第3章

分治算法——分而治之

3.1 概述


视频讲解

3.1.1 分治算法的本质

分治算法，就是把一个复杂的大问题分成两个或更多个规模较小的相同子问题，子问题相互独立，递归求解各子问题，直到最后各子问题可以简单地直接求解为止，然后归并各子问题的解得原问题的解。

可见，分治算法本质是将一个难以直接解决的大问题，分解成一些规模较小的相同问题，以便各个击破，分而治之。

【例 3-1】 大整数乘法问题。

给定两个 n 位的大整数 A、B，求 A 与 B 的乘积。

大整数乘法问题，如果按照数学中的多位数乘法进行求解，用 B 的每一位乘以 A 的每一位，然后再错位相加得到结果。显然这种方法的时间复杂度为 $O(n^2)$。

我们来尝试设计一个分治算法：将整数 A 分成两个 $n/2$ 位的整数 A_1 和 A_2，将整数 B 分成两个 $n/2$ 位的整数 B_1 和 B_2，如图 3-1 所示。

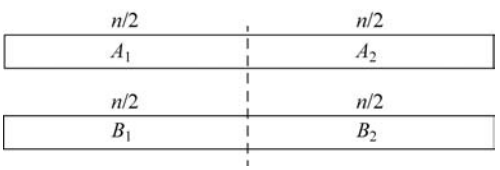

图 3-1 A 和 B 划分情况

$$\begin{aligned} A \times B &= (10^{n/2} \times A_1 + A_2) \times (10^{n/2} \times B_1 + B_2) \\ &= 10^n A_1 B_1 + 10^{n/2}(A_1 B_2 + A_2 B_1) + A_2 B_2 \end{aligned}$$

这样，两个 n 位数的大整数乘法就变成了 4 个 $n/2$ 为的大整数乘法。子问题规模变小了；与原问题相同，都是整数乘法；4 个子问题相互独立，互不影响。递归求解 4 个子问题，然后将递归的结果(4 个子问题的解)按照上式可以归并得到 A 与 B 的乘积。

该算法的时间复杂度在 $n>1$ 时，划分耗时 $O(1)$，将 4 个子问题的解归并为原问题的解时，乘以 10^n 可以认为是在数字后面补 n 个 0，耗时为 $O(n)$，所以递推方程如下：

$$T(n)=\begin{cases}O(1) & n=1\\ 4T(n/2)+O(n) & n>1\end{cases}$$

用主方法求解：$a=4,b=2,d=1,b^d=2^1=2,a>b^d$，所以 $T(n)=O(n^{\log_b a})=O(n^{\log_2 4})=O(n^2)$。

分治算法求解大整数乘法的阶与传统乘法的阶是一样的，均为 $O(n^2)$。根据主方法求解的过程，我们可以发现：减少 a 的值可以降低算法的阶。

现在将括号内加上 A_1B_1 和 A_2B_2，然后再减去 A_1B_1 和 A_2B_2 来降低 a 的值，即减少子问题的个数。

$$\begin{aligned}A\times B&=10^nA_1B_1+10^{n/2}(A_1B_2+A_2B_1)+A_2B_2\\&=10^nA_1B_1+10^{n/2}(A_1B_2+A_2B_1+A_1B_1+A_2B_2-A_1B_1-A_2B_2)+A_2B_2\\&=10^nA_1B_1+10^{n/2}[(A_1-A_2)(B_2-B_1)+A_1B_1+A_2B_2]+A_2B_2\end{aligned}$$

这样，A_1-A_2 最多是 $n/2$ 位，B_2-B_1 最多也是 $n/2$ 位，我们就将 4 个 $n/2$ 个子问题变成了 3 个最多 $n/2$ 位的子问题。算法的时间复杂度为 $T(n)=O(n^{\log_b a})=O(n^{\log_2 3})$。

【例 3-2】 最小值问题。

给定 n 个可以比较的元素，求它们的最小值。

最小值问题的分治算法：将 n 个元素平均分成两部分，递归求解两部分的最小值，比较两部分的最小值，取较小的作为 n 个元素的最小值。

在该算法中，两部分元素规模为 $n/2$，规模变小；都是在指定范围的元素中找最小值，与原问题相同，两部分元素互相独立。递归求解两个 $n/2$ 规模的子问题，将子问题的解归并成原问题的解：比较递归的结果，较小的值就是原问题的解。

算法分解消耗的时间为 $O(1)$，递归求解耗时为 $2T(n/2)$，将子问题的解归并成原问题的解耗时 $O(1)$。当规模为 1 时，最小值即是它本身，耗时 $O(1)$。时间复杂度递推方程如下：

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

用主方法求解该递推方程：$a=2,b=2,d=0,b^d=2^0=1,a>b^d,T(n)=O(n^{\log_b a})=O(n)$。

由上述两个例子，总结得到分治算法求解问题所具有以下 4 个基本特征。

(1) 问题的规模小到一定程度时，常数时间就能解决。

(2) 问题可以分解为若干个规模较小的相同子问题。

(3) 问题所分解出的各个子问题是相互独立的。

(4) 子问题的解能够归并为原问题的解。

上述的第(1)个特征是绝大多数问题都可以满足的，因为问题的计算复杂度一般是

随着问题规模的增大而增加；第(2)个特征是应用分治算法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第(3)个特征涉及分治算法的效率，如果各个子问题是不独立的，则分治算法要做许多不必要的工作——重复求解公共的子问题；第(4)个特征是关键，若子问题解不能归并得到原问题的解，则分解与递归求解子问题的工作都是徒劳。


视频讲解

3.1.2 分治算法的求解步骤

通常，分治算法的求解过程都要遵循两大步骤：分解和治理。

第一步，分解。

将问题分解为若干个规模较小、相互独立、与原问题形式相同的子问题。

那么，究竟该如何合理地对问题进行分解呢？应把原问题分解为多少个子问题才合适呢？每个子问题是否规模大致相等才为适当？这些问题很难给予肯定的回答。人们从大量的实践中发现，在用分治算法设计算法时，最好分解得到的子问题规模大致相等，即将一个问题分为规模大致相等的 k 个子问题(通常 $k=2$)。如最小值问题，把 n 个元素平均分成两部分，得到两个子问题；大整数乘法是将两个整数的位数分别分成大致相等的两部分，得到 4 个子问题。

这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它总是比子问题规模不等的做法要好。也有 $k=1$ 的划分，这仍然是把问题划分为两部分，取其中的一部分，而丢弃另一部分。例如，二分查找(折半查找)问题在采用分治算法求解时就是这样划分的。

子问题的个数直接影响分治算法的时间复杂度，比如大整数乘法问题，我们通过恒等变形，将子问题的个数从 4 个减为 3 个，算法的时间复杂度从 $O(n^2)$ 降为 $O(n^{1.58})$。

【例 3-3】 矩阵相乘问题。

给定矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，均为 n 阶矩阵，$n=2^k$，即 n 是 2 的幂，求 $\boldsymbol{A}\times\boldsymbol{B}$。

矩阵乘法是线性代数中最常见的运算之一，它在数值计算、图像处理、数据挖掘等有广泛的应用。如回归、聚类、主成分分析、决策树等挖掘算法常涉及大规模矩阵运算。

若 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是两个 $n\times n$ 的矩阵，则它们的乘积 $\boldsymbol{C}=\boldsymbol{AB}$ 同样是一个 $n\times n$ 的矩阵，按照传统的矩阵乘法，需要 n^3 的乘法次数，时间复杂度为 $O(n^3)$。

矩阵相乘的分治算法，将矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 均划分为 4 个 $n/2$ 阶矩阵，结果矩阵也为 4 个 $n/2$ 阶矩阵。划分情况如下所示：

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

其中，$C_{11}=A_{11}B_{11}+A_{12}B_{21}$，$C_{12}=A_{11}B_{12}+A_{12}B_{22}$，$C_{21}=A_{21}B_{11}+A_{22}B_{21}$，$C_{22}=A_{21}B_{12}+A_{22}B_{22}$。

当规模为 1 时，分治算法消耗的时间为常数 $O(1)$；当问题规模大于 1 时，分解耗时 $O(1)$，递归子问题耗时 $8T(n/2)$，将子问题的结果矩阵相加耗时 $O(n^2)$，则算法的时间复杂度递推方程如下：

$$T(n)=\begin{cases} O(1) & n=1 \\ 8T(n/2)+O(n^2) & n>1 \end{cases}$$

利用主方法求解：$a=8,b=2,d=2,b^d=2^2=4,a>b^d$，$T(n)=O(n^{\log_b a})=O(n^{\log_2 8})=O(n^3)$。

Strassen 提出了一种变换方法，将子问题个数由 8 变为 7。其 7 个子问题具体如下：

$$
\begin{aligned}
M_1 &= A_{11}(B_{12}-B_{22})\\
M_2 &= (A_{11}+A_{12})B_{22}\\
M_3 &= (A_{21}+A_{22})B_{11}\\
M_4 &= A_{22}(B_{21}-B_{11})\\
M_5 &= (A_{11}+A_{22})(B_{11}+B_{22})\\
M_6 &= (A_{12}-A_{22})(B_{21}+B_{22})\\
M_7 &= (A_{11}-A_{21})(B_{11}+B_{12})\\
C_{11} &= M_5+M_4-M_2+M_6\\
C_{12} &= M_1+M_2\\
C_{21} &= M_3+M_4\\
C_{22} &= M_5+M_1-M_3-M_7
\end{aligned}
$$

这样变换增加矩阵加减法运算，时间复杂度递推方程如下：

$$
T(n)=\begin{cases}O(1) & n=1\\ 7T(n/2)+O(n^2) & n>1\end{cases}
$$

计算结果为 $T(n)=O(n^{\log_b a})=O(n^{\log_2 7})=O(n^{2.8075})$，算法的阶降低了。

第二步，治理。

(1) 求解各个子问题。若子问题规模较小且容易被解决，则直接求解；否则，再继续分解为更小的子问题，直到容易解决为止。

由于采用分治算法求解的问题被分解为若干个规模较小的相同子问题，各个子问题的解法与原问题的解法是相同的。因此，很自然想到采取递归技术来对各个子问题进行求解。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而规模不断缩小，最终使子问题缩小到很容易求解的规模，这导致递归过程的产生。分治与递归就像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。有时候，递归处理也可以采用循环来实现。

(2) 合并。它是将已求得的各个子问题的解合并为原问题的解。

合并这一步对分治算法的算法性能至关重要，算法的有效性在很大程度上依赖于合并步的实现，因情况的不同合并的代价也有所不同。

有些问题中，子问题的解就是原问题的解，如二分查找(折半查找)，以及后续要讲解的快速排序、棋盘覆盖等。

【例 3-4】 棋盘覆盖问题。

给定一个 $2^k\times 2^k$ 的棋盘(如图 3-2 所示 $k=2$ 的一种棋盘)，有一个特殊棋格，拥有一个特殊棋格的棋盘称为特殊棋盘。现要用 4 种 L 形骨牌(如图 3-3 所示)覆盖特殊棋盘上除特殊棋格外的全部棋格，不能重叠，找出覆盖方案。

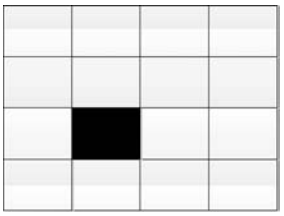

图 3-2　$k=2$ 的特殊棋盘

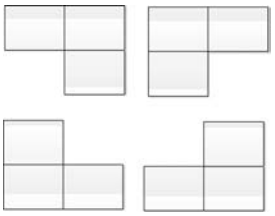

图 3-3　4 种 L 形骨牌

棋盘覆盖问题的分治算法：将棋盘行数平均分成两部分，列数也平均分成两部分，变成 $2^{k-1}\times 2^{k-1}$ 的 4 个小棋盘，如图 3-4 所示，将 4×4 特殊棋盘划分为 4 个 2×2 棋盘。其中，特殊棋格位于左下角的小棋盘。

现在考虑一下，4 个小棋盘的覆盖问题与原问题相同吗？原问题是在特殊棋盘上用 L 形骨牌覆盖，L 形骨牌互不重叠。4 个小棋盘中，只有左下角的棋盘是特殊棋盘，与原问题相同。但其他 3 个都不是特殊棋盘，与原问题不同。

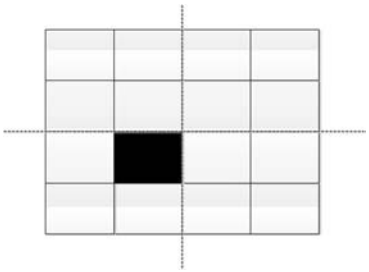

图 3-4　棋盘划分情况

在不改变原问题初衷的前提下，如何将它们变成与原问题相同的子问题呢？用一块 L 形骨牌覆盖 3 个小棋盘，每个棋盘覆盖一个棋格，如图 3-5 所示，这样，所有的子问题都与原问题相同，子问题规模变小且相互独立。

递归解决 4 个子问题，当子问题覆盖完成，整个棋盘就覆盖完成了，子问题的解不用归并就能得到原问题的解。如图 3-6 所示，用骨牌编号表示棋盘覆盖问题的解。

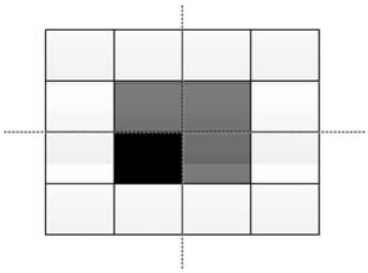

图 3-5　覆盖第一个骨牌情况

2	2	3	3
2	1	1	3
4	-1	1	5
4	4	5	5

图 3-6　棋盘覆盖的解

有些问题中，子问题的解并不是原问题的解，需要找出子问题的解与原问题的解之间的关系，由子问题的解归并得到原问题的解。如最小值问题，需要比较两个子问题的解，取较小的作为原问题的解，耗时为 $O(1)$；大整数乘法问题，需要通过公式 $\boldsymbol{A}\times\boldsymbol{B}=10^{n}A_1B_1+10^{n/2}(A_1B_2+A_2B_1)+A_2B_2$ 将子问题的解归并成原问题的解，耗时为 $O(n^2)$；矩阵乘法也是需要通过公式将子问题的解归并成原问题的解，耗时为 $O(n^3)$；后续要讲的幂乘问题、循环赛日程表问题、合并排序问题等都需要归并子问题的解，从而得到原问题的解，消耗的时间也不尽相同。

【例 3-5】　幂乘问题。

给定实数 a 和自然数 n，求 a^n。

问题很简单，传统方法是累乘，下面设计求幂乘的分治算法。

当 n 为偶数时，计算 $a^{n/2}$，然后用 $a^n=a^{n/2}\times a^{n/2}$ 治理求得原问题的解。

当 n 为奇数时，计算 $a^{(n-1)/2}$，然后用 $a^n=a^{(n-1)/2}\times a^{(n-1)/2}\times a$ 治理求得原问题的解。

该算法由子问题的解归并成原问题的解耗时 $O(1)$，递归子问题的规模每次递减一半，故递减 $\log_2 n$ 次便能将问题的规模降到 1，所以算法的时间复杂度为 $O(\log_2 n)$，空间复杂度也为 $O(\log_2 n)$。

3.2　二分查找


视频讲解

二分查找又称折半查找，它要求数据元素必须是按关键字大小有序排列的。该问题为给定已排好序的 n 个元素 $s_1,\cdots,s_n$，要在这 n 个元素中查找一特定元素 x 是否存在，若存在，则返回 x 在序列中的位置；否则，返回 -1。

3.2.1　问题分析——分与治的方法

假定用 a_list 表示 n 个元素的有序序列，该序列由小到大排序。

(1) 分解：将有序序列分成规模大致相等的两部分，即每部分的规模大致为 $n/2$。

(2) 治理：用序列中间位置的元素与特定元素 x 比较，如果 x 等于中间元素，那么算法终止；如果 x 小于中间元素，那么在序列的左半部递归查找；否则，在序列的右半部递归查找。递归停止的条件是序列规模为 0 或 1。

3.2.2　算法设计

1. 设计思想

通过问题分析，我们知道：①不同的子问题具有不同的规模；②不同的子问题在有序列 a_list 中的位置不同。采用辖定边界的方法统一表示不同问题的规模及在序列中的位置。

若令 left 表示子问题的下边界，right 表示子问题的上边界，则 a_list[left:right]表示不同的子问题。其分解步骤表示为 mid = (left+right)/2；递归的边界条件表示为 left>right。

治理表示为①判断 x 是否与 a_list[mid]相等，若相等，则返回 mid；②判断 x 是否大于 a_list[mid]，若大于，则在右边的子问题递归查找 x；否则，在左边的子问题递归查找 x。

2. 伪码描述

伪码描述如下：

```
算法:binary_search(a_list, left, right, x)
输入:a_list,x,left,right
输出:x在序列a_list中的位置或-1
if left > right then
    return -1
```

```
mid ← (left + right) / 2
if x == a_list[mid] then
return mid
if x < a_list[mid] then
    return binary_search(a_list, left, mid - 1, x)
else
    return binary_search(a_list, mid + 1, right, x)
```

3.2.3 实例构造

(1) 用二分查找算法在有序序列(6,12,15,18,22,25,28,35,46,58,60)中查找元素12。假定该有序序列存放在一维数组 a_list[0:10]中,则查找过程如下。

① 令 left=0,right=10,计算 mid=(0+10)/2=5,如图 3-7 所示。

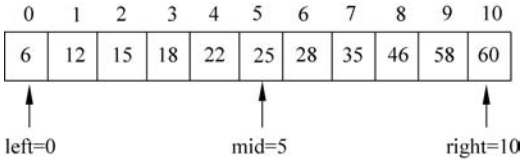

图 3-7 查找元素 12 的第一次分解示意

② 将 x 与 a_list[mid]进行比较。此时 x<a_list[mid],说明 x 可能位于序列的左半部 a_list[left:mid−1]中,应在左半部分递归查找。令 right=mid−1=4,计算 mid=(0+4)/2=2,如图 3-8 所示。

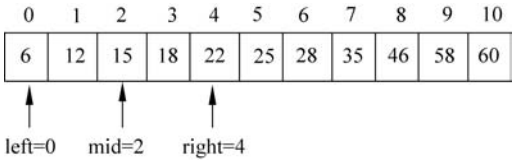

图 3-8 查找元素 12 的第二次分解示意

③ 将 x 与 a_list[mid]进行比较。此时 x<a_list[mid],说明 x 可能位于序列的左半部分 a_list[left:mid−1]中。令 right=mid−1=1,计算 mid=(0+1)/2=0,如图 3-9 所示。

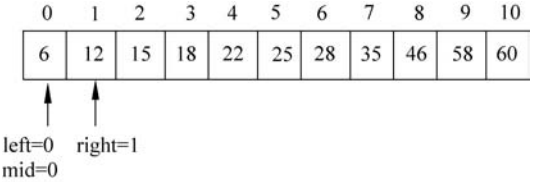

图 3-9 查找元素 12 的第三次分解示意

④ 将 x 与 a_list[mid]进行比较。此时 x>a_list[mid],说明 x 可能位于序列的右半部分 a_list[mid+1:right]中。令 left=mid+1=1,计算 mid=(1+1)/2=1,如图 3-10 所示。

此时 x=s[mid]=12,查找成功。

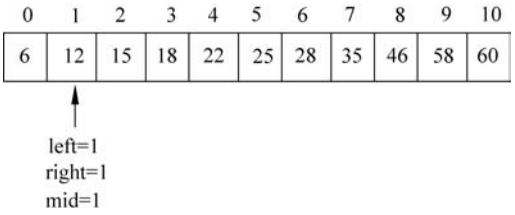

图 3-10　查找元素 12 的第四次分解示意

(2) 若用二分查找算法在有序序列(6,12,15,18,22,25,28,35,46,58,60)中查找元素 36,则查找过程如下。

① 令 left=0,right=10,计算 mid=5,即利用中间位置 mid 将序列一分为二,如图 3-11 所示。

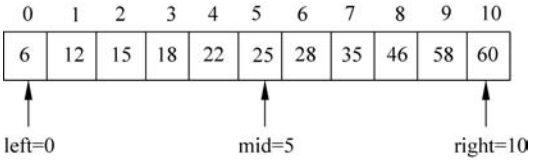

图 3-11　查找元素 36 的第一次分解示意

② 将 x 与 a_list[mid]进行比较。此时 $x>$a_list[mid],说明 x 可能位于序列的右半部 a_list[mid+1:right]中。令 left=mid+1=6,计算 mid=(6+10)/2=8,如图 3-12 所示。

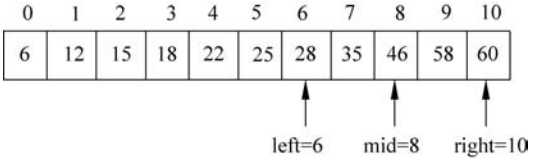

图 3-12　查找元素 36 的第二次分解示意

③ 将 x 与 a_list[mid]进行比较。此时 $x<$a_list[mid],说明 x 可能位于序列的左半部 a_list[left:mid−1]。令 right=mid−1=7,计算 mid=(6+7)/2=6,如图 3-13 所示。

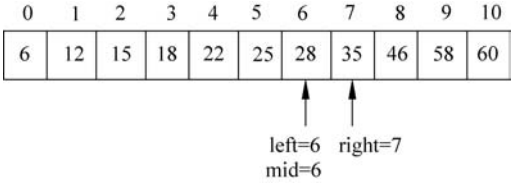

图 3-13　查找元素 36 的第三次分解示意

④ 将 x 与 a_list[mid]进行比较。此时 $x>$a_list[mid],说明 x 可能位于序列的右半部 a_list[mid+1:right]。令 left=mid+1=7,计算 mid=(7+7)/2=7,如图 3-14 所示。

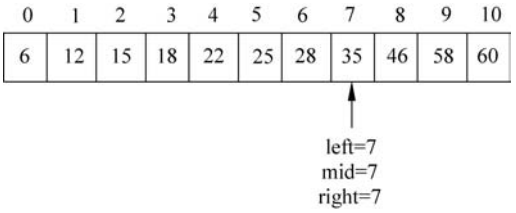

图 3-14　查找元素 36 的第四次分解示意

⑤ 将 x 与 a_list[mid]进行比较。此时 $x>$a_list[mid],说明 x 可能位于序列的右半部。令 left=mid+1=8,此时 left>right,返回−1,算法结束。

3.2.4　算法分析

1. 时间复杂度分析

算法 binary_search 在问题规模 n 等于 0 或 1 时,元素最多比较 1 次,耗时 $O(1)$。

问题的规模大于 1 时,分解操作耗时 $O(1)$;然后和中间位置的元素比较,若相等,则为最好情况,元素比较一次,耗时 $O(1)$。所以算法在最好情况下的时间复杂度为 $O(1)$。若不相等,算法则进入左半部分或右半部分查找。

在最坏情况下,元素比较操作和序列分解操作一直继续,直到元素规模为 1 或 0。该情况下算法耗时递推方程如下:

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

其中,n 为问题的规模,即序列中元素个数;$T(n)$表示规模为 n 的二分查找耗时,$T(n/2)$表示规模为 $n/2$ 的二分查找耗时。

利用主方法求解:$a=1,b=2,d=0,b^d=1,a=b^d$,$T(n)=O(n^d\log n)=O(\log n)$。

平均情况下,元素有可能比较 1 次、有可能 2 次、…、有可能 $\log n$ 次,在等概率条件下,时间复杂度为:$(1+2+\cdots+\log n)/\log n=(1+\log n)/2=O(\log n)$。

2. 空间复杂度分析

该算法借助的辅助空间为 left、right、mid 三个变量,由于算法是递归,所以需要借助等于递归深度的栈空间。由于递归的深度最小为 0,最坏为 $\log n$,故算法的空间复杂度最好为 $O(1)$,最坏为 $O(\log n)$,平均情况下为 $O(\log n)$。

3.2.5　Python 实战

1. 数据结构选择

选用 Python 中的顺序存储结构 list 存储 n 个元素,记为 lis[0:$n-1$]。

2. 编码实现

编写 binary_search()函数,接收输入 n 个元素序列 lis,下界 left,上界 right,待查找的元素 num,结果输出为 num 在 lis 中的位置或−1。

```
def binary_search(lis, left, right, num):
        if left > right:                    #递归结束条件
            return -1
        mid = (left + right) //2            #分解操作
        if num < lis[mid]:
            return binary_search(lis, left, mid-1, num)
        elif num > lis[mid]:
            return binary_search(lis, mid + 1, right, num)
        else:
            return mid
```

定义 Python 入口——main()函数，在 main()函数中，给定 lis 序列和待查找元素 num，先调用 sort()函数将 lis 由小到大排序，然后调用 binary_search()函数，然后将结果打印输出到显示器。

```
if __name__ == "__main__":
    lis = [11, 32, 51, 21, 42, 9, 5, 6, 7, 8]
    lis.sort()
    left = 0
    right = len(lis) - 1
    num = 8
    res = binary_search(lis, left,right,num)
    print("list[" + str(res) + "] = " + str(num))
```

输出结果为

list[3]=8

3.3　选第二大元素


视频讲解

给定 n 个元素，找出元素中的第二大元素。该问题如果用线性扫描的方法，首先找出最大值，比较 $n-1$ 次；然后从 $n-1$ 个元素中找出最大值即为 n 个元素的第二大元素，线性扫描比较 $n-2$ 次，所以找到 n 个元素的第二大元素需要比较 $2n-3$ 次。下面考虑设计一个选第二大元素的分治算法。

3.3.1　问题分析——分与治的方法

(1) 分解：将 n 个元素从中间一分为二，当 n 为奇数时，两个子问题的规模大致相等；当 n 为偶数时，两个子问题的规模完全相等，均为 $n/2$。

(2) 治理：递归两个子问题，分别求出两个子问题的最大值，同时将被淘汰的较小元素记录在较大元素的列表中。比较子问题的最大值，取较大元素为原问题的最大值。最后在最大值淘汰的元素列表中找出最大值即为原问题的第二大元素。

如找 10 个元素 6,12,3,7,2,18,90,87,54,23 的第二大元素。

首先从中间一分为二，递归找出左半部分的最大值，递归找出右半部分的最大值，同时将被淘汰的较小元素记录在淘汰它的元素列表中，如图 3-15 所示。

比较左半部分的最大值和右半部分的最大值，取较大者作为整个问题的最大值，

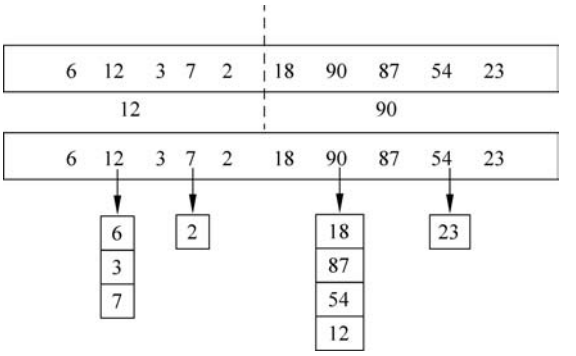

图 3-15 分解及淘汰元素列表示意

即 90。最后在最大值 90 的淘汰元素列表中找最大值,即 87,即为整个问题的第二大元素。

3.3.2 算法设计

1. 设计思想

通过问题分析,n 个元素存储在列表 a_list 中,令 left 表示子问题的下边界,right 表示子问题的上边界,则 a_list[left:right]表示不同的子问题。其分解步骤表示为 mid = (left+right)/2; 递归的边界条件表示为 left≥right,当问题规模为 1 时,单个元素本身就是序列的最大值。

治理划分为两个阶段: ①递归求解 a_list[left:mid]和 a_list[mid+1:right]两个子问题的最大值 left_max 和 right_max,同时将被淘汰的元素放入淘汰它的元素对应的列表中。比较 left_max 和 right_max,取较大的作为 n 个元素的最大值。②在 n 个元素的最大值淘汰的元素列表中,顺序查找最大值,即为第二大元素。

2. 伪码描述

第一个阶段找 n 个元素的最大值,伪码描述如下:

```
算法:find_max(a_list)
输入:n 个元素序列
输出:n 个元素的第二大元素
if n == 1 then
    return 元素本身
if n > 1 then
    mid ← (left + right)/2
    left_max←find_second(a_list[left:mid])
    right_max← find_second(a_list[mid + 1:right])
    if left_max > right_max then
        return left_max
        将 right_max 插入 left_max 的淘汰元素列表中
    else
        return right_max
        将 left_max 插入 right_max 的淘汰元素列表中
```

第二个阶段，在最大值淘汰的元素中顺序找最大值，伪码描述如下：

```
算法:find_second(s_max)
输入:最大值淘汰的元素序列 s_max
输出:序列 s_max 的最大值
first_max ← s_max[0]                    //序列中第一个元素
for i ← 1 to n - 1 do                   //从第二个元素开始比较,记录更大的
    if first_max < s_max[i] then
        first_max ←s_max[i]
return first_max
```

3.3.3　实例构造

找 10 个元素 6,12,3,7,2,18,90,87,54,23 的第二大元素。

第一阶段，找出 10 个元素的最大值。

(1) 将元素放入 a_list[0:9]中，令 left=0，right=9，计算 mid=(0+9)/2=4，两个子问题元素如图 3-16 所示。

(2) 递归左边的子问题 left=0，right=4，计算 mid=(0+4)/2=2，两个子问题元素如图 3-17 所示。

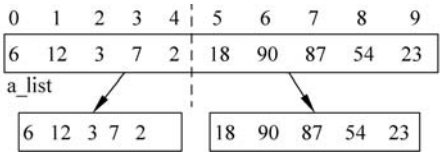

图 3-16　第一次分解示意

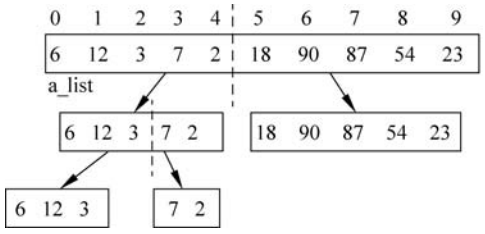

图 3-17　第二次分解示意

(3) 递归左边的子问题 left=0，right=2，计算 mid=(0+2)/2=1，两个子问题元素如图 3-18 所示。

(4) 递归左边的子问题 left=0，right=1，计算 mid=(0+1)/2=0，两个子问题元素如图 3-19 所示。

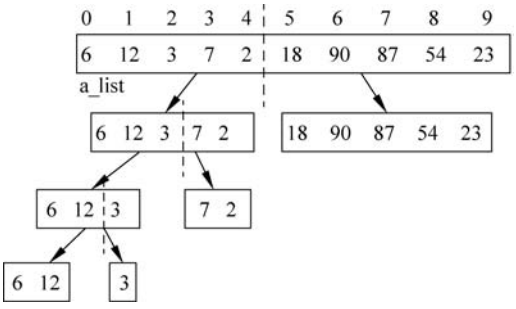

图 3-18　第三次分解示意

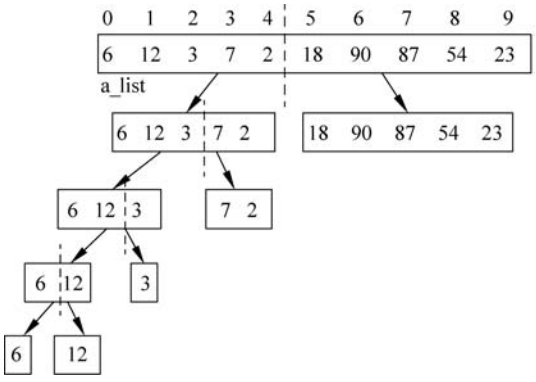

图 3-19　第四次分解示意

(5) 此时到达递归的边界条件,开始回归:左边子问题的解为 6,右边子问题的解为 12,12 淘汰 6,将 6 插入 12 的列表中,如图 3-20 所示。

(6) 继续回归,左边子问题的解为 12,右边子问题的解为 3,12 淘汰 3,将 3 插入 12 的列表中,如图 3-21 所示。

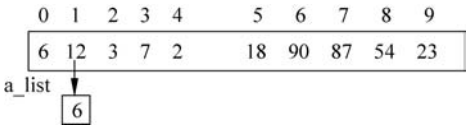

图 3-20 记录回归过程淘汰示意 1

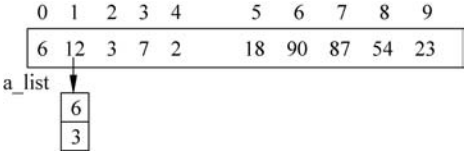

图 3-21 记录回归过程淘汰示意 2

(7) 继续回归,左边子问题的解为 12,右边子问题的解为 7(7 淘汰 2,2 插入 7 的列表中),12 淘汰 7,将 7 插入 12 的列表中,如图 3-22 所示,a_list[0:4]子问题递归过程结束。

(8) 同样地,a_list[5:9]子问题的递归过程中,首先 90 淘汰 18,然后 90 淘汰 87,54 淘汰 23,最后 90 淘汰 54,如图 3-23 所示。

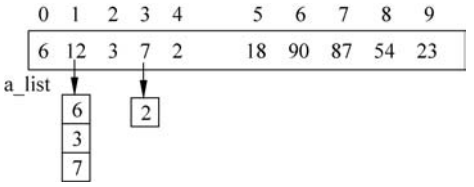

图 3-22 记录回归过程淘汰示意 3

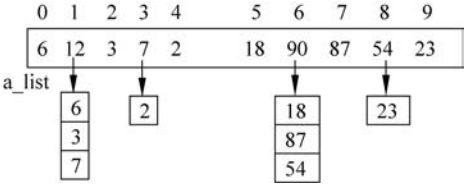

图 3-23 记录回归过程淘汰示意 4

(9) a_list[0:4]子问题的解为 12,a_list[5:9]子问题的解为 90,90 淘汰 12,获得冠军,如图 3-24 所示。

第二阶段,在 18,87,54,12 四个元素中找最大值,为 87,即 87 是要找的第二大元素。

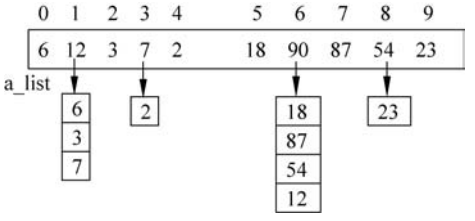

图 3-24 记录回归过程淘汰示意 5

3.3.4 算法分析

1. 时间复杂度分析

算法第一阶段找最大值,分解操作耗时 $O(1)$。治理操作:规模为 n 时消耗的时间为 $T(n)$,递归两个子问题时,问题的规模大致为 $n/2$,耗时为 $2T(n/2)$;比较子问题的解得到原问题的解耗时 $O(1)$;将被淘汰的元素插入到淘汰它的元素的列表中,耗时 $O(1)$;所以该阶段消耗的时间递推方程如下:

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

解此递推方程可得 $O(n)$。

算法第二阶段,在最大值淘汰的元素中找最大值,采用顺序查找的方法,消耗的时间是最大值淘汰的元素个数。那么,最大值淘汰了多少个元素呢?

仔细观察元素的比较过程,两个元素相比,较大的元素胜出,继续参加下一轮的比

较。若出现轮空，则直接进入下一轮的比较。该递归过程实际上是锦标赛淘汰赛产生冠军的过程。产生冠军的过程打了几轮比赛，则冠军选手就淘汰多少个对手。从递归的过程图来看，处于同一深度的则是同一轮比赛，比赛的轮次等于递归的深度，所以 n 个选手的锦标赛，被冠军淘汰的有$\lceil \log n \rceil$位选手。由此可知，第二阶段查找的元素个数为 $\log n$，算法时间复杂度为$\lceil \log n \rceil$。

第一阶段和第二阶段时间相加，得到第二大元素算法的时间复杂度为 $O(n+\lceil \log n \rceil)$。

2. 空间复杂度分析

第一阶段求最大值递归的深度为 $\log n$，所以需要借助的辅助栈空间为 $\log n$；另外需要记录被淘汰的元素，按照淘汰赛的规则，第一轮淘汰 $n/2$ 个元素，第二轮淘汰 $n/4$ 个元素，……，最后一轮淘汰 1 个元素，所以共需要的辅助空间为$(1+2+4+\cdots+n/2)=n-1$。故算法的空间复杂度为 $O(n+\log n)$。

3.3.5　Python 实战

1. 数据结构选择

选用 list 存储 n 个元素，在比较的过程中需要用列表存储被较大元素淘汰的元素，所以选用字典存储，字典中每个元素的 key 是给定的 n 个元素，value 是一个列表，记录由 key 淘汰的元素。

2. 编码实现

首先定义一个 find_max()函数，实现第一阶段的操作：接收 n 个元素的列表 a_list，要查找的问题的边界 left 和 right。输出 n 个元素的最大值，同时将被淘汰的元素插入到字典相应 key 的 value 中。其代码如下：

```
def find_max(a_list,left,right):
    global dic
    if left >= right:
        return a_list[left]
    mid = (left + right)//2
    left_max = find_max(a_list,left,mid)
    right_max = find_max(a_list,mid+1,right)
    if left_max > right_max:
        dic[left_max].append(right_max)
        return left_max
    else:
        dic[right_max].append(left_max)
        return right_max
```

其次，定义 find_second()函数完成第二阶段的操作：接收最大值淘汰的元素列表，顺序查找列表中的最大值，即为 n 个元素的第二大元素。其代码如下：

```
def find_second(n_max):
    n = len(n_max)
```

```
    second_max = n_max[0]
    for i in range(1,n):
        if n_max[i] > second_max:
            second_max = n_max[i]
    return second_max
```

最后定义 Python 入口——main()函数，在 main()函数中，提供了测试数据，调用 find_max()和 find_second()函数，并打印结果。

```
if __name__ == "__main__":
    a_list = [6,12,3,7,2,18,90,87,54,23]
    n = len(a_list)
    dic = {}
    for i in range(n):
        dic.update([(a_list[i],[])])#初始化字典列表
    first_max = find_max(a_list,0,n-1)
    second_max = find_second(dic[first_max])
    print( "最大值淘汰的元素为:",dic[first_max])
    print("第二大元素为:",second_max)
```

输出结果为

最大值淘汰的元素：[18，87，54，12]

第二大元素：87


视频讲解

3.4 循环赛日程表

循环赛日程表问题：设有 $n=2^k$ 个运动员要进行羽毛球循环赛，现要设计一个满足以下要求的比赛日程表：

(1) 每个选手必须与其他 $n-1$ 个选手各赛一次；

(2) 每个选手一天只能比赛一次；

(3) 循环赛一共需要进行 $n-1$ 天。

由于 $n=2^k$，显然 n 为偶数。

3.4.1 问题分析——分与治的方法

(1) 分解：根据分治算法的思想，将选手一分为二，n 个选手的比赛日程表可以通过对 $n/2=2^{k-1}$ 个选手设计的比赛日程表来实现，而 2^{k-1} 个选手的比赛日程表可通过对 $2^{k-1}/2=2^{k-2}$ 个选手设计的比赛日程表来实现，以此类推，2^2 个选手的比赛日程表可通过对两个选手设计的比赛日程表来实现。此时，问题的求解将变得异常简单。

(2) 治理：递归解决两个规模为 2^{k-1} 个选手的子问题，然后让两组选手对打，就可以排出整个循环赛日程表。递归停止的条件为问题的规模为 1 时，1 个选手不用安排比赛日程。

如 $n=2$，分两组，每组 1 个选手，边界条件，不用安排。然后两组对打，安排如图 3-25

所示。

如 $n=4$，分两组(1,2)和(3,4)，每组 2 个选手，递归安排两组，然后两组对打，安排如图 3-26 所示。

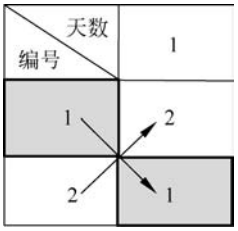

图 3-25　$n=2$ 的竞赛安排示意

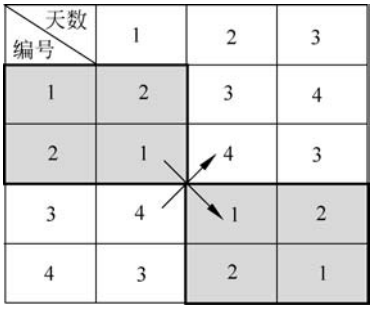

图 3-26　$n=4$ 的竞赛安排示意

3.4.2　算法设计

根据问题分析，算法需要填写循环赛日程表格，所以需要确定递归的子问题的规模、子问题位置。只要规模大于 1，就分解，治理；算法伪码描述如下：

```
算法:arrange(p,q,n,arr)
输入:子问题的位置(p,q)、子问题的规模 n、循环赛日程表 arr
if (n>1) then
    arrange(p,q,n/2,arr)
    arrange(p,q+n/2,n/2,arr)
    //两组对打
//填左下角
    for i←p+n/2 to p+n do
        for j←q to q+n/2 do
            arr[i][j]←arr[i-n/2][j+n/2]
  //填右下角
    for i ←p+n/2 to p+n do
        for j ←q+n/2 to q+n do
            arr[i][j]←arr[i-n/2][j-n/2]
    return arr
```

3.4.3　实例构造

安排 $n=2^3$ 个选手 $n-1$ 天的比赛日程。安排过程如下：

(1) 将 8 个选手(1,2,3,4,5,6,7,8)分为两组(1,2,3,4)、(5,6,7,8)，每组 4 个选手。

(2) 将 4 个选手分为两组，每组 2 个选手，分别为(1,2)、(3,4)、(5,6)、(7,8)。

(3) 将 2 个选手分为两组，每组 1 个选手，到了递归的边界条件，开始回归。回归时，组内比赛日程已经安排好，剩下只需两组对打就行了。第 1 天的比赛安排如图 3-27 所示。

(4) 继续回归，每组 2 个选手的已经安排好，将两组对打，便可以得到前 3 天的比赛日程表，如图 3-28 所示。

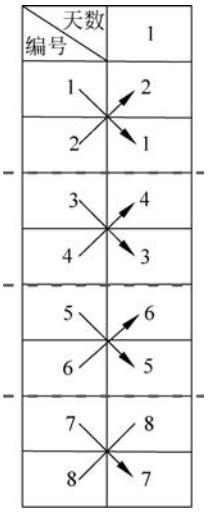

图 3-27　$n=8$ 的第 1 天竞赛安排示意

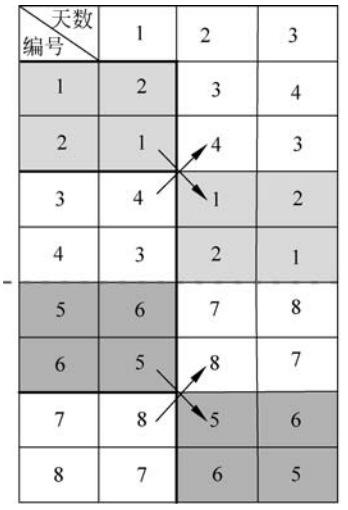

图 3-28　$n=8$ 的前 3 天竞赛安排示意

(5) 继续回归,每组 4 个选手的已经安排好,将两组对打,便可以得到 7 天的比赛日程表,如图 3-29 所示。

编号\天数	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

图 3-29　$n=8$ 时 7 天的竞赛安排示意

3.4.4　算法分析

1. 时间复杂度分析

规模 $n=1$ 时,不用安排,只需要判断一下规模就可以了,故耗时为 $O(1)$。

规模 $n>1$ 时,循环赛日程表安排耗时为 $T(n)$,规模为 $n/2$ 的循环赛日程表安排耗时为 $T(n/2)$,将子问题的解归并为原问题的解(两组对打)耗时为 $O(n^2)$。故时间复杂度递推方程如下:

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

用主方法解此递推方程：$a=2,b=2,d=2,b^d=4,a<b^d,T(n)=O(n^d)=O(n^2)$。

2. 空间复杂度分析

算法采用递归实现，递归过程借助的栈空间为 $\log n$，所以算法的空间复杂度为 $O(\log n)$。

3.4.5　Python 实战

1. 数据结构选择

采用二维列表 arr 存储循环赛日程表。

2. 编码实现

定义一个 arrange()函数，接收子问题的位置(p,q)、规模 n 和空的循环赛日程表 arr，输出已安排好的循环赛日程表 arr。其代码如下：

```
def arrange(p,q,n,arr):
    if(n>1):                              #规模大于1时,分解
        arrange(p,q,n//2,arr)             #递归解决子问题
        arrange(p,q+n//2,n//2,arr)        #递归解决子问题
      #两组对打
      #填左下角
        for i in range(p+n//2,p+n):
            for j in range(q,q+n//2):
                arr[i][j]=arr[i-n//2][j+n//2]
        #填右下角
        for i in range(p+n//2,p+n):
            for j in range(q+n//2,q+n):
                arr[i][j]=arr[i-n//2][j-n//2]
        return arr
```

定义 Python 入口——main()函数，在 main()函数中，首先确定解决的问题规模 n，并初始化循环赛日程表 arr，然后调用 arrange()函数完成循环赛日程安排，最后打印循环赛日程表。其代码如下：

```
if __name__ == "__main__":
    import numpy as np
    k = 4
    n = 2 ** k
    arr = np.zeros((n,n),dtype = int)
    for i in range(n):
        arr[0][i]=i+1
    arrange(0,0,n,arr)
    print (arr)
```

输出结果为

```
[[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
 [ 2  1  4  3  6  5  8  7 10  9 12 11 14 13 16 15]
 [ 3  4  1  2  7  8  5  6 11 12  9 10 15 16 13 14]
 [ 4  3  2  1  8  7  6  5 12 11 10  9 16 15 14 13]
 [ 5  6  7  8  1  2  3  4 13 14 15 16  9 10 11 12]
 [ 6  5  8  7  2  1  4  3 14 13 16 15 10  9 12 11]
 [ 7  8  5  6  3  4  1  2 15 16 13 14 11 12  9 10]
 [ 8  7  6  5  4  3  2  1 16 15 14 13 12 11 10  9]
 [ 9 10 11 12 13 14 15 16  1  2  3  4  5  6  7  8]
 [10  9 12 11 14 13 16 15  2  1  4  3  6  5  8  7]
 [11 12  9 10 15 16 13 14  3  4  1  2  7  8  5  6]
 [12 11 10  9 16 15 14 13  4  3  2  1  8  7  6  5]
 [13 14 15 16  9 10 11 12  5  6  7  8  1  2  3  4]
 [14 13 16 15 10  9 12 11  6  5  8  7  2  1  4  3]
 [15 16 13 14 11 12  9 10  7  8  5  6  3  4  1  2]
 [16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1]]
```


视频讲解

3.5 合并排序

给定 n 个元素，要求将 n 个元素排成有序序列。可以升序，也可以降序，不妨假设升序排列 n 个元素。

3.5.1 问题分析——分与治的方法

(1) 分解：将待排序的 n 个元素分成规模大致相同的 2 个子序列，即从中间一分为二，2 个子序列的元素个数要么相等，要么相差 1 个元素。

(2) 治理：递归解决 2 个子问题。2 个子序列有序了，然后将排好序的 2 个子序列合并成 1 个有序的序列，即得到原问题的解。

3.5.2 算法设计

1. 设计思想

通过问题分析，n 个元素存储在列表 arr 中，令 left 表示子问题的下边界，right 表示问题的上边界，则 arr[left:right]表示不同的子问题。其分解步骤表示为 mid = (left+right)/2，递归的边界条件表示为 left≥right，当问题规模为 1 时，单个元素本身就是有序序列。

治理时，首先递归子问题 arr[left:mid]和 arr[mid+1:right]，然后将排好序的 arr[left:mid]和 arr[mid+1:right]2 个子序列归并成 1 个子序列 arr[left:right]。

2. 伪码描述

将排好序的 arr[left:mid]和 arr[mid+1:right] 2 个子序列归并成 1 个子序列

arr[left:right]的伪码如下：

```
算法:merge(arr, left, mid, right)
输入:待排序元素 arr、子问题的边界 left、mid、right
输出:排好序的序列 arr[left:right]
L ← arr[left:mid]                          //提取左边子序列元素
R ← arr[mid + 1:right]                     //提取右边子序列元素
i ← 0
j ← 0
k ← left
while L 和 R 都未遍历结束 do
        if L[i] <= R[j]:
            arr[k] ← L[i]
            i += 1
        else:
            arr[k] ← R[j]
            j += 1
        k += 1
  //拷贝 L□ 的保留元素
    while L 未遍历结束:
        arr[k] ← L[i]
        i += 1
        k += 1
  //拷贝 R□ 的保留元素
    while R 未遍历结束:
        arr[k] ← R[j]
        j += 1
        k += 1
```

合并排序伪码如下：

```
算法:mergeSort(arr,left,right)
输入:待排序元素序列 arr、子问题的边界 left 和 right
输出:由小到大排好序的元素序列
if left < right then
    mid ← (left + right)/2                 //分解
    mergeSort(arr, left, mid)              //递归左边
    mergeSort(arr, mid + 1, right)         //递归右边
    merge(arr, left, mid, right)           //归并两个有序子序列
```

3.5.3　实例构造

设待排序序列 arr=[8,3,2,9,7,1,5,4]，采用 MergeSort 算法对序列 arr 进行排序。具体排序过程如图 3-30 所示。

其实，综合算法的设计思想和上述求解过程展示，很容易看出合并排序算法的求解过程实质上是经过迭代分解，待排序序列 arr 最终被分解成 8 个只含一个元素的序列，然后两两合并，最终合并成一个有序序列。

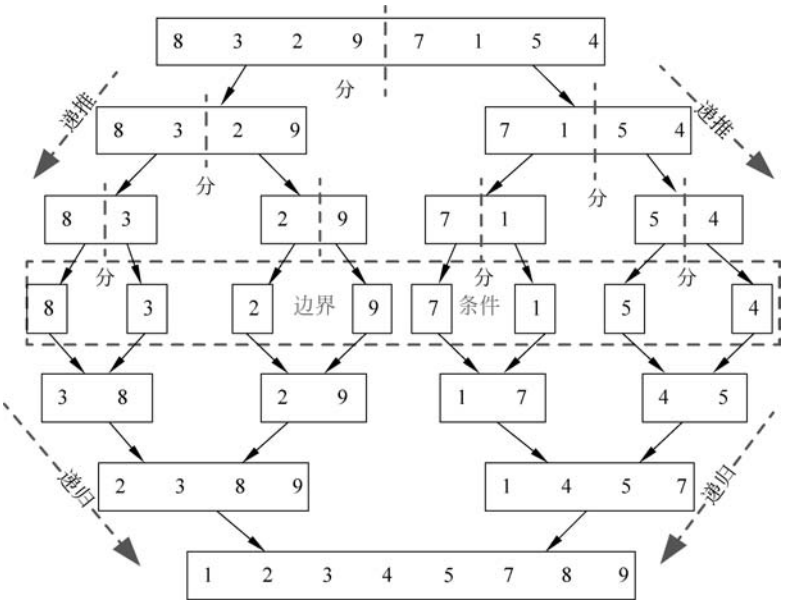

图 3-30 合并排序过程示意

3.5.4 算法分析

1. 时间复杂度分析

假设待排序序列中元素个数为 n。

显然,当 $n=1$ 时,合并排序一个元素需要常数时间,因而 $T(n)=O(1)$。

当 $n>1$ 时。

分解:这一步仅仅是计算出子序列的中间位置,需要常数时间 $O(1)$。

解决子问题:递归求解两个规模为 $n/2$ 的子问题,所需时间为 $2T(n/2)$。

合并:对于一个含有 n 个元素的序列,Merge 算法可在 $O(n)$时间内完成。

将以上阶段所需的时间进行相加,即得到合并排序算法对 n 个元素进行排序所需的运行时间 $T(n)$的递推方程如下:

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

采用主方法求解递推方程:$a=2,b=2,d=1,b^d=2,a=b^d$,$T(n)=O(n^d\log n)=O(n\log n)$。

2. 空间复杂度分析

算法采用递归实现,递归过程借助的栈空间为 $\log n$,将两个有序序列归并成一个有序序列借助的辅助空间为 $O(n)$,每次回归都需要 $O(n)$个辅助空间,所以算法的空间复杂度为 $O(\log n)+O(n\log n)=O(n\log n)$。

3.5.5 Python 实战

选用 list 存储待排序的 n 个元素,定义一个 merge()函数用于完成将两个有序子序

列 arr[left:mid]和 arr[mid+1:right]归并成一个有序的序列 arr[left:right]。其代码如下：

```
def merge(arr, left, mid, right):
    n1 = mid - left + 1                 #子问题 1 的元素个数
    n2 = right- mid                     #子问题 2 的元素个数
    #创建临时数组,L 用于存储子问题 1 的元素,R 用于存储子问题 2 的元素
    L = arr[left:mid + 1]
    R = arr[mid + 1:right + 1]
    #归并临时数组到 arr[l..r]
    i = 0                               #初始化第一个子数组的索引
    j = 0                               #初始化第二个子数组的索引
    k = left                            #初始归并子数组的索引
    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    #拷贝 L□ 的保留元素
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    #拷贝 R□ 的保留元素
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

定义一个 mergeSort()函数用于完成合并排序，函数中完成分解、治理和边界条件的判定。其代码如下：

```
def mergeSort(arr,left,right):
    if left < right:                    #边界条件判断
        mid = (left + right)//2         #分解
        mergeSort(arr, left, mid)       #递归子问题 1
        mergeSort(arr, mid + 1, right)  #递归子问题 2
        merge(arr, left, mid, right)    #将两个子问题的解归并成原问题的解
```

定义 Python 入口——main()函数，在 main()函数中，首先初始化一个实例，然后调用 mergeSort()函数完成合并排序，最后输出排序结果。其代码如下：

```
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7,20,10,100,34]
    n = len(arr)
    mergeSort(arr,0,n - 1)
    print (arr)
```

输出结果为

[5, 6, 7, 10, 11, 12, 13, 20, 35, 100]


视频讲解

3.6 快速排序

快速排序是 C. R. A. Hoare 于 1962 年提出的一种划分交换排序,其基本思想是通过一趟扫描将待排序的元素分割成独立的三个序列:第一个序列中所有元素均不大于基准元素、第二个序列是基准元素、第三个序列中所有元素均大于基准元素。由于第二个序列已经处于正确位置,因此需要再按此方法对第一个序列和第三个序列分别进行排序,整个排序过程可以递归进行,最终可使整个序列变成有序序列。

3.6.1 问题分析——分与治的方法

(1) 分解:快速排序的分解是基于基准元素的,所以首先要选定一个元素作为基准元素,然后以选定的基准元素为标杆,将其他元素分成两部分,一部分不大于基准元素,另一部分大于基准元素。

(2) 基准元素的选取:从待排序序列中选取的基准元素是决定算法性能的关键。基准元素的选取应该遵循平衡子问题的原则,即使得划分后的两个子序列的长度尽量相同。基准元素的选择方法有很多种,常用的有以下 5 种。

① 取第一个元素。即以待排序序列的首元素作为基准元素。

② 取最后一个元素。即以待排序序列的尾元素作为基准元素。

③ 取位于中间位置的元素。即以待排序序列的中间位置的元素作为基准元素。

④ “三者取中的规则”。即在待排序序列中,将该序列的第一个元素、最后一个元素和中间位置的元素进行比较,取三者之中值作为基准元素。

⑤ 随机取一个元素作为基准元素。

(3) 治理:递归不大于基准元素的子序列,然后再递归大于基准元素的子序列,这样不大于基准元素的子序列和大于基准元素的子序列都有序了。基准元素位于正确的位置,整个序列就都有序了,完成了排序的目的。边界条件为子序列且为空或只有 1 个元素。

3.6.2 算法设计

将待排序的 n 个元素放到列表 lis 中,用 left 和 right 记录子问题在 lis 中的位置及规模,基准元素记为 pivot。首先定义一个 partition()函数完成分解任务,然后采用递归方法完成子问题的排序。

在划分过程中,选取待排序元素的第一个元素作为基准元素,然后从左向右、从右向左两个方向轮流找出位置不正确的元素,将其交换位置。该过程一直持续到所有元素都比较结束。最后将基准元素放到正确的位置。

划分操作伪码描述如下:

```
算法:partition(lis,left,right):#lis——> 待排序元素 left ——> 起始索引 right ——> 结束索引
输入:待排序元素序列 lis,子问题的边界 left 和 right
输出:基准元素的位置
```

```
i←子问题的下边界 left
j←子问题的上边界 right + 1
pivot←lis[left]                          #用序列的第一个元素作为基准元素
while(True) do
    //推动左指针 i,开始从左向右扫描
    i ←i + 1
    while(lis[i]< pivot) do              //从左向右扫描,找到比基准元素大的停止,该元素位
                                         //置不正确
        i ←i + 1
    //推动右指针 j,开始从右向左扫描
    j ←j - 1
    while(lis[j]> pivot) do              //从尾向前扫描,找到比基准元素小的停止,该元素位
                                         //置不正确
    j ←j - 1
    //若所有元素都扫描结束,则结束 while(true)循环
    if(i > = j) then
        break
    lis[i]↔lis[j])                       //交换 lis[i]和 lis[j]
lis[j]↔lis[left]                         //将基准元素放入正确的位置
return j                                 //返回基准元素的位置
```

基于划分的结果，快速排序递归左边的子序列 lis[left:j－1]，递归右边的子序列 lis[j＋1:right]，伪码描述如下：

```
算法:quickSort(lis,left,right)
输入:待排序序列、子问题的边界 left 和 right
输出:有序序列 lis
if left < right:
    j ← partition1(lis,left,right)
    quickSort(arr, left, j - 1)
    quickSort(arr, j + 1, right)
```

3.6.3　实例构造

给定序列 23,15,10,50,93,12,2,68，采用快速排序方法将其升序排列。

(1) 初始序列，如图 3-31 所示，23 为基准元素，$i=0$，$j=8$。

(2) 从左向右扫描元素：先推动 i 指针，后比较，若 i 指向的元素小于或等于基准元素，则继续推动指针、再比较，直到 i 指向的元素比基准元素大为止，此时 i 指向的元素位置不正确，应该在右子序列中，所以应该调走，如图 3-32 所示。

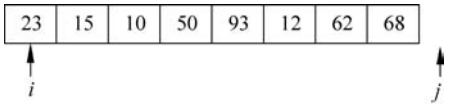

图 3-31　划分初始状态

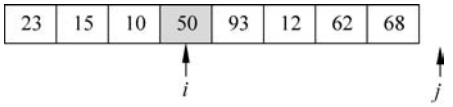

图 3-32　i 指针停止状态

(3) 从右向左扫描元素：先推动 j 指针，后比较，若 j 指向的元素大于基准元素，则继续推动指针、比较，直到 j 指向的元素比基准元素小或相等为止，此时 j 指向的元素位置不正确，应该在左子序列中，所以应该调走，如图 3-33 所示。

(4) 交换 i 指向的元素和 j 指向的元素,如图 3-34 所示。

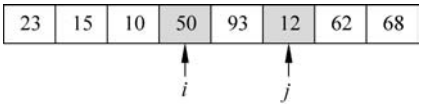

图 3-33 j 指针停止状态

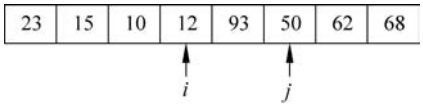

图 3-34 元素交换后的状态

(5) 由于 $i<j$,所以继续循环,推动 i 指针,直到 i 指向的元素大于基准元素;推动 j 指针,直到 j 指向的元素小于或等于基准元素,如图 3-35 所示。

(6) 此时 $i>j$,跳出 while(True)循环,将基准元素与 j 指向的元素交换位置,则 j 便是基准元素的位置,如图 3-36 所示。

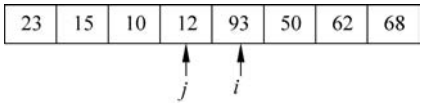

图 3-35 i 指针和 j 指针第二次扫描停止的状态

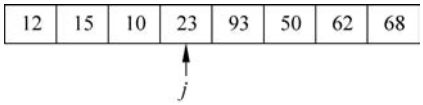

图 3-36 基准元素放到正确位置的状态

(7) 递归处理左子序列 12,15,10,递归处理右子序列 93,50,62,68。递归结束的时候,左右子序列均有序了,整个序列也都有序了,如图 3-37 所示。

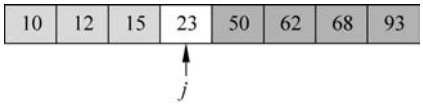

图 3-37 左右子问题递归结束的状态

3.6.4 算法分析

1. 时间复杂度分析

快速排序算法的时间主要耗费在划分操作上,与划分得到的子问题是否平衡密切相关。对于长度为 n 的待排序序列,一次划分算法 partition 需要对整个待排序序列扫描一遍,其所需的计算时间显然为 $O(n)$。

下面从三种情况来讨论一下快速排序算法 quickSort 的时间复杂度。

(1) 最坏时间复杂度。最坏情况是每次划分选取的基准元素都是在当前待排序序列中的最小(或最大)元素,划分的结果是基准元素左边的子序列为空(或右边的子序列为空),而划分所得的另一个非空的子序列中元素个数,仅仅比划分前的排序序列中元素个数少一个。

在这样的情况下,快速排序算法 quickSort 的运行时间 $T(n)$的递推方程如下:

$$T(n)=\begin{cases}O(1) & n\leqslant 1\\ T(n-1)+O(n) & n>1\end{cases}$$

采用迭代法求解递推方程:当 $n>1$ 时,

$$\begin{aligned}T(n)&=T(n-1)+O(n)\\&=T(n-2)+O(n-1)+O(n)\\&=T(n-3)+O(n-2)+O(n-1)+O(n)\end{aligned}$$

$$=\cdots=T(1)+O(2)+\cdots+O(n-1)+O(n)$$
$$=O(1+2+\cdots+n)$$
$$=O(n(n+1)/2)$$

因此，快速排序算法 QuickSort 的最坏时间复杂度为 $O(n^2)$。

如果按上面给出的 partition 划分算法，每次取当前排序序列的第 1 个元素为基准，那么当序列中的元素已按递增序（或递减序）排列时，每次划分所取的基准元素就是当前序列中值最小（或最大）元素，则完成快速排序所需的运行时间反而最多。

（2）最好时间复杂度。在最好情况下，每次划分所取的基准元素都是当前待排序序列的“中值”元素，划分的结果是基准元素的左、右两个子序列的长度大致相等，此时，算法的运行时间 $T(n)$的递推方程如下：

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

用主方法求解：$a=2,b=2,d=1,a=b^d$，$T(n)=O(n^d\log n)=O(n\log n)$，快速排序算法的最好时间复杂度为 $O(n\log n)$。

（3）平均时间复杂度。在平均情况下，设基准元素的位置为第 $k(1\leqslant k\leqslant n)$个，则有：

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}[T(n-k)+T(k-1)]+n=\frac{2}{n}\sum_{k=1}^{n}T(k)+n$$

采用差消法，最终求得 $T(n)$的数量级也为 $O(n\log n)$。

尽管快速排序的最坏时间为 $O(n^2)$，但就平均性能而言，它是基于元素比较的内部排序算法中速度最快者，快速排序亦因此而得名。

2. 空间复杂度分析

由于快速排序算法是递归执行，需要一个栈来存放每一层递归调用的必要信息，其最大容量应与递归调用的深度一致。在最好情况下，若每次划分较为均匀，则递归树的高度为 $O(\log n)$，故递归所需栈空间为 $O(\log n)$；在最坏情况下，递归树的高度为 $O(n)$，所需的栈空间为 $O(n)$；在平均情况下，所需栈空间为 $O(\log n)$。

3.6.5　Python 实战

选用 list 存储待排序的 n 个元素，定义一个 partition()函数完成以待排序序列的第一个元素为基准元素，将其他元素分为两部分，一部分不大于基准元素，另一部分大于基准元素。

```
def partition(lis,left,right):#lis → 待排序元素 left → 起始索引 right → 结束索引
    i = left
    j = right + 1
    pivot = lis[left]                    #用序列的第一个元素作为基准元素
    while(True):
        i += 1                           #向右推动指针
        while(lis[i]< pivot):  #从首往后扫描,找到比基准元素大的停止,该元素位置不正确
            i += 1                       #向右推动指针
        j -= 1                           #向左推动指针
        while(lis[j]> pivot):  #从尾向前扫描,找到比基准元素小的停止,该元素位置不正确
```

```
            j -= 1                          #向左推动指针
        if(i >= j):
            break
        lis[i],lis[j] = lis[j],lis[i] #交换 lis[i]和 lis[j],交换后 i 执行加 1 操作
    lis[j],lis[left] = lis[left],lis[j]
    return j
```

定义一个 quickSort()函数用于完成快速排序,函数中判断边界条件,调用 partition()函数完成分解,递归排序两个子序列。其代码如下:

```
def quickSort(lis,left,right):
    if left < right:                        #边界条件判断
        j = partition(lis,left,right) #分解
        quickSort(arr, left, j-1)       #递归左子序列
        quickSort(arr, j+1, right)      #递归左子序列
```

定义 Python 入口——main()函数,在 main()函数中,首先初始化一个实例,然后调用 quickSort()函数完成排序,最后输出排序结果。其代码如下:

```
if __name__ == "__main__":
    arr = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    n = len(arr)
    quickSort(arr,0,n-1)
    print ("排序后的数组:")
    print (arr))
```

输出结果为

排序后的数组:

[17, 20, 26, 31, 44, 54, 55, 77, 93]


视频讲解

3.7 线性时间选择——找第 k 小问题

给定线性序集中 n 个元素和一个整数 k,$1 \leqslant k \leqslant n$,要求找出这 n 个元素中第 k 小的元素。在某些特殊情况下,很容易设计出解选择问题的线性时间算法。如当要选择最大元素或最小元素时,显然可以在 $O(n)$时间完成。要找出 n 个元素的第 k 小,一般情况下有没有线性时间算法呢?下面来讨论线性时间选择的分治算法。

3.7.1 问题分析——分与治的方法

在本章第 3.6 节快速排序算法中,采用基准元素将待排序的序列分成两部分,一部分不大于基准元素,另一部分大于基准元素。受该方法的启发,如果基准元素所在的位置是第 k 个元素的位置,则基准元素就是要找的第 k 小元素;否则,如果基准元素所在的位置小于 k,则要找的第 k 小元素肯定在大于基准元素的子序列中;否则,如果基准元素所在的位置大于 k,则要找的第 k 小元素肯定在不大于基准元素的子序列中;因此,最坏情况只需要到其中一个子问题中查找。

(1) 分解:选取基准元素,将待排序序列中的元素与基准元素比较,不大于基准元素

的组成一个子序列，大于基准元素的组成另外一个子序列。

(2) 治理：检查基准元素的位置与 k 的大小关系，若两者相等，则基准元素就是要找的第 k 小元素；若基准元素的位置大于 k，则在不大于基准元素的子序列中找第 k 小；若基准元素的位置小于 k，则在大于基准元素的子序列中找第(k-基准元素位置)小。

3.7.2　算法设计

该算法的关键是基准元素的选取，若选得不好，最坏情况下则会导致每一次查找的子问题规模递减 1。这样，算法消耗的时间递推方程如下：

$$T(n)=\begin{cases}O(1) & n\leqslant 1\\ T(n-1)+O(n) & n>1\end{cases}$$

求解递推方程可得，$T(n)=O(n^2)$。

如何选取基准元素呢？我们采用以下方法：

(1) 将规模为 n 的序列分组，每 5 个元素一组，共 $n/5$ 组，不足一组的元素忽略不计。

(2) 取出每组的中位数 mid，共 $n/5$ 个。

(3) 取 $n/5$ 个中位数 mid 的中位数 mid_mid，mid_mid 就是选取的基准元素。

基准元素选取以后，将其与第一个位置的元素交换位置，然后以第一个位置的元素为基准进行划分。令 i 是序列中小于或等于基准元素的元素个数。检查 i 与 k 的大小关系，若两者相等，则基准元素就是要找的第 k 小元素；若 $k<i$，则在不大于基准元素的子序列中找第 k 小；若 $k>i$，则在大于基准元素的子序列中找第($k-i$)小。

算法伪码如下：

```
算法:select(A,left,right,k)
输入:序列 A,查找范围 A[left:right]、k
输出:第 k 小元素在 A 中的位置
//序列 A[left:right]中元素个数
n ← right - left + 1                    //A[left]到 A[right]的长度
if n == 1 then                          //若只有一个元素,则元素本身就是要找的元素,返回
                                        //其位置
    return left
if n < 5 then                           //元素个数太少,不足一组
    local_sort(A, left, right)          //将元素插入排序
    return left + k - 1                 //相对 left,第 k 个位置的元素就是第 k 小
groups← n/5                             //将元素分组,5 个元素一组
//取每组的中位数并将它们放到序列 A[left:right]的首部
for i in range(1, groups + 1) then
    A = local_sort(A, left + 5 * (i - 1), left + 5 * i - 1)
                                        //逐段对每五个元素进行插入排序
    A[left + i - 1],A[left + 5 * i - 3] = A[left + 5 * i - 3],A[left + i - 1]
                                        //每组中位数放到首部
j ← select(A, left, left + groups - 1, groups //2)
                                        //用 select 找出上述所有中位数的中位数
q ← partition(A,left,right,j)           //以 j 位置的元素为基准进行划分
i ← q - left + 1
if k == i then                          //边界条件
    return q
if k < i then
```

```
        return select(A,left,q - 1,k)          #在不大于基准元素的序列中查找第 k 小
    if k > i then
        return select(A,q + 1,right,k - i)     #在大于基准元素的序列中查找第 k-i
```

3.7.3 实例构造

(1) 在序列 89,10,21,5,2,8,33,27,63,55,66 中找第 3 小。

① 初始,left=0,right=10,将 11 个元素分为 2 组,分别为 89,10,21,5,2 和 8,33,27,63,55。

② 取每组的中位数 10、33。

③ 取每组中位数组成序列的中位数 10。

④ 以 10 为基准元素,将序列 89,10,21,5,2,8,33,27,63,55,66 划分结果为 8,5,2,10,89,21,27,33,55,63,66。

⑤ 小于或等于基准元素的元素个数为 4,3<4,故在子序列 8,5,2 中查找第 3 小。

⑥ 此时元素个数不足一组,将其插入排序,序列中的第三个元素 8 就是要找的第 3 小。

(2) 在序列 89,10,21,5,2,8,33,27,63,55,66 中找第 8 小。

① 初始,left=0,right=10,将 11 个元素分为 2 组,分别为 89,10,21,5,2 和 8,33,27,63,55。

② 取每组的中位数 10、33。

③ 取每组中位数组成序列的中位数 10。

④ 以 10 为基准元素,将序列 89,10,21,5,2,8,33,27,63,55,66 划分结果为:8,5,2,10,89,21,27,33,55,63,66。

⑤ 不大于基准元素的元素个数为 4,8>4,故在子序列 89,21,27,33,55,63,66 中查找第 8-4=4 小。

⑥ 此时,left=5,right=11,将 7 个元素分为 1 组为 89,21,27,33,55。

⑦ 取组的中位数 33。

⑧ 取每组中位数组成序列的中位数 33。

⑨ 以 33 为基准元素,将序列 89,21,27,33,55,63,66 划分结果为 21,27,33,55,89,63,66。

⑩ 小于或等于基准元素的元素个数为 3,4>3,故在子序列 55,89,63,66 中查找第 4-3=1 小。

⑪ 此时元素个数不足一组,将其插入排序,序列中的第一个元素 55 就是要找的整个序列的第 8 小。

3.7.4 算法分析

1. 时间复杂度分析

该算法分组耗时 $O(1)$,每组有 5 个元素,每组排序并取组的中位数耗时 $O(1)$,共有 $n/5$ 组,故取每组中位数消耗的时间为 $O(n)$。

取 $n/5$ 个中位数的中位数,采用从 $n/5$ 个元素中找第 $(n/5)/2$ 小的方法。假设从 n 个元素的序列选择第 k 小耗时为 $T(n)$,则取 $n/5$ 个中位数的中位数耗时为 $T(n/5)$。

划分耗时为 $O(n)$，根据划分结果，判断不大于基准元素的元素个数和 k 之间的大小关系，最坏情况进入其中一个子序列中继续查找，假设子序列的规模为 n/b，则耗时为 $T(n/b)$。

由此可得线性时间选择算法的时间复杂度递推方程如下：

$$T(n)=\begin{cases}O(1) & n<5\\ T(n/5)+T(n/b)+O(n) & n\geqslant 5\end{cases}$$

在该递推方程中，b 是未知的，我们需要估算 n/b。

如图 3-38 所示，图中共有 $n/5$ 组，每组元素从上到下排列，假设由小到大已排好序。然后将 $n/5$ 组从左向右排列，且按照每组的中位数由小到大排列。由此可知，左上角圈起来的元素都比 mid_mid 小，右下角圈起来的元素都比 mid_mid 大。

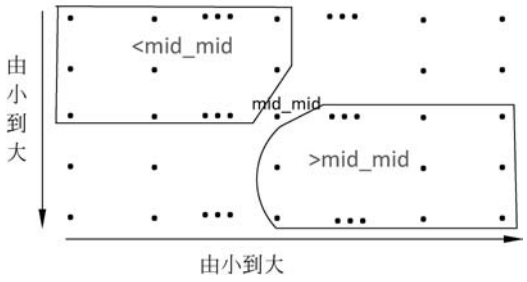

图 3-38　估算子问题规模示意

若算法到大于基准元素的子序列查找，则至少舍弃左上角圈起来的元素。反之，算法到不大于基准元素的子序列查找，则至少舍弃右下角圈起来的元素。那算法至少舍弃了多少个元素呢？

简单计算一下，左上角圈起来的元素（右下角圈起来的元素）至少位于 $(n/5-1)/2$ 组中，每组 3 个元素，则舍弃的元素至少 $3[(n/5-1)/2]+2=3n/10+0.5\geqslant 3n/10$。也就是说，要查找的子序列的规模至多是 $7n/10$。

由此算法的时间复杂度递推方程如下：

$$T(n)\leqslant\begin{cases}O(1) & n<5\\ T(n/5)+T(7n/10)+O(n) & n\geqslant 5\end{cases}$$

用树求解如图 3-39 所示。

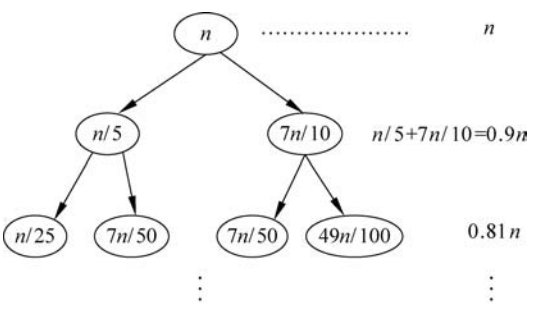

图 3-39　树

我们可以观察到,递归树中第一层耗时 n,第二层耗时为第一层耗时的 0.9 倍,第三层耗时是第二层的 0.9 倍,以此类推,算法的时间复杂度递推方程如下:

$$T(n)=n+0.9n+0.9^2n+\cdots=n/(1-0.9)=10n=O(n)$$

2. 空间复杂度分析

该算法组内的排序采用插入排序的方法就地进行,划分操作也是就地进行,存储 $n/5$ 个组的中位数采用原序列元素的存储空间,因此,该算法的时间复杂度为 $O(1)$。

3.7.5 Python 实战

首先定义一个 local_sort()函数,采用插入排序的方法完成组内元素的排序工作。该算法接收一组元素 A[left:right],对 A 的 left 到 right 部分进行插入排序,返回排序结果。其代码如下:

```
def local_sort(A,left,right):            #对 A 的 left 到 right 部分进行插入排序
    for j in range(left + 1,right + 1):
        x = A[j]
        for i in range(j,left - 1, - 1):
            if A[i - 1]> x:
                A[i] = A[i - 1]
            else:
                break
        A[i] = x
    return A
```

定义一个 partition()函数,完成以 A 中第 p 个位置的元素为基准,将 A 划分为不大于基准元素的子序列和大于基准元素的子序列。该算法接收待划分的元素序列 A[left:right],对 A 的 left 到 right 部分以 $A[p]$为基准进行划分,返回划分后基准元素的位置。其代码如下:

```
def partition(A,left,right,p):   #A[left:right]待划分的元素序列,p 为基准元素在 A 中的位置
    i = left
    j = right + 1
    A[left],A[p] = A[p],A[left]
    pivot = A[left]              #记录基准元素
    while(True):
        i += 1
        while(A[i]< pivot):      #从前往后扫描,找到比基准元素大的停止
            i += 1
        j -= 1
        while(A[j]> pivot):      #从尾向前扫描,找到小于或等于基准元素的停止
            j -= 1
        if(i >= j):
            break
        A[i],A[j] = A[j],A[i]  #交换 A[i]和 A[j],交换后 i 执行加 1 操作
    A[j],A[left] = A[left],A[j]
    return j
```

定义一个 select()函数,完成从 A[left:right]中线性时间查找第 k 小元素的功能。

该算法接收待查找元素序列 A[left:right]和 k,返回序列 A 中的第 k 小元素的位置。

```
def select(A,left,right,k):
    n = right - left + 1                #A[p]到 A[r]的长度
    if n == 1:
        return left
    if n < 5:
        local_sort(A, left, right)
        return left + k - 1
    groups = n//5
    for i in range(1, groups + 1):
        A = local_sort(A, left + 5 * (i - 1), left + 5 * i - 1)
                                        #逐段对每五个元素进行插入排序
        A[left + i - 1],A[left + 5 * i - 3] = A[left + 5 * i - 3],A[left + i - 1]
                                        #记录每组中位数
    j = select(A, left, left + groups - 1, groups // 2)   #用 select 找出上述所有中位数
                                                          #的中位数
    print("j = " + str(A[j]))
    q = partition(A,left,right,j)
    print("q = " + str(q))
    print(A)
    i = q - left + 1
    if k == i:                                  #边界条件
        return q
    if k < i:
        return select(A,left,q - 1,k)           #在小于基准元素的序列中查找第 k 小
    if k > i:
        return select(A,q + 1,right,k - i)      #在大于基准元素的序列中查找第 k－i 小
```

定义 Python 入口——main()函数,在 main()函数中,提供序列 A,调用 select()函数找到第 k 小在 A 中的位置,最后打印第 k 小元素。其代码如下:

```
if __name__ == '__main__':
    A = [89,10,21,5,2,8,33,27,63,55,66]
    k = select(A,0,10,8)
    print("第" + str(k + 1) + "小元素为:" + str(A[k]))
```

输出结果为

第 8 小元素为: 55