第18章

分而治之算法

分而治之算法思想：

- 1、把问题分解成两个或多个更小的问题；
- 2、分别解决每个小问题；
- 3、把各小问题的解答组合起来，即可得到原问题的解。

分而治之算法思想：

- 问题的并行化：分治算法把一个问题实例分解为若干个小型而独立的实例，从而可以在并行计算机上执行，那些小型而独立的实例可以在并行计算机的不同处理器上完成

分而治之算法思想举例

- 【找出假币】
- 一个袋子有16个硬币，其中一个是假币，并且假币比真币轻，只有一台机器可用来比较两组硬币的重量，如何找出？

本章学习内容

- 18.2.2　归并排序
- 18.2.3　快速排序
- 18.2.4　选择

18.2.2 归并排序(Merge Sort)

- 利用分而治之方法进行排序算法：
- 将n 个元素按非递增顺序排列.
  - 若n 为1，算法终止；
  - 否则
    - 将这一元素集合分割成两个或更多个子集合
    - 对每一个子集合分别排序
    - 将排好序的子集合归并为一个集合

分而治之排序-1

- 将n 个元素的集合分成两个子集合A和B 。如何进行子集合的划分。
- 1.
  - 把前面n-1个元素放到第一个子集A中，最后一个元素放到子集B中。按照这种方式对A递归地进行排序。由于B仅含一个元素，所以它已经排序完毕，在A排完序后，用程序2-10中函数insert将A和B合并起来。
  - 插入排序(insertionSort程序2-15)的递归算法
  - 时间复杂性 :$O(n^2)$

分而治之排序-1

- 回忆插入排序：从第一个元素构成的单元数组开始，不断实行插入操作。插入第二个元素，得到2个元素的有序数组；插入第三个元素，得到3个元素的有序数组

分而治之排序-2

- 2.
  - 将含有最大值的元素放入B(用函数Max(见程序1-37)来找出最大元素)，剩下的放入A中。然后A被递归排序。合并排序后的A和B，只需要将B添加到A中即可。
  - 选择排序selectionSort(见程序2-7)的递归算法
  - 时间复杂性: $\Theta(n^2)$

分而治之排序-2

- 2. 回忆选择排序：
  - 首先找到最大元素，把它移动到a[n-1].然后在余下的n-1个元素中找到最大的元素，把它移动到a[n-2]

分而治之排序-3

- 3.
  - 以上方法中，含有最大值的元素集合B(用冒泡过程(见程序2-9)来寻找最大元素并把它移到最右边的位置)
  - 冒泡排序bubbleSort(见程序2-9)的递归算法。
  - 时间复杂性: $\Theta(n^2)$

分而治之排序-4

- 上述分割方案将n 个元素分成两个极不平衡的集合A和B。A有n-1个元素，而B仅含一个元素。
- 4. 平衡分割法的情况：
  - A:含有***n/k*** 个元素
  - B:其余的元素
  - 递归地使用分而治之方法对A和B进行排序
  - 将排好序的A和B归并为一个集合。

分而治之排序算法的伪代码

```
template<class T>
void sort( T E, int n)
{ / /对E中的n个元素进行排序， k为全局变量
    if (n>=k) {
        i = n/k;
        j = n-i;
        令A 包含E中的前i 个元素
        令B 包含E中余下的j 个元素
        sort(A, i);
        sort(B, j);
        merge(A, B, E, i, j); //把A和B合并到E
        }
    else 使用插入排序算法对E进行排序
}
```

分而治之排序算法的时间

- 设t(n) 为分而治之排序算法在最坏情况下所需花费的时间.

- $$t(n)=\begin{cases} d & n<k \\ t(n/k)+t(n-n/k)+cn & n\geq k \end{cases}$$

- 当 k=2，t(n)最小，t(n)= Θ(nlogn) .
- 归并排序(merge sort)，或**二路归并排序**(two-way merge sort) : **k=2的分而治之排序方法**.

归并排序算法

```
template<class T>
mergeSort( T *a, int left, int right)
{//对a[left:right]中的元素进行排序
    if (left < right) {//至少两个元素
    int middle = (left + right)/2; //中心位置
    mergeSort(a, left, middle);
    mergeSort(a, middle +1, right);
    merge(a, b, left, middle, right); //从a归并到b
    copy(b, a, left, right); //排序结果复制到a
    }
}
```

划分为两个子序列时，不需要分别复制，只需要简单记录它们在原始序列中的左右边界，然后将排序后的子序列归并到一个新数组b中，最后再将它们复制回a中

```
template<class T>
void merge(T c[], T d[], int startOfFirst, int endOfFirst,
                                    int endOfSecond)
{//把c[startOfFirst : endOfFirst]和c[endOfFirst+1 : endOfSecond]
 //归并到d[startOfFirst : endOfSecond].
    int  first = startOfFirst, // 第一段的游标
         second = endOfFirst+1, // 第二段的游标
         result = startOfFirst; // 结果段的游标
    //当两个被归并段都未处理完，则不断进行归并
    while ((first <= endOfFirst) && (second <= endOfSecond))
         if (c[first] <= c[second]) d[result++] = c[first++];
                                    else   d[result++] = c[second++];
    // 考虑余下的部分
    if  (first > endOfFirst)
           for (int q = second; q <= endOfSecond; q++)
                        d[result++] = c[q];
     else for (int q = first; q <= endOfFirst; q++)
                    d[result++] = c[q];
}
```

归并排序

- 长度为1的序列被归并为长度为2的有序序列；
- 长度为2的序列被归并为长度为4的有序序列；
- ……
- 长度为$n/2$的序列被归并为长度为n的序列。

归并排序示例

- 初始段a: [8], [4], [5], [6], [2], [1], [7], [3]
- 归并到b: [4, 8], [5, 6], [1, 2], [3, 7]
- 复制到a:[4, 8], [5, 6], [1, 2], [3, 7]
- 归并到b: [4, 5, 6, 8], [1, 2, 3, 7]
- 复制到a:[4, 5, 6, 8], [1, 2, 3, 7]
- 归并到b: [1, 2, 3, 4, 5, 6, 7, 8]
- 复制到a: [1, 2, 3, 4, 5, 6, 7, 8]

消除递归可改善性能，如何？

一种迭代算法：mergeSort实现

```
template<class T>
void mergeSort(T a[], int n)
{// 使用归并排序算法对a[0:n-1] 进行排序
    T *b = new T [n];
    int segmentSize = 1; // 段的大小
    while (segmentSize < n) {
        mergePass(a, b, segmentSize, n); // 从a归并到b
        segmentSize += segmentSize;
        mergePass(b, a, segmentSize, n); // 从b 归并到a
        segmentSize += segmentSize;
        }
}
```

一趟归并实现

```
template<class T>
void mergePass(T x[], T y[], int segmentSize, int n)
{   // 归并大小为segmentSize的相邻段
     int i=0;
     while (i <= n-2*segmentSize) {
          // 归并两个大小为segmentSize的相邻段
          merge(x, y, i, i+segmentSize-1, i+2*segmentSize-1);
          i=i+2*segmentSize;
          }
     // 剩下不足2*segmentSize个元素
     if  (i + segmentSize < n)
          merge(x, y, i, i+segmentSize-1, n-1);
     else   for (int j = i; j <= n-1; j++) // 把最后一段复制到y
                    y[j] = x[j];
}
```

自然归并排序

- 自然归并排序(natural merge sort)是基本归并排序(见程序18 -3)的一种变化。它首先对输入序列中已经存在的有序子序列进行归并。
- 例:4, 8, 3, 7, 1, 5, 6, 2
  - [4, 8], [3, 7], [1, 5, 6], [2]
  - [3, 4, 7, 8], [1, 2, 5, 6]
  - [1, 2, 3, 4, 5, 6, 7, 8]

18.2.3 快速排序(Quick Sort)

- 快速排序方法描述
  - //使用快速排序方法对$a[0:n-1]$排序
  - 从$a[0:n-1]$中选择一个元素作为*middle*，该元素为**支点**
  - 把余下的元素分割为两段*left*和*right*，使得*left*中的元素都不大于支点，而*right*中的元素都不小于支点
  - 递归地使用快速排序方法对*left*进行排序
  - 递归地使用快速排序方法对*right*进行排序
  - 所得结果为*left*+*middle*+*right*

快速排序实现

```
template<class T>
void quickSort(T a[], int n)
{// 对a[0:n-1] 进行快速排序
 if(n<=1) return;
//把最大元素移动数组右端,如果不满足，例如，当支点是最
    大元素时，第一个do循环语句的结果是左索引值大于n-1
 int max= indexOfMax(a, n);
 swap(a[n-1], a[max])
 quickSort(a, 0, n-2);
}
template<class T>
void quickSort(T a[], int leftEnd, int rightEnd)
{//排序a[leftEnd : rightEnd]，a[rightEnd+1]有最大关键值
     if (leftEnd >= rightEnd) return;
     int leftCursor = leftEnd, // 从左至右的游标
         rightCursor = rightEnd+1; // 从右到左的游标
     T  pivot = a[leftEnd];
```

```
// 把左侧不小于pivot的元素与右侧不大于pivot 的元素进行交换
while (true) {
    do {// 在左侧寻找不小于pivot 的元素
        leftCursor = leftCursor+1;
        } while (a[leftCursor] <  pivot);
    do {// 在右侧寻找不大于pivot 的元素
        rightCursor = rightCursor-1;
        } while (a[rightCursor] > pivot);
    if (leftCursor >= rightCursor) break; // 未发现交换对象
    Swap(a[leftCursor], a[rightCursor]);
    }
//设置pivot
a[leftEnd] = a[rightCursor];
a[rightCursor] = pivot;
quickSort(a, leftEnd, rightCursor-1); // 对左段排序
quickSort(a, rightCursor+1, rightEnd); // 对右段排序
}
```

快速排序示例

4, 8, 3, 7, 1, 5, 6, 2

快速排序习题

10，9，6，7，5，14，2，8

复杂性分析

- **空间复杂性：** 递归栈空间: O(n).
- **时间复杂性**
- 最坏情况:
  - 支点元素是数组中的最小元素或最大元素, *left* 总是为空(或right总是为空)
  - 时间： $\Theta(n^2)$
- 最好情况:
  - *left*和*right*中的元素数目大致相同。
  - 时间： **O(nlogn)**
- 快速排序的平均复杂性也是**Θ(nlogn).**

中值快速排序

- 中值快速排序(median-of-three quick sort) 算法有更好的平均性能。
  - 不必使用a[1eftEnd]做为支点
  - 取{a[1eftEnd],

    a[(1eftEnd +rightEnd)/2],

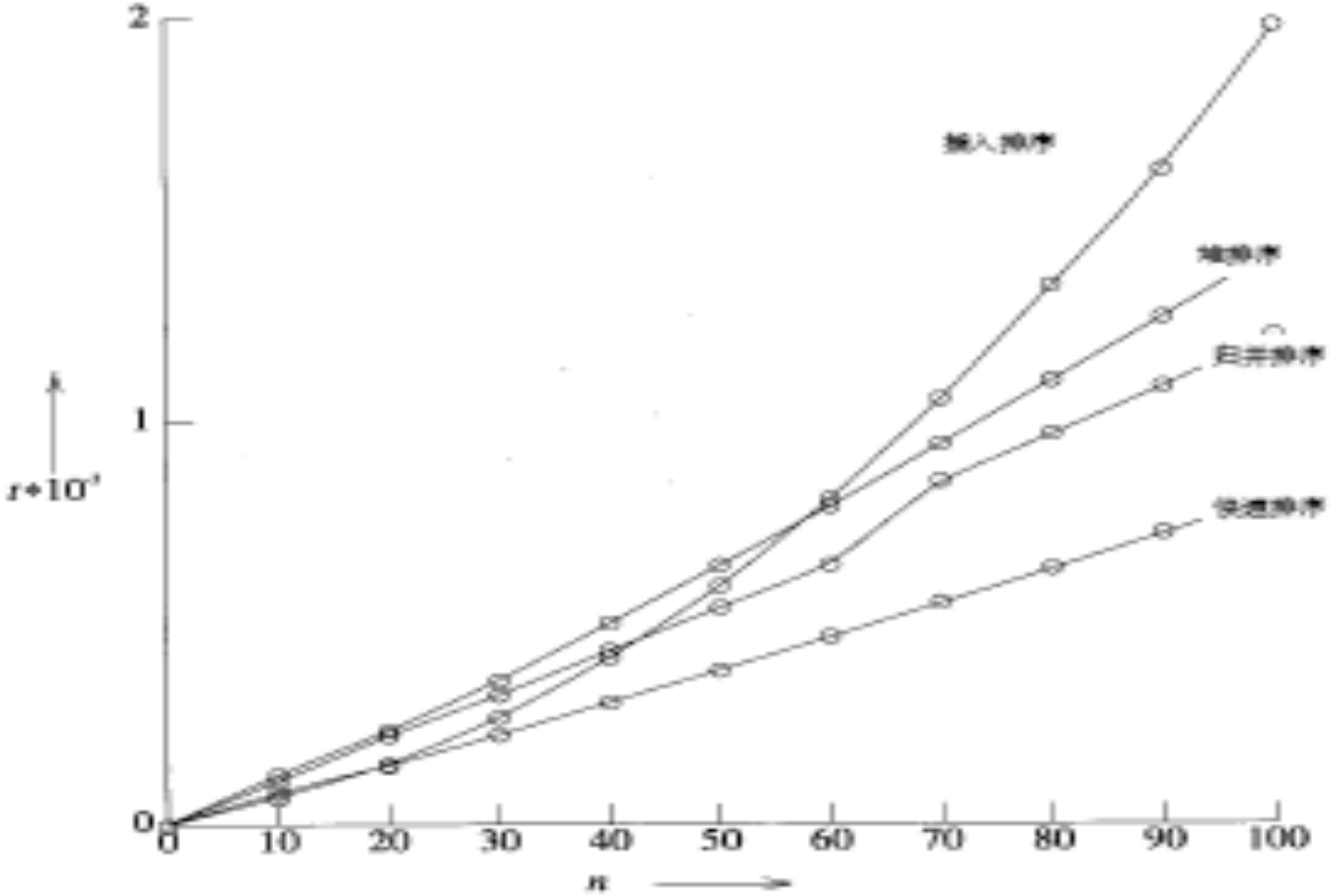
    a[rightEnd]}

    中大小居中的那个元素作为支点。

各种排序算法的比较

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
基数排序	n	n
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	nlogn	nlogn
归并排序	nlogn	nlogn
快速排序	n^2	nlogn

各排序算法平均时间曲线

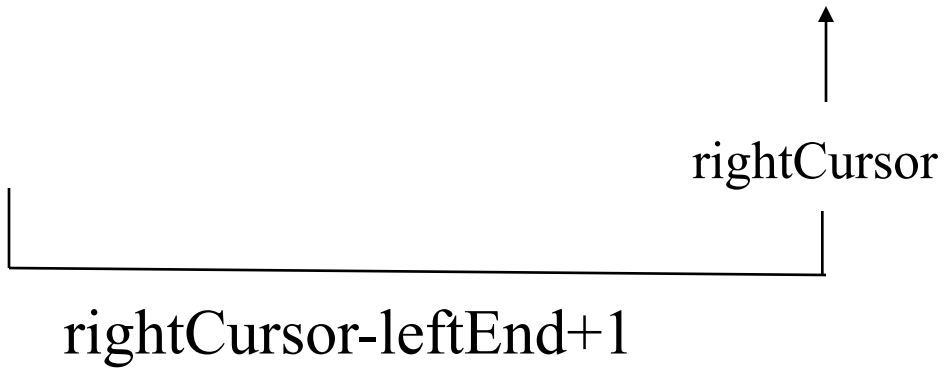


18.2.4 选择(Selection)

- 对于给定的n个元素的数组a[0:n-1]，要求从中找出第k小的元素。当a[0:n-1]被排序时，该元素就是a[k-1]。
- 方法1.
  - 对这n个元素进行排序(如使用堆排序式或归并排序)，
  - 取出a[k-1]中的元素
  - 时间:O(nlogn)

选择

- 方法2. 使用快速排序:
- a[leftEnd] a[leftEnd+1] …… a[ ] …… a[rightEnd]



rightCursor-leftEnd+1 = k:
rightCursor-leftEnd+1 > k:　左面部分第k小的元素
rightCursor-leftEnd+1 < k:　右面部分第
(k - (rightCursor-leftEnd+1))小的元素

程序18-7 寻找第k个元素

```
template<class T>
T select(T a[], int n, int k)
    {//返回a[0:n-1]中第k小的元素
    // 假定a[n] 是一个伪最大元素
    if (k < 1 || k > n) throw ……..;
    return select(a, 0, n-1, k);
    }

template<class T>
T select(T a[], int leftEnd, int rightEnd, int k)
{//在a[leftEnd : rightEnd]中选择第k小的元素
if (leftEnd >= rightEnd) return a[leftEnd];
int leftCursor = leftEnd, // 从左至右的游标
    rightCursor = rightEnd+1; // 从右到左的游标
```

```
T pivot = a[leftEnd];
// 把左侧>= pivot的元素与右侧<= pivot 的元素进行交换
while (true) {
    do {// 在左侧寻找>= pivot 的元素
        leftCursor=leftCursor+1;
        } while (a[leftCursor] < pivot);
    do {// 在右侧寻找<= pivot 的元素
        rightCursor=rightCursor-1;
        } while (a[rightCursor]>pivot);
    if (leftCursor>=rightCursor) break; // 未发现交换对象
    Swap(a[leftCursor], a[rightCursor]);
    }
if (rightCursor-leftEnd+1= =k) return pivot;
```

```
//设置pivot
a[leftEnd] = a[rightCursor];
a[rightCursor] = pivot;
// 对一个段进行递归调用
if (rightCursor-leftEnd+1 < k)
return select(a, rightCursor+1, rightEnd,
                        k-rightCursor+leftEnd-1);
else return select(a, leftEnd, rightCursor-1, k);
}
```

可以获得更好的平均性能，尽管该算法有一个比较差的渐近复杂性$O(n^2)$。