第 3 章
奖学金争先

本章主要通过“谁能拿到奖学金”案例来介绍排序的概念，以及常用的排序方法。通过对“荷兰国旗”“货物移动”问题的分析及求解，进一步掌握排序知识。

3.1 谁能拿到奖学金

问题描述：某校的惯例是在每个学期的期末考试之后发放奖学金，奖学金共有 5 种，获得奖学金的条件如下：A 种为院士奖学金，奖金为 8000 元，期末平均成绩高于 80 分，并且在本学期内发表一篇或一篇以上论文的学生均可获得；B 种为五四奖学金，奖金为 4000 元，期末平均成绩高于 85 分，并且班级评议成绩高于 80 分的学生均可获得；C 种为成绩优秀奖，奖金为 2000 元，期末平均成绩高于 90 分的学生均可获得；D 种为西部奖学金，奖金为 1000 元，期末平均成绩高于 85 分的西部地区的学生均可获得；E 种为班级贡献奖，奖金为 850 元，班级评议成绩高于 80 分的学生干部均可获得。

奖学金分为 A、B、C、D、E 共 5 种，每种奖学金的获奖人数没有限制，只要符合条件都可以获得，每名学生也可以同时获得多种奖学金。例如，姚林的期末平均成绩为 87 分，班级评议成绩为 82 分，同时他还是学生干部，那么他可以同时获得五四奖学金和班级贡献奖，即奖学金总额为 4850 元。

现给出若干名学生的相关数据，要求计算：

（1）哪三名学生获得的奖学金总额最高？（假设总有学生能满足获得奖学金的条件。）

（2）按获得奖学金总额由高到低的顺序将所有学生重新排序。

3.2 常用排序方法介绍

3.2.1 概述

首先，对有关排序的一些概念进行简单的介绍。

什么是排序？排序是计算机内经常进行的一种操作，其目的是将一组无序的记录序列调整为有序的记录序列。例如，将关键字序列 52,49,80,36,14,58,61,23,97,75 调整为 14,23,36,49,52,58,61,75,80,97，此过程就是排序。

定义：假设含 n 个记录的序列为 $\{R_1,R_2,\cdots,R_n\}$，其相应的关键字序列为 $\{K_1,K_2,\cdots,K_n\}$，这些关键字之间可以相互比较，即它们之间存在如下关系：

$$K_{p1} \leqslant K_{p2} \leqslant \cdots \leqslant K_{pn}$$

按此固有关系，将上述记录序列重新排列为 $\{R_{p1},R_{p2},\cdots,R_{pn}\}$，这样的过程或操作称为排序。

内部排序和外部排序：若整个排序过程无须访问外存便能完成，则称此类排序问题为内部排序；反之，若参加排序的记录数量很大，整个排序过程不可能在内存中完成，则称此类排序问题为外部排序。在排序过程中，需要在内存和外存之间进行数据交换。

稳定排序和非稳定排序：假设 $K_i=K_j$（$i\neq j$），并且在排序前的序列中，R_i 领先于 R_j（$i<j$），若在排序后的序列中，R_i 仍然领先于 R_j，则称该排序方法是稳定的，即为稳定排序；反之，若在排序后的序列中 R_j 领先于 R_i，则称该排序方法是非稳

定的，即为非稳定排序。一个排序方法是稳定的还是非稳定的，并不影响整个排序的结果。

通常，在排序过程中需要进行下列两种基本操作：①比较两个关键字的大小；②将记录从一个位置移动至另一个位置。在本章的后续讨论中，设待排序的一组记录以顺序方式存储，记录的关键字均为整数，待排序记录类型定义如下所示。

```
#define MAXSIZE  1000 //待排顺序表最大长度
typedef  int  KeyType;//关键字类型为整型

typedef  struct {
    KeyType   key;        //关键字项
    InfoType  otherinfo;  //其他数据项
} RcdType;               //记录类型
typedef  struct {
    RcdType  R[MAXSIZE+1]; //r[0]闲置
    int      length;       //顺序表长度
} SqList;                  //顺序表类型
```

3.2.2　直接插入排序

直接插入排序是一种最简单的排序方法，其基本操作是将一个记录插入到已完成排序的有序表中，从而得到一个新的、记录数增 1 的有序表。一趟插入排序算法的基本思想示意图，如图 3.1 所示。

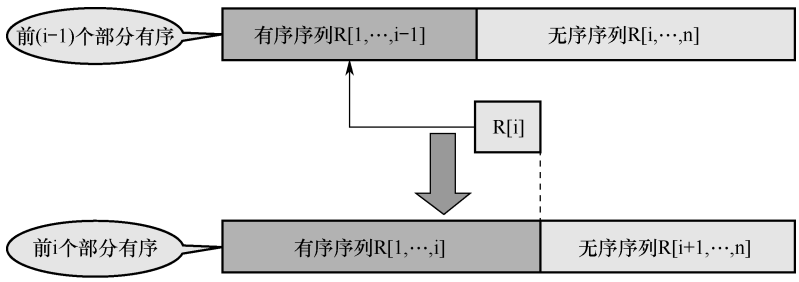

图 3.1
一趟插入排序算法的基本思想示意图

一趟插入排序算法的基本思想是：将整个序列分成两部分，前半部分为有序序列，即 R[1,⋯,i−1]，后半部分为无序序列。现在要把第 i 个关键字插入到前半

部分有序序列中合适的位置。经过查找处理，确定其合适的插入位置，然后将其插入，使得前 i 个关键字部分有序，此时完成第 i 个关键字的插入。这个过程称为一趟插入排序。

实现一趟插入排序，可以分为以下三步：

（1）在 R[1,⋯,i−1]中查找 R[i]的插入位置 j+1，即

$$R[1,\cdots,j].key \leqslant R[i].key < R[j+1,\cdots,i-1].key$$

（2）将 R[j+1,⋯,i−1]中的所有记录全部后移一个位置。

（3）将 R[i]插入（复制）到 R[j+1]位置。

直接插入排序就是利用顺序查找实现在 R[1,⋯,i−1]中查找 R[i]的插入位置。以序列 49,38,65,97,76,13,27,49 为例，其直接插入排序过程如图 3.2 所示。

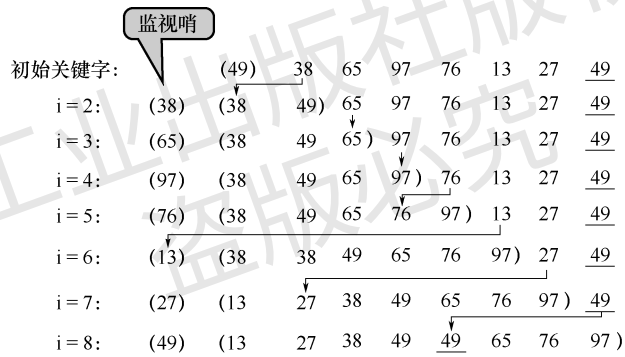

图 3.2
直接插入排序过程

直接插入排序的算法流程图如图 3.3 所示。

对于给定的一组关键字，从第二个关键字开始依次插入，直至插入第 n 个关键字为止。当插入第 i 个关键字时，首先将 L.R[i]放置于 L.R[0]处，然后 j 从 i−1 开始向前依次进行判断。若 L.R[j].key>L.R[0].key，则 L.R[j]后移，同时 j 减 1，继续判断；若 L.R[j].key <L.R[0].key，则 L.R[j+1]为 L.R[i]的正确位置，将 L.R[0]放置于 L.R[j+1]处。其中，L.R[0]为监视哨。

直接插入排序算法的示例代码如下所示。

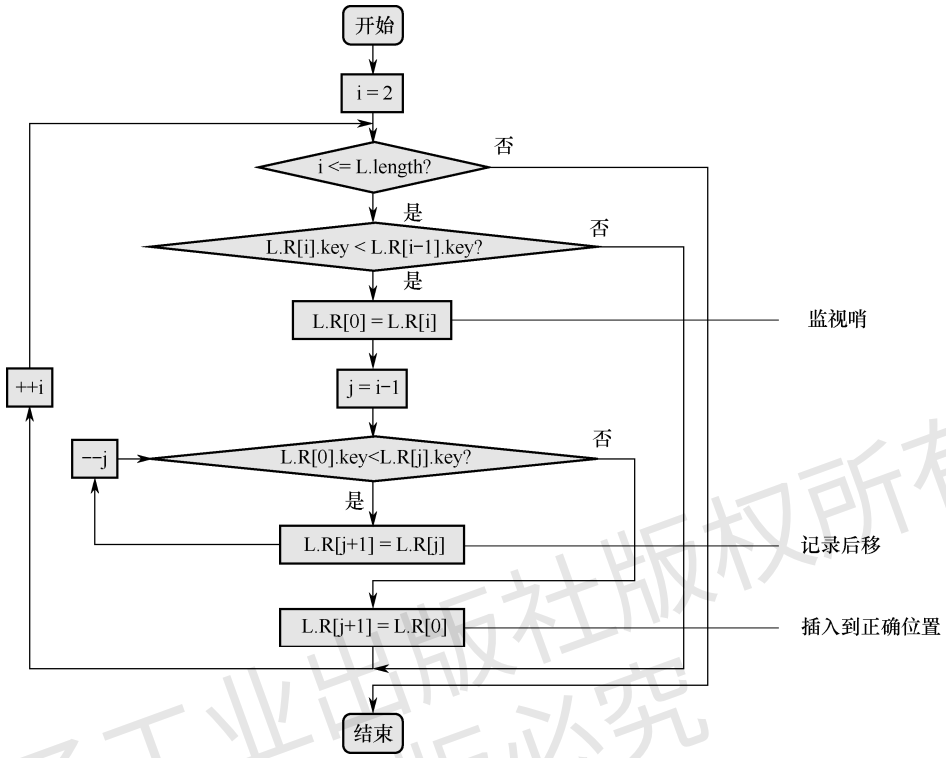

图 3.3
直接插入排序的算法流程图

```
void InsertionSort ( SqList &L ) {//对顺序表 L 进行直接插入排序
    for ( i=2; i<=L.length; ++i )
          if (L.R[i].key < L.R[i-1].key) {
          L.R[0] = L.R[i];           //复制为监视哨
          for ( j=i-1; L.R[0].key < L.R[j].key;  -- j )
                L.R[j+1] = L.R[j];       //记录后移
          L.R[j+1] = L.R[0];       //插入到正确位置
       }
} // InsertSort
```

对直接插入排序进行分析，最好的情况是关键字在记录序列中已经有序了，则插入第 i 个关键字的比较次数为 1，移动次数为 0，所以总的比较次数为最小值，即

$$\sum_{i=2}^{n}1=n-1$$

此时，总的移动次数为 0。最坏的情况是关键字在记录序列中完全逆序，此时，插入第 i 个关键字的比较次数的最大值为 i，移动次数最大值为 i+1，所以总的比较次数为

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

总的移动次数为

$$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$$

因此，在关键字基本有序的情况下，使用直接插入排序插入关键字，其比较次数和移动次数均较少，效率较高。

3.2.3 起泡排序

起泡排序是一种简单的交换排序。首先将第 1 个记录的关键字和第 2 个记录的关键字进行比较，若为逆序（L.R[1].key>L.R[2].key），则将两个记录交换，然后比较第 2 个记录和第 3 个记录的关键字，以此类推，直至第(n−1)个记录的关键字和第 n 个记录的关键字比较结束。上述过程称为第 1 趟起泡排序，其结果使得关键字最大的记录移动到最后一个位置。

第 i 趟起泡排序是指从第 1 个记录的关键字和第 2 个记录的关键字进行比较开始，将相邻的两个记录的关键字依次进行比较，当出现逆序时，将两个记录交换，以此类推，直到第(n−i+1)个记录中关键字最大的记录移动到(n−i+1)位置上，使得后面的记录序列由原来的(i−1)个记录有序变成 i 个记录有序。第 i 趟起泡排序的过程如图 3.4 所示。

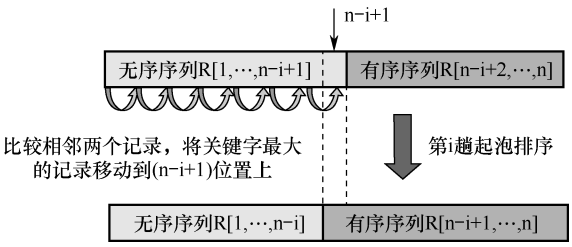

图 3.4
第 i 趟起泡排序的过程

仍以序列 49,38,65,97,76,13,27,49 为例，为了更形象地表述起泡排序，将该序列按从上到下的顺序来写，起泡排序过程如图 3.5 所示。

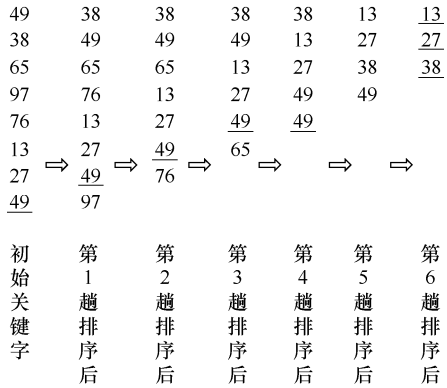

图 3.5
起泡排序过程

起泡排序的算法流程图如图 3.6 所示。Swap(L.R[j],L.R[j+1])表示将 L.R[j]和 L.R[j+1]进行交换。变量 lastExchangeIndex 记录在本趟排序中最后进行交换的记录的位置，其初值为 1。若在本趟排序结束后，其值仍为 1，则表明本趟排序无记录交换，即记录的关键字已经有序，整个排序过程可以结束。

起泡排序算法的示例代码如下。

```
void BubbleSort(SqList&L, int n) {
    i = n;
    while (i >1) {
        lastExchangeIndex = 1;
        for (j = 1;  j < i;  j++)
             if (L.R[j+1].key < L.R[j].key) {
                       Swap(L.R[j], L.R[j+1]);
        lastExchangeIndex = j; //记录进行交换的记录的位置
             } //if
        i = lastExchangeIndex; //本趟最后进行交换的记录的位置
      } // while
} // BubbleSort
```

起泡排序的时间性能分析：最好的情况是记录的关键字都已经有序，只进行一趟比较就结束，而且没有记录移动，即比较次数为 $n-1$，移动次数为 0。最坏的

情况是记录的关键字完全逆序，此时需要进行$(n-1)$趟起泡排序，且每次比较后都需要将 L.R[j]和 L.R[j+1]进行交换，因此这种情况的移动次数与比较次数一致，即比较次数为

$$\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2},\quad 2<i<n$$

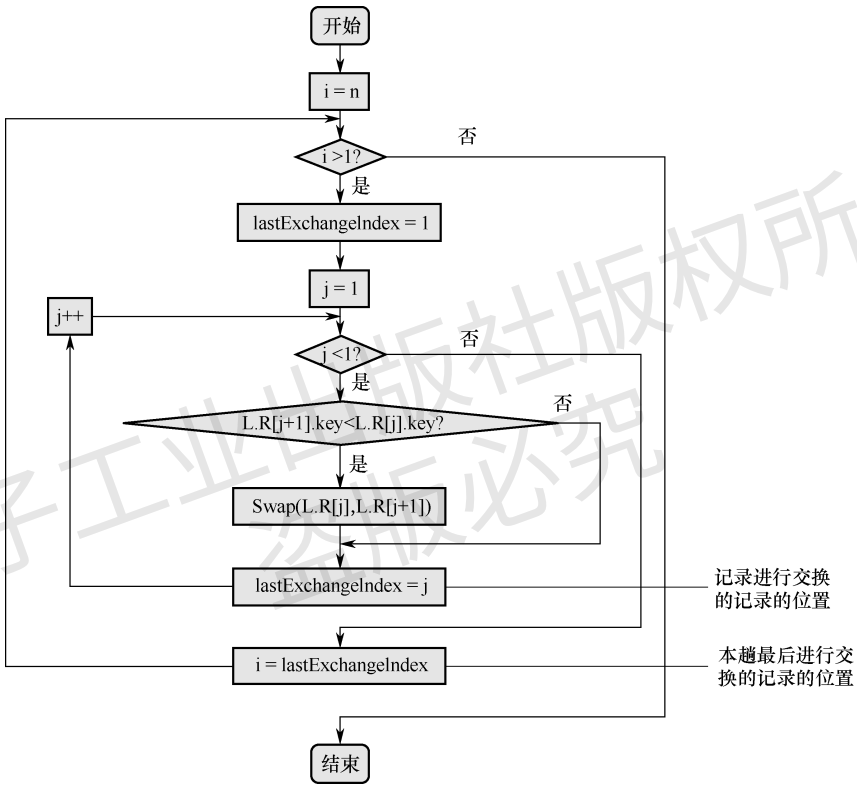

图 3.6
起泡排序的算法流程图

移动次数为

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

注意，移动次数前面乘以 3，是因为在计算机中当两个数据进行交换时，需要通过 3 次数据移动来实现。因此，在数据基本有序的情况下，起泡排序还是有优势的。

3.2.4　快速排序

顾名思义，快速排序是各种内部排序方法中效率最高的方法之一。其基本思想是：选择一个记录，并将其关键字作为枢轴，通常选取第一个记录的关键字作为枢轴。关键字小于枢轴的记录均移动至该枢轴记录之前，反之，关键字大于枢轴的记录均移动至该枢轴记录之后。这样，经过一趟快速排序处理（也称为一次划分）后，记录被分割为两部分，关键字比枢轴小的记录都在枢轴记录之前，关键字比枢轴大的记录都在枢轴记录之后。

假设待排序记录序列 L.R[s,⋯,t]经过一次划分后，将记录的无序序列 L.R[s,⋯,t]分割成两部分：L.R[s,⋯,i−1]和 L.R[i+1,⋯,t]。其中，L.R[s,⋯,i−1]中记录的关键字都小于枢轴，L.R[i+1,⋯,t]中记录的关键字都大于枢轴，即

$$L.R[j].key \leqslant L.R[i].key \leqslant L.R[p].key$$

$$(s \leqslant j \leqslant i-1) \qquad (i+1 \leqslant p \leqslant t)$$

其中，L.R[i].key 为枢轴。

例如，记录关键字序列为 52, 49, 80, 36, 14, 58, 61, 97, 23, 75，完成一趟快速排序的过程如图 3.7 所示。

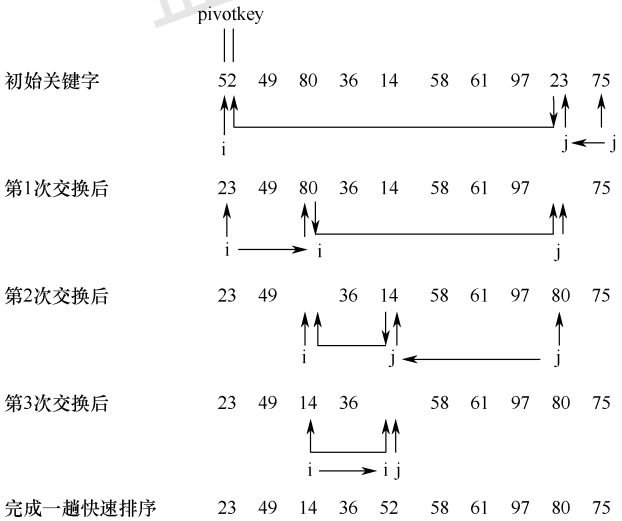

图 3.7
完成一趟快速排序的过程

经过一次划分，关键字序列分割成以枢轴为界的两部分，23,49,14,36,(52),58,61,97,80,75。在调整过程中，定义了两个指针 low 和 high，初值分别为 s 和 t，在排序过程中，high 逐步向前移动，并保证 L.R[high].key＞＝枢轴，low 逐步向后移动，并保证 L.R[low].key＜＝枢轴，两者交替进行。当 high=low 时，表示一趟排序结束，此时 low 的位置就是枢轴记录的位置。快速排序一次划分的算法流程图如图 3.8 所示。

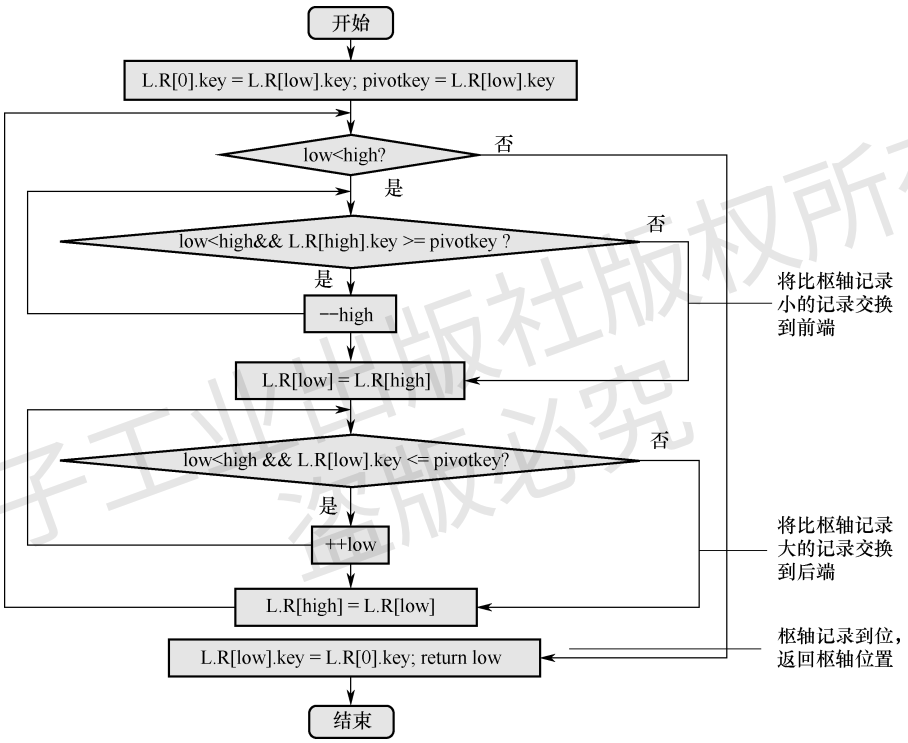

图 3.8
快速排序一次划分的算法流程图

快速排序算法示例代码如下。

```
int Partition (SqList&L, int low, int high) {
    L.R[0].key = L.R[low].key;  pivotkey = L.R[low].key;  //枢轴
    while (low<high) {  //从表的两端交替地向中间扫描
      while(low<high&& L.R[high].key>=pivotkey)
          -- high;    //将比枢轴记录小的记录交换到前端
      L.R[low] = L.R[high];
      while (low<high && L.R[low].key<=pivotkey)
```

```
            ++ low;     //将比枢轴记录大的记录交换到后端
          L.R[high] =L.R[low];
        }
        L.R[low].key = L.R[0].key;
        return low;     //枢轴记录到位，返回枢轴位置
    }// Partition
```

在对无序的记录序列进行一次划分后，整个记录以枢轴为界分割成两部分，前半部分的关键字都比枢轴小，后半部分的关键字都比枢轴大。然后对记录序列的前半部分和后半部分重复这个过程，即对其前、后两部分再分别进行快速排序，其过程如图 3.9 所示。

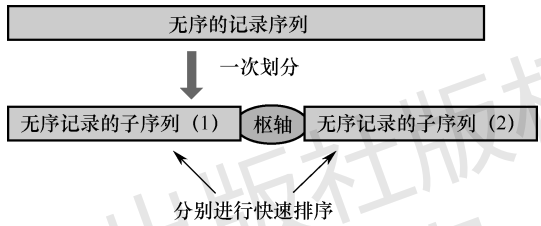

图 3.9
对记录序列前、后两部分分别进行快速排序的过程

完整的快速排序算法的示例代码如下。

```
void QSort (SqList&L, int low, int high ) {
// 对记录序列 L.R[low···high]进行快速排序
     if (low < high){  //长度大于 1
        pivotloc = Partition(L.R, low, high);
        //对 L.R[low···high] 进行一次划分，返回枢轴位置 pivotloc
        QSort(L.R, low, pivotloc-1);    //对前半部分序列进行递归排序
        QSort(L.R, pivotloc+1, high); //对后半部分序列进行递归排序
     }
} // QSort
```

从前面对快速排序的介绍可以看出，在经过一次划分后，枢轴记录把整个记录序列分割成前、后两部分，然后对这两部分重复同样的过程，这就是递归算法。关于递归和递归算法，将在本书第 5 章中进行介绍。

3.2.5　简单选择排序

选择排序的基本思想是：每趟排序均在(n−i+1)个记录中选取关键字最小的记

录，作为有序序列中的第 i 个记录，其中 i=1,2,⋯,n−1。首先选择最小的记录作为第 1 个记录，然后选择次最小的记录移至第 2 个位置，以此类推。简单选择排序是选择排序中较为简单的一种，在某些特殊情况下，它也是一种比较高效的排序方法，其特点是移动次数较少。第 i 趟简单选择排序过程如图 3.10 所示。

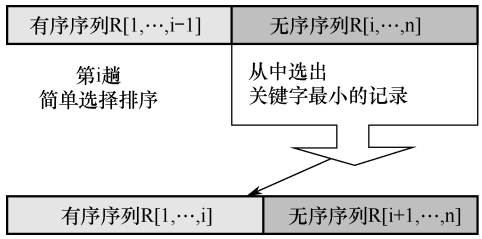

图 3.10
第 i 趟简单选择排序过程

在第 i 趟简单选择排序过程中，从 L.R[i,⋯,n]这些无序记录中，选择出关键字最小的记录，并将其移至第 i 个位置，即 L.R[i]。

简单选择排序的算法流程图如图 3.11 所示。

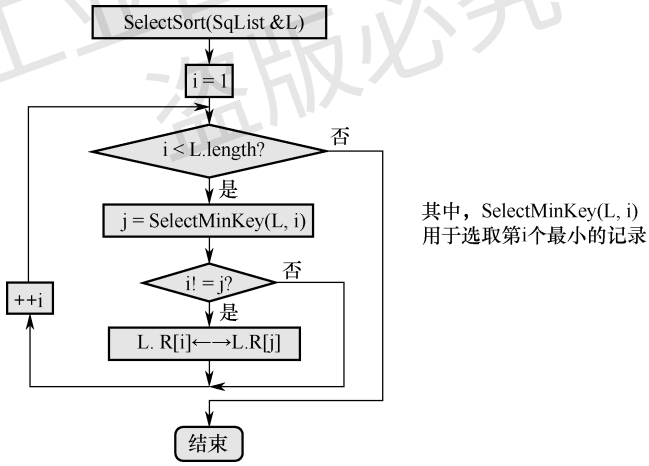

图 3.11
简单选择排序的算法流程图

简单选择排序算法的示例代码如下。

```
void SelectSort (SqList &L ) {
    //对顺序表&L 进行简单选择排序
   for (i=1; i<L.length; ++i) {   //选择第 i 个小的记录，并交换到位
```

```
        j = SelectMinKey(L, i);
              //在 L.R[i··· L.length] 中选择关键字最小的记录
        if (i!=j) L.R[i]⟷L.R [j];   //与第 i 个记录交换，移动次数最小值为 0，
                                        最大值为 3(n-1)
                    }
    } // SelectSort
```

简单选择排序的时间性能分析：从算法示例代码容易看出，其移动次数最小值为 0，最大值为 3(n−1)。然而，无论记录的初始排列如何，需进行的关键字的比较次数均相同，即为

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

所以，相比而言，简单选择排序的移动次数是最少的。在排序过程中，如果希望移动次数尽量少，那么可以选择简单选择排序。

3.3　奖学金竞争问题的求解

3.3.1　问题分析

根据 3.1 节中的问题描述，学生只要符合条件就可以获得奖学金，每种奖学金的获奖人数均没有限制，每名学生可以同时获得多种奖学金。问题的处理包括两个方面：①计算每名学生应得的各种奖学金，并求每名学生获得奖学金的总额；②将每名学生获得的奖学金总额按由高到低的顺序进行排序，并取前三名学生的信息。

3.3.2　算法分析

算法处理过程包括以下三步：

（1）输入所有学生的信息。包括学生自身信息和所涉及的奖学金的信息。

（2）统计每名学生获得奖学金的情况。

（3）将所有学生的奖学金总额按由高到低的顺序进行排序，并按要求输出排序后的前三名学生的信息。

本问题涉及的数据包括：学生姓名、期末平均成绩、班级评议成绩、是否是学生干部、是否是西部地区的学生，以及是否发表论文。数据元素之间是一种线性结构，考虑到学生人数是相对稳定的，故采用顺序存储结构来存放全部学生信息。

3.3.3 算法设计

计算所有学生奖学金的算法流程图如图 3.12 所示，其中 N 代表学生总数。

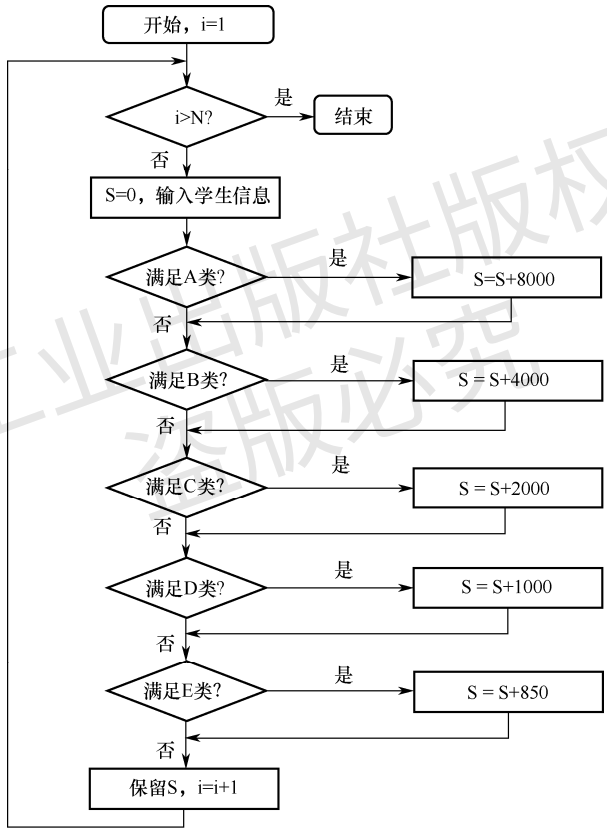

图 3.12
计算所有学生奖学金的算法流程图

计算完每名学生的奖学金总额后，就可以按获得奖学金总额由高到低的顺序进行排序。具体排序算法的示例代码如下。

```
    void InsertionSort (decide(struct STUDENT[]) {
```

```
    //采用直接插入排序
     for ( k=2; k<=N; ++k )
      if (stu[k].scholarship < stu[k-1].scholarship) {
          stu[0] = stu[k];          //复制为监视哨
          for ( j=k-1; stu[0].scholarship < stu[j].scholarship ;  -- j )
            stu[j+1] = stu[j];      //记录后移
          stu[j+1] = stu[0];        //插入到正确位置
      }
  } // InsertSort
```

本算法采用直接插入排序。总体排序完毕后，最后输出排序后的前三名学生的信息即可。

对于“哪三名学生获得的奖学金总额最高？”的问题，也可以采用其他思路。例如，不进行全部排序，只进行局部排序，即只求出获得奖学金总额最高的前三名学生即可。这时可以采用冒泡排序或简单选择排序，这两种方法都只进行前三趟排序就可以解决该问题。

3.4　应用

3.4.1　荷兰国旗问题

1．问题描述

现有红、白、蓝三种不同颜色的小球，乱序排列在一起，请重新将这些小球排序，使得相同颜色的小球分别放在一起。这个问题之所以称为荷兰国旗问题，是因为可以将红、白、蓝三色小球想象成条状物，在有序排列后，正好组成了荷兰国旗。

2．问题分析

本问题可以视为一个数组排序问题。数组分为前部、中部和后部三个部分，每个元素（红、白、蓝分别对应 0、1、2）必属于其中之一。由于红、白、蓝三色小球的数量并不一定相等，所以这三个部分不一定是等分的。思路如下：将红色（由 0 表示）元素和蓝色（由 2 表示）元素分别排在数组的前部和后部，白色

（由 1 表示）元素自然就在中部了，这就满足了排序的要求。

3．算法设计

设置两个标志位 begin 和 end，开始时分别指向该数组的开头和末尾，标志位 current 指向当前遍历位置，其初值为数组开始位置。

（1）若 current 遍历到的位置值为 0，则说明它一定属于前部，就与 begin 位置的值进行交换，然后 current 和 begin 的值各自加 1（表示前边的位置已经排好）。

（2）若 current 遍历到的位置值为 1，则说明它一定属于中部，根据问题分析，中部的数据不动，current 直接加 1。

（3）若 current 遍历到的位置值为 2，则说明它一定属于后部，就与 end 位置的值进行交换。由于交换后 current 指向的位置可能是属于前部的，若此时 current 前进，则会导致该位置不能被交换到前部，因此，此时 current 不动，end 值减 1，即向前退一个位置。

（4）重复此过程，直至 current=end。

4．算法流程图与示例代码

根据前面的算法设计，荷兰国旗问题的算法流程图如图 3.13 所示。

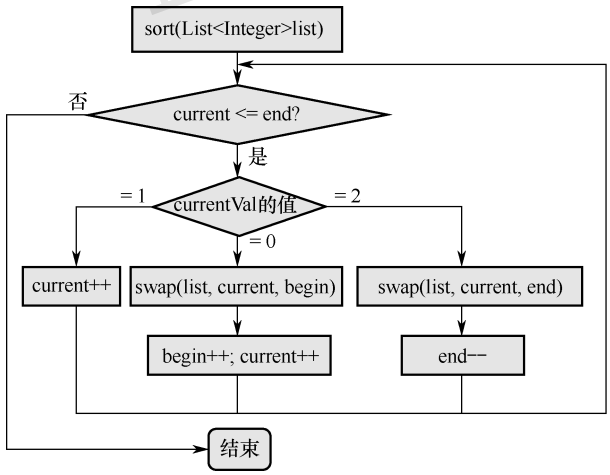

图 3.13
荷兰国旗问题的算法流程图

根据荷兰国旗问题的求解方法，其基本思想是快速排序，实际上求解该问题是快速排序的一个灵活应用。该问题的示例代码如下。

```
public void sort(List<Integer> list)
{
    int size = list.size(); int begin = 0;
    int end = size - 1; int current = 0;
    while(current < = end)  //还未结束
    {  int currentVal = list.get(current);
      if(currentVal =0)
         {  swap(list, current, begin);
            begin++;  current++;
           }
      else if(currentVal = 2 )
           { swap(list, current, end);
             end--;
           }
           else
           { current++;
           }
           }
}
```


3.4.2　货箱移动问题

1．问题描述

某家快递公司的员工得到了一项任务，要求将若干个货箱按照发货时间排放。虽然比较发货时间很容易（对照标签即可），但是将两个货箱交换位置则很困难（移动很麻烦），这是因为仓库已经快满了，只有一个空闲仓位。请问该员工应怎样完成这项任务？

2．问题分析

问题要求：找到一种合适的排序算法来解决货箱移动问题。由于货箱交换位置较为困难，因此要求货箱的移动次数是最少的，即在排序中尽量减少货箱交换次数，同时要考虑到只有一个空闲仓位的情况。因此，在货箱移动过程中，应该选取交换次数少且占用额外空间最少的方法。

货箱移动过程为：首先找到发货时间最近的货箱，将它与第一个货箱交换位置。之后，在剩下的货箱中找到发货时间最近的，将它与第二个货箱交换位置。重复此过程，直到所有货箱按发货时间的先后完成排序。所以这个问题用简单选择排序的方法就可以处理。

3.5 总结与思考

本章通过“谁能拿到奖学金”案例引出了排序的概念。简单介绍了排序的基本知识，对直接插入排序、起泡排序、快速排序和简单选择排序的排序过程、算法及特点和适用情况进行讲解。通过对“谁能拿到奖学金”问题的求解，以及对荷兰国旗问题、货箱移动问题的分析及算法设计，进一步掌握所学的排序知识。

思考题：

1. 发现你身边与排序相关的问题。若已有解决方法，则给出算法设计（算法流程图）；否则给出你的解决方法。

2. 已知线性表$(a_1,a_2,a_3,\cdots,a_n)$按顺序存于内存，每个元素都是整数，试设计算法，用最短的时间把所有负数都移动到所有正数之前。例如，将$(x,-x,-x,x,x,-x,\cdots,x)$变为$(-x,-x,-x,\cdots,x,x,x)$。

3. 某位老板有 n 个金块，他要找出其中最重和最轻的金块。假设有一台比较质量的仪器，每次可以比较两个金块的质量，思考如何用尽量少的比较次数找出最重和最轻的金块。