实验一 分治策略

实验目的

理解分治法的算法思想，阅读实现书上已有的部分程序代码并完善程序，加深对分治法的算法原理及实现过程的理解。

实验内容

一、用分治法实现一组无序序列的**两路合并排序**和**快速排序**。要求清楚合并排序及快速排序的基本原理，编程实现分别用这两种方法将输入的一组无序序列排序为有序序列后输出。
二、采用基于“五元中值组取中值分割法”（median-of-median-of-five partitioning）的线性时间选择算法，找出 N 个元素集合 S 中的**第 k 个最小的元素**，使其在**线性时间**内解决。（**参考教材 5.5.3 节**）

实验步骤

一、两路合并排序和快速排序

1、排序是数据处理中常用的重要手段，是指将一个元素序列调整为按指定关键字值的递增（或递减）次序排列的有序序列。

用分治法求解排序问题的思路是，按某种方式将序列分成两个或多个子序列，分别进行排序，再将已排序的子序列合并成一个有序序列。

合并排序和快速排序是两种典型的符合分治策略的排序算法。

2、如果采用顺序存储的可排序表作为算法实现的数据结构，则需要定义一个可排序表类 SortableList，两路合并算法和快速排序算法均由定义在该类上的函数实现。

```
class SortableList
{
public:
    SortableList(int mSize)
    {
        maxSize=mSize;
        l=new int[maxSize];
        n=0;                          //数组中已有元素个数
    }
    ~SortableList()
    {
        delete []l;
    }
```

```
    void SortableList::Input();
    void SortableList::Output();
    ……
private:
    ……
    int *l;
    int maxSize;
    int n;
};
```

其中 Input 函数和 Output 函数分别用于向可排序表中输入待排序序列，以及输出已经排序好的序列，请自行补充完成。

3、结合书上已有的程序代码，使用分治法的两路合并排序算法，实现对初始序列的排序。

两路合并排序算法的基本思想是：将待排序元素平分成大小大致相同的两个子序列，然后对每个子序列分别使用递归的方法进行两路合并排序，直到子序列长度变为 1，最后利用合并算法将得到的已排序好的两个子序列合并成一个有序的序列。

两路合并排序算法的核心部分是将子问题的解组合成原问题解得合并操作。常用的操作是新建一个序列，序列的大小等于要合并的两个子序列的长度之和。比较两个子序列中的最小值，输出其中较小者到新建的序列中，重复此过程直到其中一个子序列为空。如果另一个子序列中还有元素未输出，则将剩余元素依次输出到新建序列中即可。最终得到一个有序序列。

因此类中应定义成员函数 MergeSort 来完成递归两路合并排序的调用和成员函数 Merge 来完成两个子序列的合并操作：

```
class SortableList
{
public:
    ……
    void SortableList::MergeSort();
private:
    void MergeSort(int left,int right);
    void Merge(int left,int mid,int right);
    ……
};
```

利用递归调用函数 MergeSort(int left,int right)，将待排序元素序列一分为二，对得到的两个子序列分别排序。如此细分下去，直到子序列的长度不超过 1 时，子序列自然有序。

```
void SortableList::MergeSort()
{
    MergeSort(0,n-1);
}
void SortableList::MergeSort(int left,int right)
{
    if (left<right)
    {
```

```
        int mid=(left+right)/2;
        MergeSort(left,mid);
        MergeSort(mid+1,right);
        Merge(left,mid,right);
    }
}
```

当分解所得的子序列已有序时，再调用 Merge(int left,int mid,int right)函数依次将两个有序子序列合并成一个有序子序列。

```
void SortableList::Merge(int left,int mid,int right)
{
    int *temp=new int[right-left+1];
    int i=left,j=mid+1,k=0;
    while((i<=mid)&&(j<=right))
        if (l[i]<=l[j]) temp[k++]=l[i++];
        else temp[k++]=l[j++];
    while (i<=mid) temp[k++]=l[i++];
    while (j<=right) temp[k++]=l[j++];
    for (i=0,k=left;k<=right;) l[k++]=temp[i++];
}
```

注意：在合并排序的 Merger(int left,int mid,int right)函数中，一般需要使用与原序列相同长度的辅助数组 temp，因此需要的额外辅助空间为 O(n)。

在 main 函数中调用 SortableList 类中的成员函数 void SortableList::MergeSort()，实现可排序表上的两路合并排序。

4、结合书上已有的程序代码，使用分治法的快速排序算法，实现对初始序列的排序。

快速排序算法的基本思想是：

（1）在待排序序列 K[left:right]上选择一个基准元素（通常是最左边的元素 K_{left}），通过一趟分划操作将序列分成左右两个子序列，左子序列中所有元素都小于等于该基准元素，有子序列中所有元素都大于等于该基准元素。

则当前基准元素所在的位置位于左、右子序列的中间，即是其排序完成后的最终位置。

（2）通过递归调用，对左子序列和右子序列再分别进行快速排序算法的调用。

（3）由于每一趟分划结束后，左子序列中的元素均不大于基准元素，右子序列中的元素均不小于基准元素。而每次分划后，对分划得到的左、右子序列的快速排序又均是就地进行，所以一旦左、右两个子序列都已分别排好序后，无需再执行任何计算，整个序列就是所要求的有序序列了。

因此类中应定义成员函数 QuickSort 来完成递归快速排序算法的调用和成员函数 Partition 来完成每一趟分划操作：

```
class SortableList
{
    ……
    void SortableList::QuickSort();
private:
    void Swap(int i,int j);                    //交换下标为 i 和 j 的数组元素
    void QuickSort(int left,int right);
```

```
    int Partition(int left,int right);
    ……
};
void SortableList::Swap(int i,int j)
{
    int c=l[i];
    l[i]=l[j];
    l[j]=c;
}
```

递归调用函数 QuickSort(int left,int right) 每次选择序列最左边的元素 l[left]作为主元，通过调用 Partition(int left,int right)函数进行一趟分划操作，将下标在[left,right]范围内的序列分成两个子序列。

```
int SortableList::Partition(int left,int right)
{
    int i=left,j=right+1;
    do{
        do i++;  while (l[i]<l[left]);
        do j--;   while (l[j]>l[left]);
        if (i<j) Swap(i,j);
    }while (i<j);
    Swap(left,j);
    return j;
}
```

然后分别再递归调用自身，对这两个子序列实施快速排序。一旦左右子序列已经有序，则无需再进行额外处理，整个序列就自然有序了。

```
void SortableList::QuickSort()
{
    QuickSort(0,n-1);
}
void SortableList::QuickSort(int left,int right)
{
    if (left<right)
    {
        int j=Partition(left,right);
        QuickSort(left,j-1);
        QuickSort(j+1,right);
    }
}
```

5、比较合并排序和两种算法的异同。

问题分解过程：

- 合并排序——将序列一分为二即可。 （十分简单）
- 快速排序——需调用 Partition 函数将一个序列划分为子序列。(分解方法相对较困难)

子问题解合并得到原问题解的过程：

- 合并排序——需要调用 Merge 函数来实现。（Merge 函数时间复杂度为 O(n)）
- 快速排序——一旦左、右两个子序列都已分别排序，整个序列便自然成为有序序列。（异常简单，几乎无须额外的工作，省去了从子问题解合并得到原问题解的过程）

二、线性时间选择算法——寻找第 k 个最小元

1、五元中值组取中值分割法基本思路

◆ 算法思想：

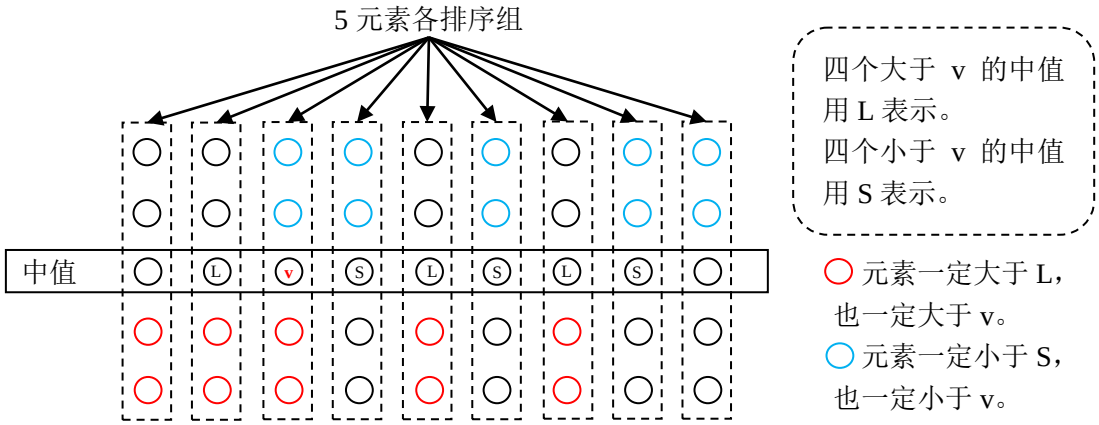
※假设 N 可以被 5 整除，因此不存在多余元素；

※假设 N/5 为奇数，这样集合 M 就包含奇数个元素（以保持对称性）；

※因此，方便起见可假设 N 为 10k+5 的形式。并且假设所有元素均互异。即使 N 不是 10k+5 的形式，类似的论证仍可进行而不影响基本结果。

1）把 N 个元素分成⌊N/5⌋组，每组 5 个元素，忽略其中 4 个元素，保留每组的中值，得到⌊N/5⌋个中值的集合 M。

2）递归调用自身，从中值的样本 M 中再找出中值（作为枢纽元 v）。选择方式如图所示：



对于 N=10k+5 的一般情形，有 3k+2 个元素保证大于 v，有 3k+2 个元素保证小于 v。因此除去 v 自身，接下来的递归调用最多可以包含 7k+2<0.7N 个元素。

◆ **寻找枢纽元的时间分析：**

寻找 5 元素的中值为常数时间，这样的运算进行⌊N/5⌋次，因此这一步花费 O(N)时间。

然后若采用常见排序算法在有⌊N/5⌋个元素的中值组中再寻找中值（枢纽元），时间复杂度仍然要达到 O(⌊N/5⌋log⌊N/5⌋)=O(NlogN)。

所以需要递归的调用自身，寻找枢纽元的时间为 **T（⌊N/5⌋）+O(N)**。

2、快速选择第 k 小元素

◆ **基本思路：**

1）枢纽元 v 将集合分割成 S_1 和 S_2，规模分别约为 0.3N 和 0.7N。

2）如果 k≤$|S_1|$，那么第 k 个最小元必然在 S_1 中。如果 k=1+$|S_1|$，那么枢纽元就是第 k 小的元素，返回答案。如果 k>1+$|S_1|$，第 k 个最小元就在 S_2 中，它是 S_2 中的第（k-$|S_1|$-1）个最小元。

3）然后对第 k 小元素所在的部分再进行递归调用(规模为 0.3N 或 0.7N)。

◆ **快速选择的时间分析：**

对规模最多为 0.7N 的子表继续求解，寻找第 k 小元素的时间 **≤T(0.7N)**。

3、寻找第 k 小元素时间分析

总的时间递推式为：**T(N) ≤O(N)+T(N/5)+T(0.7N)。**

有**定理：如果** $\sum_{i=1}^{k}\alpha_i<1$ **，则方程 T(N)=** $\sum_{i=1}^{k}T(\alpha_i N)+O(N)$ **的解为 T(N)=O(N)。**

因此，寻找第 k 小元总的时间为 **O(N)线性时间**。

4、递归调用程序主体部分（提示）

```
int Select(int k,int left,int right,int r)
{   //每个分组r个元素，寻找第k小元素
    int n=right-left+1;
    if (n<=r)   //若问题足够小，使用直接插入排序
    {
        InsertSort(left,right);
        return left+k-1;    //取其中的第k小元素，其下标为left+k-1
    }
    for (int i=1;i<=n/r;i++)
    {
        InsertSort(left+(i-1)*r,left+i*r-1);
                                //二次取中规则求每组的中间值
        Swap(left+i-1,left+(i-1)*r+(int)ceil((double)r/2)-1);
                                //将每组的中间值交换到子表前部集中存放
    }
    int j=Select((int)ceil((double)n/r/2),left,left+(n/r)-1,r);
                                            //求二次中间值，其下标为j
    Swap(left,j);   //二次中间值为枢纽元，并换至left处
    j=Partition(left,right);    //对表（子表）进行分划操作
    if (k==j-left+1) return j;  //返回第k小元素下标
    else if (k<j-left+1)
                return Select(k,left,j-1,r);//在左子表求第k小元素
          else
                return Select(k-(j-left+1),j+1,right,r);
                                    //在右子表求第k-(j-left+1)小元素
        }
}
```

思考

1、在上述快速排序算法的执行过程中，跟踪程序的执行会发现，若初始输入序列递减有序，则调用 Partition 函数进行分划操作时，下标 i 向右寻找大于等于基准元素的过程中会产生下标越界，为什么？如何修改程序，可以避免这种情况的发生？

这是因为原有的程序在序列最右边未设置一个极大值作为哨兵，则下标 i 在向右寻找大于等于基准元素的过程中，一直没有满足条件的元素值出现，就一直不会停止，直至越界。

所以只要在序列的最后预留一个哨兵元素，将它的值设为极大值∞就可以解决：

```
const int INF=2147483647;    //定义一个极大值∞
l=new int[maxSize+1];        //预留最后一个哨兵的位置
void SortableList::Input()
```

```
{    ……
     l[n]=INF;                    //最后一个元素为哨兵∞
}
```

2、分析这两种排序算法在最好、最坏和平均情况下的时间复杂度。

两路合并排序：最好、最坏、平均情况下的时间复杂度均为 O(nlogn)。

快速排序：最好、平均情况下的时间复杂度为 O(nlogn)，最坏情况下为 $O(n^2)$。

3、当初始序列是递增或递减次序排列时，通过改进主元（基准元素）的选择方法，可以提高快速排序算法运行的效率，避免最坏情况的发生。

有三种主元选择方式。

一是取 $K_{(left+right)/2}$ 为主元；

二是取 left~right 之间的随机整数 j，以 K_j 作为主元；

三是取 K_{left}、$K_{(left+right)/2}$ 和 K_{right} 三者的中间值为主元。

试选择其中的一种，在原程序的基础上修改实现。下面给出第二种主元选择方式的具体实现：

随机数的产生是由 srand 函数以 time 函数值（即当前时间）作为种子，rand 函数产生一个 0-RAND_MAX 范围内的随机数。因此文件的开头应包含两个头文件 time.h 和 stdlib.h。

为了不改动原有程序的基本结构，在原有程序上增加一个 RandomizedPartition 函数。由 QuickSort 先调用该函数，产生一个 left~right 范围内的随机下标 i，将该下标处的元素 K_j 和原有的主元 K_{left} 交换，然后照常调用原有的 Partition 函数即可。

```
void SortableList::QuickSort(int left,int right)
{
     ……
     int j=RandomizedPartition(left,right);  //调用 RandomizedPartition 函数
     ……
}
int SortableList::RandomizedPartition(int left, int right)
{
     srand( (unsigned)time(NULL));     //用当前时间作为种子
     int i=rand()%(right-left)+left;   //产生一个 left~right 范围内的随机数
     Swap(i,left);
     return Partition(left, right);    //调用 Partition 函数
}
```