第3章

列表

[3-1] 考查列表结构的查找操作。

a) 试针对教材 72 页代码 3.5 中的 List::find()，以及 78 页代码 3.17 中的 List::search()，就其在最好、最坏和平均情况下的效率做一分析对比；

【解答】

二者的效率完全一致：在最好情况下，均只需O(1)时间；在最坏情况下，均需要O(n)时间；平均而言，二者均需O(n)时间。

b) 有序性对于列表查找操作效率的提高有多大作用？

【解答】

由上可见，即便附加了有序性的条件，列表的查找效率也不能有实质的提高。究其原因在于，列表结构是通过位置来访问其中的元素——即“循位置访问”（call-by-position），这与向量的“循秩访问”（call-by-rank）迥然不同。

[3-2] 考查如教材 73 页代码 3.7 和 74 页代码 3.8 所示的列表节点插入算法 LisrNode::insertAsPred() 和 ListNode::insertAsSucc()。

a) 在什么情况下，新插入的节点既是首节点也是末节点？

【解答】

若将某元素插入当时为空的列表，则插入之后列表仅含单个节点（列表规模为1），该节点将同时扮演首节点和末节点的角色。

b) 此时，这两种算法是否依然适用？为什么？试通过实测验证你的结论。

【解答】

教材所给的算法实现，在以上特殊情况下依然可行，能够顺利地完成插入操作。

之所以能够如此，是得益于这里在内部统一设置的哨兵节点（sentinel node）。如此插入的新节点，在列表内部居于头节点和尾节点之间。

[3-3] 考查如教材 75 页代码 3.11 所示的 List::remove()算法。

当待删除的节点既是首节点也是末节点（即列表仅含单个节点）时，该算法是否依然适用？为什么？

【解答】

教材所给的算法实现，在以上特殊情况下依然可行，能够顺利地完成删除操作。

之所以能够如此，也是得益于内部的哨兵节点。当最后一个节点被删除之后，头节点和尾节点在列表内部彼此相邻，互为前驱和后继。

[3-4] 考查如教材 76 页代码 3.14 所示的 List::deduplicate()算法。

a） 给出其中循环体所具有的不变性，并通过数学归纳予以证明；

【解答】

这里的不变性是：在迭代过程中的任意时刻，当前节点p的所有前驱互不相同。

算法启动之初，p没有前驱，以上命题自然成立。

以下假定在当前迭代之后，不变性依然成立，考查随后的下一步迭代。

在此步迭代过程中，首先转至下一节点p，并通过find()接口，在其前驱中查找与之雷同者。既然此前不变性是满足的，则与p雷同的元素至多仅有一个。这个雷同元素q若果真存在，则必然会被找到，并随即通过remove(q)接口被剔除。于是无论如何，以上不变性必然再次成立。

因此根据数学归纳原理，这一不变性将始终保持，直至算法结束。那时，p即是列表的尾哨兵元素，其余元素均为它的前驱，由不变性可知它们必然互异。由此可见，该算法是正确的。

b） 试举例说明，该算法在最好情况下仅需 $\mathcal{O}$(n)时间；

【解答】

当所有元素均彼此雷同时，即属于该算法的最好情况。此时，deduplicate()算法依然需要执行$\mathcal{O}$(n)步迭代，但可以证明每一步只需$\mathcal{O}$(1)时间。

实际上，根据以上所指出的不变性，当前节点p始终只有1个前驱（并且与之雷同）。因此，每步迭代中的find()操作仅需常数时间。

由此也可顺便得出最坏情况——所有元素彼此互异。在此种情况下，当前节点p的前驱数目（亦即各次find()操作所对应的时间）将随着迭代的推进线性地递增，平均为$\mathcal{O}$(n)，算法总体的时间复杂度为$\mathcal{O}(n^2)$。

c） 试改进该算法，使其时间复杂度降至 $\mathcal{O}$(nlogn)；

【解答】

最简明的一种改进方法是：首先调用sort()接口，借助高效的算法在$\mathcal{O}$(nlogn)时间内将其转换为有序列表；进而调用uniquify()接口，在$\mathcal{O}$(n)时间内剔除所有雷同元素。

d） $\mathcal{O}$(nlogn)的效率是否还有改进的余地？为什么？

【解答】

就最坏情况下复杂度的意义而言，以上方法已属最优。

为证明这一结论，只需构造一个从元素唯一性（Element Uniqueness）问题，到列表排序问题的线性归约。请读者仿照此前第2章习题[2-12]，自行给出具体的构造方法。

[3-5] 试基于列表的遍历接口 traverse()实现以下操作（假定数据对象类型支持算术运算）：

a) increase()：所有元素数值加一；

【解答】

一种可行的实现方式，如代码x3.1所示。

```
template <typename T> struct Increase //函数对象：递增一个T类对象
   {  virtual void operator() ( T& e ) { e++; }  }; //假设T可直接递增或已重载++

template <typename T> void increase ( List<T> & L ) //统一递增列表中的各元素
{  L.traverse ( Increase<T>() );  } //以Increase<T>()为基本操作进行遍历
```

代码x3.1 基于遍历实现列表的increase()功能

与教材代码2.16中向量increase()接口的实现方式同理，这里也是将同一名为Increase()的函数对象作为基本操作，并通过遍历接口对列表的所有元素逐一处理。

b) half()：所有元素数值减半。

【解答】

一种可行的实现方式，如代码x3.2所示。

```
template <typename T> struct Half //函数对象：减半一个T类对象
   {  virtual void operator() ( T& e ) { e /= 2; }  }; //假设T可直接减半

template <typename T> void half ( List<T> & L ) //统一减半列表中的各元素
{  L.traverse ( Half<T>() );  } //以Half<T>()为基本操作进行遍历
```

代码x3.2 基于遍历实现列表的half()功能

与以上Increase的实现方式相仿，这里的关键依然在于定义一个名为Half的函数对象。

[3-6] 对数据结构的操作，往往都集中于数据元素的一个较小子集。因此对列表而言，若能将每次被访问的节点及时转移至查找长度更短的前端，则整体效率必将大为提高。这种能够自适应调整的列表，即所谓的自调整列表（self-adjusting list）。

试通过改造本章的 List 模板类，实现自适应列表结构。

【解答】

读者可以按照以下操作准则，独立完成改进：

1）新元素总是作为首节点被插入；

2）已有的元素一旦接受访问，也随即将其转移至最前端（作为首元素）。

通常的应用环境都具有较强甚至极强的数据局部性（data locality）——在其生命期的某一区间内，对列表结构的访问往往集中甚至限定于某一特定的元素子集。引入以上策略之后，子集中元素所对应的节点，很快会“自适应地”集中至列表的前端。在此后相当长的一段时间内，其余的元素几乎可以忽略。于是，在此期间此类列表的访问效率，将主要取决于该子集（而非整个全集）的规模，因此上述改进的实际效果非常好（参见教材8.1.1节）。

[3-7] 自学 C++ STL 中 list 容器的使用方法，阅读对应的源代码。

【解答】

请读者独立完成阅读任务。

[3-8] 考查插入排序算法。

a） 仿照教材 80 页代码 3.19，针对向量实现插入排序算法 Vector::insertionSort()；

【解答】

请读者独立完成算法的设计与实现任务。

b） 你实现的插入排序算法是稳定的吗？为什么？

【解答】

请读者根据具体的实现方法，给出分析及结论。

[3-9] 考查选择排序算法。

a） 仿照教材 81 页代码 3.20，试针对向量结构实现选择排序算法 Vector::selectionSort()；

【解答】

一种可行的实现方式，如代码x3.3所示。

```
template <typename T> //向量选择排序
void Vector<T>::selectionSort ( Rank lo, Rank hi ) { //assert: 0 < lo <= hi <= size
   while ( lo < --hi )
      swap ( _elem[max ( lo, hi ) ], _elem[hi] ); //将[hi]与[lo, hi]中的最大者交换
}

template <typename T>
Rank Vector<T>::max ( Rank lo, Rank hi ) { //在[lo, hi]内找出最大者
   Rank mx = hi;
   while ( lo < hi-- ) //逆向扫描
      if ( _elem[hi] > _elem[mx] ) //且严格比较
         mx = hi; //故能在max有多个时保证后者优先，进而保证selectionSort稳定
   return mx;
}
```

代码x3.3 向量的选择排序算法

b） 你实现的选择排序算法是稳定的吗？为什么？

【解答】

是稳定的。

这里在未排序子向量中查找最大元素时，总是自后向前地逆向扫描；相应地，唯有遇到严格更大的元素时，才更新最大元素的记录。如此，即便最大元素有重复的多个，每次都必定会选中其中最靠后者（并进而将其转移至已排序子向量）。于是，每一组重复的元素，都将按照其在原向量中的相对位置依次转移，从而最终保持它们之间的相对位置。

[3-10] 假定序列中 n 个元素的数值为独立均匀地随机分布，试证明：

a） 列表的插入排序算法平均需做约 $n^2/4 = \mathcal{O}(n^2)$次元素比较操作；

【解答】

首先，平均意义下的比较操作次数，也就是概率意义下比较操作的期望次数。

该算法共需执行$\mathcal{O}(n)$步迭代，故根据期望值的线性律（linearity of expectation），比较操作总次数的期望值，应该等于各步迭代中比较操作次数的期望值之和。

该算法中的比较操作，主要消耗于对有序子列表的search()查找过程。

由3.4.2节的分析结论，search()接口具有线性的平均复杂度。这就意味着，各步迭代内的search()过程所涉及的比较操作次数，应从0到n - 1按算术级数线性递增，故其总和应为：

$$\sum_{k=0}^{n-1}(k/2) \quad = \quad n\cdot(n - 1)/4 \quad = \quad \mathcal{O}(n^2)$$

b） 向量的插入排序算法平均需做约 $n^2/4 = \mathcal{O}(n^2)$次元素移动操作；

【解答】

与列表不同，向量的插入排序中search()查找接口可以采用二分查找之类的算法，从而使其复杂度从线性降低至$\mathcal{O}(\log n)$。

然而另一方面，在确定适当位置之后为将新元素插入已排序的子序列，尽管列表只需$\mathcal{O}(1)$时间，但向量在最坏情况下我们不得移动$\mathcal{O}(n)$个节点，而且平均而言亦是如此。故与a）同理，总体而言，平均共需执行$\mathcal{O}(n^2)$次元素移动操作。

由这个例子，可清楚地看出两种数据访问方式各自的长处：循秩访问的方式更适宜于静态的查找操作，但在频繁动态修改的场合却显得效率低下；循位置访问的方式更适宜于动态修改，却不能高效地支持静态查找。

向量结构与列表结构所呈现的这种对称性，既非常有趣，更耐人寻味。

c） 序列的插入排序算法过程中平均有 expected-$\mathcal{O}(\log n)$个元素无需移动。

【解答】

同样地，既然该算法由多步迭代构成，故其间无需移动的元素的期望数目，就应该等于各步迭代中，待插入元素无需移动的概率总和。

根据该算法的原理，对于任意$k \in [0, n)$，在第k步迭代启动之初，当前元素A[k]的k个前驱应该业已构成一个有序的子序列A[0, k)。不难看出，若A[k]无需移动即使得A[0, k]仍为有序子序列，则其充要条件是，A[k]在A[0, k]中为最大元素。

既然假定所有元素都符合独立且均匀的随机分布，故作为前k + 1个输入元素中的普通一员，A[k]在其中为最大元素的概率应与其它元素均等，都是1/(k +1)。于是，这一概率的总和即为：

$$\sum_{k=0}^{n-1}1/(k + 1) \quad = \quad \sum_{k=1}^{n}1/k \quad = \quad \Theta(\log n)$$

[3-11] 序列中元素 A[i]和 A[j]若满足 i < j 且 A[i] > A[j]，则称之为一个逆序对（inversion）。考查如教材 80 页代码 3.19 所示的插入排序算法 List::insertionSort()，试证明：

a) 若所有逆序对的间距均不超过 k，则运行时间为 $\mathcal{O}$(kn)；

【解答】

算法进入到A[j]所对应的那步迭代时，该元素（在输入序列中）的所有前驱应该业已构成一个有序子序列A[0, j)。既然其中至多只有k个元素与A[j]构成逆序对，故查找过程search()至多扫描其中的k个元素，即可确定适当的插入位置，对应的时间不超过$\mathcal{O}$(k)。

实际上，每一步迭代均具有如上性质，故累计运行时间不超过$\mathcal{O}$(kn)。

在教材12.3节对希尔排序高效性的论证中，这一结论将至关重要。

b) 特别地，当 k 为常数时，插入排序可在线性时间内完成；

【解答】

这也就是a）中结论的一个自然推论。

c) 若共有 I 个逆序对，则关键码比较的次数不超过 $\mathcal{O}$(I)；

【解答】

这里定义的每一逆序对，均涉及两个元素。为便于分析，这里约定将其计入后者的“账”上。因此，所有元素的逆序前驱的数目总和，应恰好等于I。

将a)中的分析方法作一般化推广，即不难看出：每个元素所涉及比较操作的次数，应恰好等于其逆序前驱的数目；整个算法过程中所执行的比较操作的总数，应恰好等于所有元素的逆序前驱的数目总和，亦即I。

d) 若共有 I 个逆序对，则运行时间为 $\mathcal{O}$(n + I)。

【解答】

由以上分析，算法过程中消耗于比较操作的时间可由$\mathcal{O}$(I)界定，而消耗于移动操作的时间可由$\mathcal{O}$(n)界定，二者累计即为$\mathcal{O}$(n + I)。

既然此处实际的运行时间更多地取决于逆序对的数目，而不仅仅是输入序列的长度，故插入排序亦属于所谓输入敏感的（input sensitive）算法。

实际上更为精确地，每步迭代中的查找都是以失败告终——或者找到不大于当前元素者，或者抵达A[-1]越界。若将这两类操作也归入比较操作的范畴，则还有一个$\mathcal{O}$(n)项。好在就渐进意义而言，这一因素可以忽略。

[3-12] 如教材 80 页代码 3.19 所示，考查插入排序算法 List::insertionSort()。

a) 若输入列表为{ 61, 60, 59, ..., 5, 4, 3, 2, 0, 1, 2 }，则共需要做多少次关键码比较？

【解答】

这里的关键在于，如何统计出查找过程search()（教材78页代码3.17）在该算法各步迭代中所执行的比较操作次数。

具体如表x3.1所示，列表结构中的头、尾哨兵，分别等效于元素-∞和+∞；已排序的子序列，用阴影表示；在当前迭代步经查找并归入排序子序列的元素，用方框注明。

就此例而言，共计63个元素，故对应于63步迭代，依次从0至62编号，各对应于一行。

表x3.1 列表{ 61, 60, 59, ..., 5, 4, 3, 2, 0, 1, 2 }的插入排序过程

迭代编号	列表元素															比较次数
0	-∞	$\boxed{61}$	60	59	58	...	5	4	3	2	0	1	2	+∞	1
1	-∞	$\boxed{60}$	61	59	58	...	5	4	3	2	0	1	2	+∞	2
2	-∞	$\boxed{59}$	60	61	58	...	5	4	3	2	0	1	2	+∞	3
3	-∞	$\boxed{58}$	59	60	61	...	5	4	3	2	0	1	2	+∞	4
...	-∞	...					...							+∞	...
56	-∞	$\boxed{5}$	6	7	8	...	61	4	3	2	0	1	2	+∞	57
57	-∞	$\boxed{4}$	5	6	7	...	60	61	3	2	0	1	2	+∞	58
58	-∞	$\boxed{3}$	4	5	6	...	59	60	61	2	0	1	2	+∞	59
59	-∞	$\boxed{2}$	3	4	5	...	58	59	60	61	0	1	2	+∞	60
60	-∞	$\boxed{0}$	2	3	4	...	57	58	59	60	61	1	2	+∞	61
61	-∞	0	$\boxed{1}$	2	3	...	56	57	58	59	60	61	2	+∞	61
62	-∞	0	1	2	$\boxed{2}$	...	55	56	57	58	59	60	61	+∞	60

由该表可以看出，第0步迭代经过一次（当前元素61与头哨兵-∞的）比较，即可确定其适当的插入位置——当然，此步的插入操作其实可以省略，但为简化控制逻辑，算法中不妨统一处理。实际上，从第0步至第60步迭代所插入的元素，在当时都是最小的，故每一查找过程search()都会终止于头哨兵-∞，而新元素都会被转移至最前端作为首元素。由此可见，这些迭代步所对应的比较次数，应从1至61逐步递增。

最后两步迭代原理一样，但过程与结果略有区别。第61步迭代中的查找过程search()需做61次比较，最后终止于节点0。第62步迭代中的查找过程search()需做60次比较，最后终止于节点2。请特别留意，最后一步迭代对两个雷同元素2的处理方式——既然search()算法是稳定的，故后一元素2应被插入于前一元素2之后。

累计以上各步迭代，比较操作的总次数应为：

$$(1 + 2 + 3 + ... + 60 + 61) + 61 + 60 = 2012$$

利用此前“插入排序算法复杂度主要取决于逆序对总数”的结论，也可得到同一结果。诚如前言，我们不难验证：以上各步迭代中所做比较操作的次数，恰好就是（在输入序列中）与当前元素构成逆序对的前驱总数。具体地，对于前61个元素：

{ 61, 60, 59, ..., 3, 2, 0 }

而言，逆序前驱数依次为：

{ 0, 1, 2, ..., 58, 59, 60 }

而对于最后两个元素：

    { 1, 2 }

而言，逆序前驱数分别为：

    { 60, 59 }

因此，原输入序列所含逆序对的总数应为：

    I = (0 + 1 + 2 + ... + 59 + 60) + 60 + 59 = 1949

再计入每个元素所对应的最后一次失败的比较，该算法累计执行的比较操作次数应为：

    I + n = 1949 + 63 = 2012

可见，两种方法殊途同归。

b） 试通过实测验证你的结论。

【解答】

只需在查找过程search()中，在执行比较操作的同时计数，查找返回时打印所记次数即可。

[3-13] 教材 81 页代码 3.20 中的 List::selectionSort()算法，通过 selectMax()在前端子序列中定位最大元素 max 之后，将其对应的节点整体取出，再后移并归至后端子序列之首。

这一过程中的 remove()和 insertB ()接口涉及节点存储空间的动态释放（delete）与申请（new），二者虽均属于 $O(1)$复杂度的基本操作，但根据实验统计，此类操作实际所需的时间较之一般的基本操作多出两个数量级。

其实，教材 80 页的图 3.6 已暗示了一个更好的实现方式：只需令 max 与前端子序列的末元素互换数据项即可。

a） 试按照这一思路，在代码 3.20 的基础上完成改进；

【解答】

只需将第7行改为：

```
swap( tail->pred->data, max->data );
```

b） 通过实际测试统计验证，新的版本的确比代码 3.20 更加高效。

【解答】

请读者根据自己的改进方法，独立完成。

[3-14] 考查经过以上改进之后的 List::selectionSort()算法。通过 selectMax()在前缀子序列中定位的最大元素 max，有可能恰好就是 tail 的前驱——自然，此时“二者”的交换是多余的。针对这一“问题”，你或许会考虑做些“优化”，比如将第 7 行进一步改为：

```
if ( tail->pred != max ) swap( tail->pred->data, max->data );
```

a） 以序列{ 1980, 1981, 1982, ..., 2011, 2012; 0, 1, 2, ..., 1978, 1979 }为例，这种情况共发生多少次？

【解答】

我们首先引入循环节（cycle）的概念。

考查序列A[0, n)以及与之对应的排序序列S[0, n)。若存在

$0 \le \{ k_0, k_1, k_2, ..., k_{d-1} \} < n$

使得对于任意0 ≤ i < d，都有

$A[k_i] = S[k_{(i+1) \bmod d}]$

则称对于序列A[0, n)而言，$\{ k_0, k_1, k_2, ..., k_{d-1} \}$构成[0, n)的一个循环节。

以本题所给的序列为例，不难验证有：

```
A[1980] = 1947 = S[1947]
A[1947] = 1914 = S[1914]
A[1914] = 1881 = S[1881]
    ...
A[  66] =   33 = S[  33]
A[  33] =    0 = S[   0]
A[   0] = 1980 = S[1980]
```

因此相对于该序列，以下即为一个循环节：

{ 1980, 1947, 1914, ..., 66, 33, 0 }

这是一个等差数列，公差为33，总计的项数（即该循环节的长度）为：

$d = (1980 - 0)/33 + 1 = 61$

不难理解，每个元素都应属于某个循环节（长度可能为1），但不可能同时属于两个循环节。这就意味着，按照上述定义，任何序列都可以唯一地分解为若干个彼此独立的循环节。

接下来，我们重新审视选择排序算法List::selectionSort()的每一步迭代，假设被选出的最大元为A[m]。不难看出，将m转移至tail之前的效果，等同于该元素所属的循环节长度减一；而其它元素所属循环节的长度不变。当然，若该循环节的长度减至0，则意味着该循环节消失。

特别地，若题中所建议的“优化”能够生效，则此时的A[m]就是排序序列中的S[m]；这就意味着，A[m]必然自成一个（长度为1的）循环节。反之，一旦A[m]所属循环节的长度缩减至1，则“优化”也必然生效。由此可见，该“优化”措施恰好对每个循环节生效一次；而在整个算法过程中生效的总次数，应恰好等于输入序列所含循环节的数目。

现在，我们再回到本题。根据上述分析结论，我们只需统计出题中所给序列中的循环节总数。

实际上就这一序列而言，[0, 2012]范围内每一个公差为33的等差数列，均构成一个循环节。而且，因为有：

$2013 = 33 \times 61$

所以与上面所指出的那个循环节一样，每个循环节的长度均为61，共计33个循环节。

更精细地考查代码可以发现，在处理到最后一个循环节的最后一个元素（当时的A[0] = 0）

时，该算法会直接退出而不会继续选取最大元并做移动，故上述“优化”生效的实际次数为：

33 - 1 = 32

b）试证明，在各元素等概率独立分布的情况下，这种情况发生的概率仅为 $\ln n/n \to 0$——也就是说，就渐进意义而言，上述“优化”得不偿失。

【解答】

继续考查列表的选择排序算法后不难发现，在A[m]所属循环节消失之前的瞬间，A[m]应为A[0, m]中的最大元。鉴于此前的A[0, m]一直符合独立且均匀的随机分布，故发生这一事件的概率应与区间[0, m]的长度成反比。

进一步地，再次根据期望值的线性律（linearity of expectation），在n次循环中发生这种情况的期望次数，应等于各步迭代中发生这一事件的概率总和，亦即：

$$1/n + 1/(n - 1) + 1/(n - 2) + \ldots + 1/3 + 1/2 + 1/1 = \Theta(\ln n)$$

[3-15] 在如教材 82 页代码 3.21 所示的 List::selectMax()算法中，若将判断条件由

```
! lt( (cur = cur->succ)->data, max->data )
```

改为

```
lt( max->data, (cur = cur->succ)->data )
```

则如代码 3.20 所示的 selectionSort()算法的输出有何变化？试举一例。

【解答】

教材所给的算法，是按从前（左）向后（右）的次序扫描各元素，再将选出来的当前最大元后移，并归入已排序的后缀子序列。就其语义而言，题中的两个逻辑表达式都旨在指示“当前的最大元记录需要更新”，故都能保证正确地挑选出最大元。然而在有多个雷同的最大元时，二者之间却又存在着细微而本质的差异。

按照前一逻辑表达式，在同时存在多个最大元时，算法总是会选出其中的最靠后（右）者。因此，原序列中雷同元素之间的相对次序，将在排序后的序列中得以延续。反之，若调整为后一逻辑表达式，则在同时存在多个最大元时，算法总是会选出其中的最靠前（左）者。如此，雷同元素在最终输出序列中的次序，将会完全颠倒。

比如，对于如下输入序列：

$\{ 5, 3_a, 9, 3_b, 3_c, 2 \}$

若采用后一逻辑表达式，则对应的输出序列将是：

$\{ 2, 3_c, 3_b, 3_a, 5, 9 \}$

而若采用前一逻辑表达式，则对应的输出序列将是：

$\{ 2, 3_a, 3_b, 3_c, 5, 9 \}$

由上可见，就对雷同元素的处理方式而言，教材所给的算法实现更为精细和恰当，可以保证以selectMax()为基础的选择排序算法selectionSort()是稳定的。

[3-16] 考查如教材 83 页代码 3.23 所示的 List::mergeSort()算法，试证明：

a） 若为节省每次子列表的划分时间，而直接令 m = min(c, n/2)，其中 c 为较小的常数（比如 5），则总体复杂度反而会上升至 $\mathcal{O}(n^2)$；

【解答】

做如此调整之后，在切分出来的两个子列表中，必有其一的长度不超过c，而另一个的长度不小于n - c。尽管如此可以在$\mathcal{O}$(c)时间内完成子任务的划分，但二路归并仍需$\mathcal{O}$(n)时间，因此其对应的递推方程应为：

$$T(n) \quad = \quad T(c) + T(n - c) + \mathcal{O}(n)$$

解之可得：

$$T(n) \quad = \quad \mathcal{O}(n^2)$$

b） 特别地，当取 c = 1 时，该算法等效地退化为插入排序。

【解答】

此时，参与二路归并的每一对子列表中，总有一个长度为1。故就实际效果而言，原算法的二路递归将退化为单分支的线性递归，“归并”操作将等同于将单个元素插入至另一子列表中。因此，整个计算过程及效果均完全等同于插入排序。

[3-17] 考查基于 List::merge()算法（教材 82 页代码 3.22）实现的 List::mergeSort()算法（教材 83 页代码 3.23）。

该算法是稳定的吗？若是，请给出证明；否则，试举一实例。

【解答】

是稳定的。为证明这一点，只需证明雷同元素之间的相对次序在输出序列中依然保持。

不妨采用数学归纳法。假定在每一次二路归并过程中，上述命题对任何长度短于L的列表都成立。以下考查长度为L的列表。

考查List::merge()算法中两个子列表的当前节点p（左）和q（右）。根据这里的逻辑条件，只要p不大于q，都会将p取出并归入输出列表；即便p和q相等，p也会先于q转入输出列表。也就是说，与p和q雷同的所有元素之间的相对次序将会得以延续。

[3-18] 试仿照教材 22 页代码 1.10 中向量的倒置算法，实现 List::reverse()接口，将列表中元素的次序前后倒置。

【解答】

这里，由繁至简给出该算法的三种实现方式。

第一种实现方式如代码x3.4所示。

```
template <typename T> void List<T>::reverse() { //前后倒置
   if ( _size < 2 ) return; //平凡情况
   ListNodePosi(T) p; ListNodePosi(T) q;
```

```
   for ( p = header, q = p->succ; p != trailer; p = q, q = p->succ )
      p->pred = q; //自前向后，依次颠倒各节点的前驱指针
   trailer->pred = NULL; //单独设置尾节点的前驱指针
   for ( p = header, q = p->pred; p != trailer; p = q, q = p->pred )
      q->succ = p; //自前向后，依次颠倒各节点的后继指针
   header->succ = NULL; //单独设置头节点的后继指针
   swap ( header, trailer ); //头、尾节点互换
}
```

代码x3.4 列表倒置算法的第一种实现

这里，借助两个指针逐一处理相邻的各对节点。首先通过一趟遍历，自前向后地依次颠倒各节点的前驱指针（使之指向当前的后继）。接下来再通过一趟遍历，自前向后地依次颠倒各节点的后继指针（使之指向此前的前驱）。当然，在两趟遍历之后，还需要单独地设置尾节点的前驱指针、头结点的后继指针。至此，所有节点的排列次序均已完全颠倒，故最后只需交换头、尾节点的指针，即可实现整体倒置的效果。

第二种实现方式如代码x3.5所示。

```
template <typename T> void List<T>::reverse() { //前后倒置
   if ( _size < 2 ) return; //平凡情况
   for ( ListNodePosi(T) p = header; p; p = p->pred ) //自前向后，依次
      swap ( p->pred, p->succ ); //交换各节点的前驱、后继指针
   swap ( header, trailer ); //头、尾节点互换
}
```

代码x3.5 列表倒置算法的第二种实现

这里仅需使用一个指针p。借助该指针，自前向后地对整个列表做一趟遍历，并令每个节点的前驱、后继指针互换。同样地，最后还需令头、尾节点的指针互换。

第三种实现方式如代码x3.6所示。

```
template <typename T> void List<T>::reverse() { //前后倒置
   ListNodePosi(T) p = header; ListNodePosi(T) q = trailer; //头、尾节点
   for ( int i = 1; i < _size; i += 2 ) //（从首、末节点开始）由外而内，捉对地
      swap ( ( p = p->succ )->data, ( q = q->pred )->data ); //交换对称节点的数据项
}
```

代码x3.6 列表倒置算法的第三种实现

这一实现方式的思路与策略，与教材代码1.10中的倒置算法如出一辙。具体地，这里通过一轮迭代，从首、末节点开始由外而内，捉对地交换各对称节点的数据项。

指针p和q始终指向一对位置对称的节点。请注意，尽管其初值分别为头、尾哨兵节点，但进入迭代之后，它们所指向的都是对外可见的有效节点。

无论以上何种实现，计算过程无非都是一或两趟遍历迭代，每一步迭代都只涉及常数次基本操作，因此整体的时间复杂度均为$\mathcal{O}(n)$。另外，除了列表本身，这些实现方式均只需常数的辅助空间，故也都属于就地算法。

综合而言，最后一种实现方式的形式更为简明，但若基础数据类型T本身较为复杂，则节点data项的直接交换可能导致耗时的构造、析构运算。而前两种实现方式虽形式略嫌复杂，但因为仅涉及指针赋值，故在基础类型T较为复杂的场合将更为高效。

[3-19] Josephus 环游戏的规则如下：

> **一个刚出锅的山芋，在围成一圈的 n 个孩子间传递。大家一起数数，每数一次，当前拿着山芋的孩子就把山芋转交给紧邻其右的孩子。一旦数到事先约定的某个数 k，拿着山芋的孩子即退出，并从该位置起重新数数。如此反复，最后剩下的那个孩子就是幸运者。**

a） 试实现算法 josephus(int n, int k)，输出孩子们出列的次序，并确定最终的幸运者；

【解答】

在本章所实现列表结构的基础上做扩展，使之成为所谓的循环列表（Circular list）。也就是说，首节点的前驱取作末节点，末节点的后继取作首节点。于是，无论是通过前驱还是后继引用，都可以反复地遍历整个列表。

初始时，将n个孩子组织为一个循环列表。此后借助后继引用，不难找到下一出列的孩子；其出列的动作，及对应于删除与之对应的节点。如此不断反复。

b） 该算法的时间、空间复杂度各是多少？

【解答】

整个算法所需的空间主要消耗于循环列表，其最大规模不过$\mathcal{O}(n)$。因为这里无需使用前驱引用，故实际上空间消耗还可进一步减少，但渐进地依然是$\mathcal{O}(n)$。

算法的执行时间，主要消耗于对游戏过程的模拟。每个孩子出列之前，都需沿着后继引用前进k步，故累计需要$\mathcal{O}(n \cdot k)$时间。

当然，在$n < k$时，可以通过取“k %= n”，进一步加快模拟过程。如此，时间复杂度应为：

$\mathcal{O}(n \cdot \mathrm{mod}(k, n)) = \mathcal{O}(n^2)$。