH(A,q+1,r,v)

9 **return** BINARY(A,q+1,r,v)

10 **return NIL**

该算法的 C#实现代码：

```
public static int BinarySearch<T>(T[] Input,int p,int r,T v) where T:IComparable<T>
{
    int q;
    if (p == 0 && r == 0 && Input[0].Equals(v))
        return 0;
    if (p < r)
    {
        q = (p + r) / 2;
        if (Input[q].CompareTo(v) > 0 )
        {
            BinarySearch(Input, p, q, v);
            return Binary(Input, p, q, v);
        }

        else
        {
            BinarySearch(Input, q + 1, r, v);
            return Binary(Input, q+1, r, v);
        }
    }
    return -1;
}
private static int Binary<T>(T[] Input, int p, int r, T v) where T:IComparable<T>
{
    for (int j = p; j <= r; j++)
        if (Input[j].Equals(v))
            return j;
    return -1;
}
```

由公式$N = a^{\log_a N}$得：

$$n \times \frac{1}{2^{\lg n}} = 1$$

因经过 n 次的与中点比较后肯定能找到最后一个点（最坏情况了），如果是返回下标，否

则返回 NIL，故最坏情况下时间复杂度为$\Theta(\lg n)$

2.3-6 :观察一下 2.1 节中给出的 INSERTION-SORT 过程 ,在第 5~7 行的 while 循环中，采用了一种线性查找策略，在已排序的子数组 A[1..j-1]中（反向）扫描。是否可以改为二分查找策略（见练习 2.3-5），来将插入排序的总体最坏情况运行时间改善至$\Theta(n \lg n)$？

首先引入一个二分查找策略（与 2.3-5 的 Binary Search 略有不同）

```
BINARY(A,p,r,v)
5     for j←p to r
6         if A[j]>v
7             return j
8     return NIL
```

然后是二分查找的递归过程

```
BINARY-SEARCH(A,p,r,v)
10    if p=0 and r=0 and A[0]>v
11        return 0
12    if p<r
13        q ← ⌊(p+r)/2⌋
14        if A[q]>v
15            BINARY-SEARCH(A,p,q,v)
16            return BINARY(A,p,q,v)
17        else BINARY-SEARCH(A,q+1,r,v)
18             return BINARY(A,q+1,r,v)
10     return NIL
```

利用了二分查找策略的插入排序：

BINARYINSERTION-SORT(A)

```
1    for j←2 to length[A]
2        do key←A[j]
3        i←j-1
4        k←BINARY-SEARCH(A,0,i,key)
5        if k!= NIL
6            for s←i downto k
7                A[s+1]←A[s]
8            A[k]←key
```

此算法的在最坏情况下的运行时间是$\Theta(n \lg n)$

该算法的 C#实现代码：

```
private static int BinarySearchForInsertionSort<T>(T[] Input, int p, int r, T v) where T :
IComparable<T>
        {
            int q;
            if (p == 0 && r == 0 && Input[0].CompareTo(v)>0)
                return 0;
            if (p < r)
            {
                q = (p + r) / 2;
                if (Input[q].CompareTo(v) > 0)
                {
                    BinarySearchForInsertionSort(Input, p, q, v);
                    return BinaryForInsertionSort(Input, p, q, v);
                }
                else
                {
                    BinarySearchForInsertionSort(Input, q+1, r, v);
                    return BinaryForInsertionSort(Input, q+1, r, v);
                }
            }
            return -1;
        }
```

```
        private static int BinaryForInsertionSort<T>(T[] Input, int p, int r, T v) where T :
IComparable<T>
        {
            for (int j = p; j <= r; j++)
                if (Input[j].CompareTo(v) > 0)
                    return j;
            return -1;
        }
        public static void BinaryInsertionSort<T>(T[] Input) where T : IComparable<T>
        {
            T key;
            int i, k;
            for (int j = 1; j < Input.Length; j++)
            {
                key = Input[j];
                i = j - 1;
                k = BinarySearchForInsertionSort(Input, 0, i, key);
                if (k != -1)
                {
                    for (int s = i; s>=k ; s--)
                        Input[s + 1] = Input[s];
                    Input[k] = key;
                }
            }
        }
```

***2.3-7：请给出一个运行时间为$\Theta(n \lg n)$的算法，使之能在给定一个由 n 个整数构成的集合S和另一个整数x时，判断出S中是否存在有两个其和等于x的元素。**

利用 2.3-5 中的 BINARY-SEARCH(A,v)和 2.3-6 中的 BINARYINSERTION-SORT(S)算法

ISEXISTSUM(S,x)

1 BINARYINSERTION-SORT(S)

2 **for** j←1 to n

3 k←BINARY-SEARCH(S,x-S[j])

4 **if** k!=**NIL**

5 **return TRUE**

6 **else return FALSE**

该算法的运行时间为:$\Theta(n\lg n)+(n+1)\times\Theta(\lg n)+\Theta(n)=\Theta(n\lg n)$

思考题

2-1：在合并排序中对小数组采用插入排序

尽管合并排序的最坏情况运行时间为$\Theta(n\lg n)$，插入排序的最坏情况运行时间为$\Theta(n^2)$，但插入排序中的常数因子使得它在 n 较小时，运行得要更快一些。因此，在合并排序算法中，当子问题足够小时，采用插入排序就比较合适了。考虑对合并排序做这样的修改，即采用插入排序策略，对n/k个长度为 k 的子列表进行排序，然后，再用标准的合并机制将它们合并起来，此处 k 是一个特定的值。

a) **证明最坏情况下，n/k个子列表(每一个子列表的长度为 k)可以用插入排序在$\Theta(nk)$时间内完成排序。**

b) **证明这些子列表可以在$\Theta(n\lg(n/k))$最坏情况时间内完成合并。**

c) **如果已知修改后的合并排序算法的最坏情况运行时间为$\Theta nk+n\lg(n/k)$，要使修改后的算法具有与标准合并排序算法一样的渐进运行时间，k 的最大渐进值(即Θ形式)是什么(以 n 的函数形式表示)？**

d) **在实践中，k 的值应该如何选取？**

a.$\Theta(k^2\times n/k)=\Theta(nk)$

b.每一层代价都是$\Theta(n)$，共$\lg(n/k)+1$层，因此$\Theta(n)\times(\lg(n/k)+1)=\Theta(n\lg(n/k))$

c.$k=\lg n$

d.在满足插入排序比合并排序更快的情况下，k 取最大值。

2-2：冒泡排序算法的正确性

冒泡排序(bubblesort)算法是一种流行的排序算法，它重复地交换相邻两个反序元素。

BUBBLESORT(A)

```
1   for i←1 to length[A]
2       do for j←length[A] downto i+1
3           do if A[j]<A[j-1]
4               then exchange A[j]↔A[j-1]
```

a) 设 A’表示 BULLESORT(A)的输出，为了证明 BUBBLESORT 是正确的，需要证明它能够终止，并且有：

$$A'[1] \leqslant A'[2] \leqslant \cdots \leqslant A'[n]$$

其中 n=length[A]。为了证明 BUBBLESORT 的确能实现排序的效果，还需要证明什么？

下面两个部分将证明不等式（2.3）。

b) 对第 2~4 行中的 for 循环，给出一个准确的循环不变式，并证明该循环不变式是成立的。在证明中采用本章中给出的循环不变式证明结构。

c) 利用在 b）部分证明的循环不变式的终止条件，为第 1~4 行中的 for 循环给出一个循环不变式，它可以用来证明不等式（2.3）。你的证明因采用本章中给出的循环不变式的证明结构。

d) 冒泡排序算法的最坏情况运行时间是什么？比较它与插入排序的运行时间。

a. A’中的元素全部来自于 A 中变换后的元素。

b.

初始化：j=n,子数组为 A[j..n]即 A[n..n]，此中仅有一个元素因此是已排序的。

保持：如果 A[j..n]是已排序的，按计算过程知 A[j]≤A[j+1]≤…≤A[n]，当插入元素 A[j-1]时，如果 A[j]<A[j-1]则互换 A[j]、A[j-1]，否则 A[j-1]直接插入 A[j..n]的最前，因此 A[j-1..n]也是已排序的。

终止：j=i 时循环结束，此时 A[i..n]是已排序的。与外层循环条件一直，所以算法正确。

c.

初始化：i=1 时，子数组 A[1..i-1]是空的，因此在第一轮迭代前成立。

保持：假设子数组 A[1..i-1]已排序，则之中元素是 A[1..n]中最小的 i-1 个元素，按 b 证明的循环不变式，知插入 A[i]元素后的子数组 A[1..i]是 A[1..n]中最小的 i 个元素，并且 A[1..i]亦是已排序的。

终止：当 i=n+1 时循环终止，此时已处理的子数组是 A[1..n]，A[1..n]是已排序的，这个数组就是要排序的数组。因此算法正确。

d.$(n+1)+\sum_{i=1}^{n}(n-i)+\sum_{i=1}^{n}(n-i-1)=\Theta(n^2)$，与插入排序相同

2-3：霍纳规则的正确性

以下的代码片段实现了用于计算多项式

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

的霍纳规则（Horner’s Rule）。

给定系数$a_0,a_1,a_2,\cdots,a_n$**以及 x 的值，有**

```
1       y←0
2       i←n
3       while i≥0
4           do y←i + x · y
5              i←i-1
```

a）这一段实现霍纳规则的代码的渐进运行时间是什么？

b）写出伪代码以实现朴素多项式求值（native polynomial-evaluation）算法，它从头开始计算多项式的每一个项。这个算法的运行时间是多少？它与实现霍纳规则的代码段的

运行时间相比怎样？

c) 证明一下给出的是针对第 3~5 行中 while 循环的一个循环不变式：

在第 3~5 行中 while 循环每一轮迭代的开始，有：

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$$

不包含任何项的和视为等于 0。你的证明应遵循本章中给出的循环不变式的证明结构，并应证明在终止时，有$y = \sum_{k=0}^{n} a_k x^k$**。**

d) 最后证明以上给出的代码片段能够正确的计算由系数$a_0, a_1, \cdots, a_n$**刻划的多项式。**

2-4：逆序对

设 A[1..n]是一个包含 n 个不同数的数组。如果在 i<j 的情况下，有 A[i]>A[j]，则(i,j)就称为 A 中的一个逆序对（inversion）。

a) 列出数组<2,3,8,6,1>的 5 个逆序。

b) 如果数组的元素取自集合{1,2,…,n},那么，怎样的数组含有最多的逆序对？它包含多少个逆序对？

c) 插入排序的运行时间与输入数组中逆序对的数量之间有怎样的关系？说明你的理由。

d) 给出一个算法，它能用$\Theta(\lg n)$**的最坏情况运行时间，确定 n 个元素的任何排列中逆序对的数目。（提示：修改合并排序）**

a.(2,1),(3,1),(8,6),(8,1),(6,1)

b.{n,n-1,n-2,…,1}有最多的逆序对。共$\frac{n\times(n-1)}{2}$个。

c.逆序对越多，说明运行情况越坏，所以逆序对的数量与插入排序的运行效率成反比。

d.修改 MERGE 过程的最后一个 FOR 循环即可。