第5章　分治方法

从本章开始介绍常用的算法设计策略。首先介绍分治方法的基本思想和控制流程，然后给出几个实例，包括折半搜索、归并排序、快速排序、选择算法等，最后复习集合的基本知识，构建集合的顺序表示。

- 分治方法的基本思想和控制流程。
- 搜索算法（折半搜索）。
- 排序算法（归并排序和快速排序）。
- 选择算法（从n个元素中选择第k小的元素）。
- 集合的顺序表示*。

5.1 分治方法的基本思想和控制流程

5.1.1 基本思想

当人们要解决一个输入规模很大的问题时，往往会想到将该问题分解。比如将输入分成 k 个不同的子集。如果能得到 k 个不同的可独立求解的子问题，而且在求出这些子问题的解之后，还可以找到适当的方法把它们的解合并成整个问题的解，那么复杂的难以解决的问题就可以得到解决。这种将整个问题分解成若干个小问题来处理的方法称为分治方法，用分治方法设计的算法称为分治算法。

一般来说，被分解出来的子问题应与原问题具有相同的类型。如果得到的子问题相对来说还比较大，则可再用分治方法，直到产生出不用再分解就可求解的子问题为止。使用较多的是将整个问题二分。

5.1.2 控制流程

用DivideConquer(A, p, q)表示处理输入为A[p to q]情况的问题。

```
function DivideConquer(A, p, q)
{   if(Small(A, p, q)) return Solution(A, p, q);
    m = Divide(A, p, q); // 分成2个子问题, p<=m<q
    x = DivideConquer(A, p, m); // 递归求解子问题1
    y = DivideConquer(A, m + 1, q); // 递归求解子问题2
    return Combine(x, y); // 合并子问题的解
}
```

Small(A, p, q)用于判断规模为q – p + 1的子问题是否小到可以很容易得到答案。若是，则调用能直接求解子问题的函数Solution(A, p, q)。Divide(A, p, q)是分割函数，Combine(x, y)是解的合并函数。

5.1.3 复杂性分析

假定分成的两个子问题的规模大致相等，则DivideConquer总耗时的递归关系可以表示为

$$T(n)=\begin{cases} g(n) & \text{当}n\text{比较小时} \\ 2T(n/2)+f(n) & \text{当}n\text{比较大时} \end{cases}$$

其中，$g(n)$是直接求解Solution的用时，$f(n)$是Combine的用时。

【注】在分析分治算法的效率时，通常得到的是递归不等式

$$T(n)\le\begin{cases} g(n) & \text{当}n\text{比较小时} \\ 2T(n/2)+f(n) & \text{当}n\text{比较大时} \end{cases}$$

因为通常关心的是在最坏情况下计算时间的上界。所以，用等号（=）还是用不等号（≤）没有本质区别。

5.2 搜索算法

本节使用折半搜索方法解决下列3个搜索问题。

① 在升序数组X中搜索元素v，返回等于v的某个元素的位置，-1表示未找到。

② 在升序数组X中搜索元素v，返回不小于v的第一个元素的位置。

③ 在升序数组X中搜索元素v，返回大于v的第一个元素的位置。

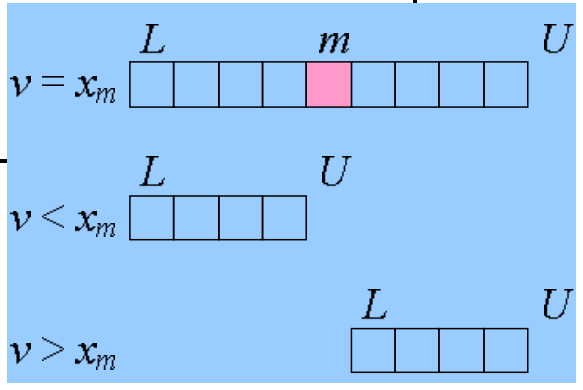
其中①可以简称为搜索元素位置，②和③可以合称为搜索元素范围。

5.2.1 折半搜索

1. 搜索元素位置

在升序数组X中搜索元素v，返回等于v的某个元素的位置，-1表示未找到。

```
template<class T, class T2>
int Search(T X[], int n, const T2 &v)
{   // 在升序数组X中搜索v所在的位置
    int low = 0, up = n;
    while(low < up)
    {   int m = (low + up) / 2;
        if(v == X[m]) return m; // 返回v所在位置
        else if(v < X[m]) up = m; // 左侧继续搜索
        else low = m + 1; // 右侧继续搜索
    }
    return -1; // -1表示未找到v
} // 耗时O(log(n))
```



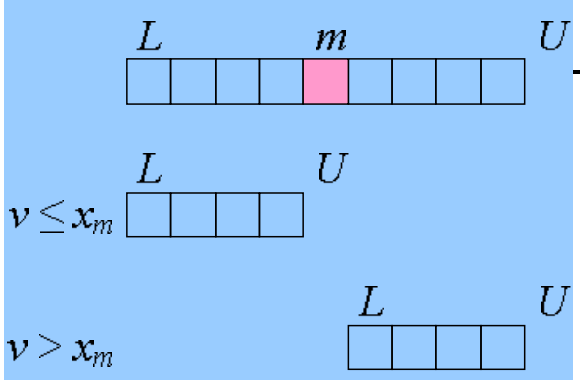
2. 搜索元素范围

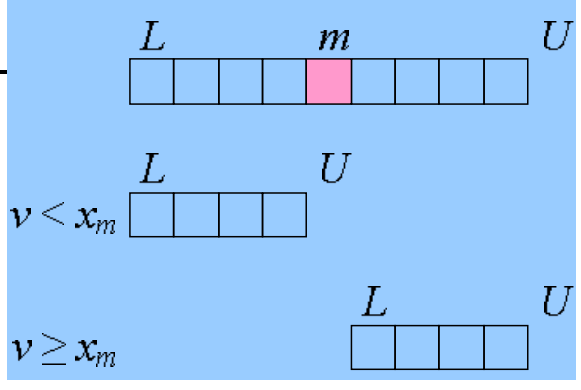
① 在升序数组X中搜索元素v，返回不小于v的第一个元素的位置。

② 在升序数组X中搜索元素v，返回大于v的第一个元素的位置。

```
template<class T, class T2>
int lower_bound(T X[], int n, const T2 &v)
{   // 搜索第一个不小于v的元素
    int low = 0, up = n;
    while(low < up)
    {   int m = (low + up) / 2;
        if(v <= X[m]) up = m; // 左侧
        else low = m + 1; // 右侧
    }
    return low;
}
```

```
template<class T, class T2>
int upper_bound(T X[], int n, const T2 &v)
{   // 搜索第一个大于v的元素
    int low = 0, up = n;
    while(low < up)
    {   int m = (low + up) / 2;
        if(v < X[m]) up = m; // 左侧
        else low = m + 1; // 右侧
    }
    return low;
}
```





3. 复杂性分析

while的每次循环（最后一次除外）都以减半的比例缩小搜索范围，所以，该循环在最坏情况下需要执行 $\Theta(\log n)$ 次。由于每次循环需耗时 $\Theta(1)$，因此在最坏情况下，总的时间复杂性为 $\Theta(\log n)$ 。

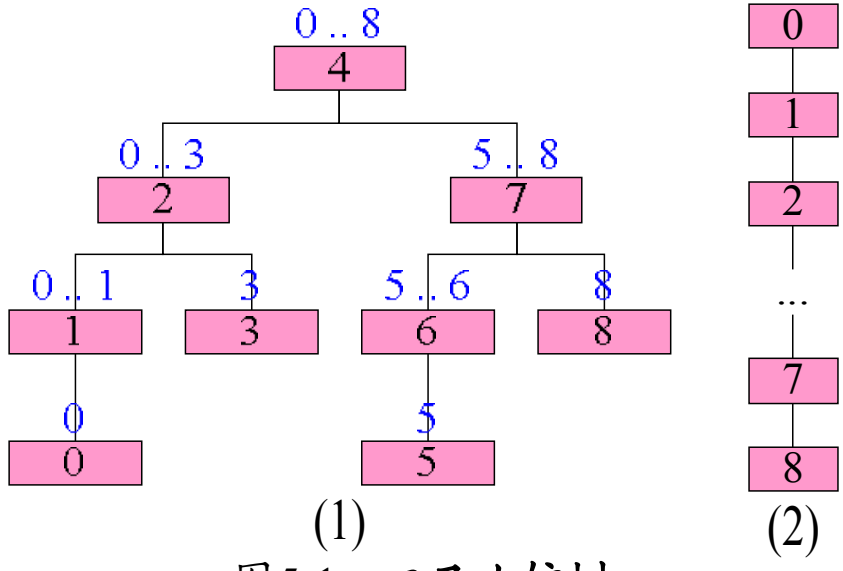
4. 测试程序*

```
#include "algorithm.h"
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    sort(X, X + n);
    cout << to_string(X, n) << endl; // 2 3 4 6 6 7 8 9 9
    cout << Search(X, n, 6) << endl; // 4
    cout << lower_bound(X, n, 6) << endl; // 3
    cout << upper_bound(X, n, 6) << endl; // 5
}
```

5.2.2 搜索算法的时间下界

1. 二叉比较树

可以用一个二叉比较树来分析折半搜索算法的时间复杂性。例如，对于9个元素的有序数组，折半搜索的二叉比较树如图5-1(1)所示。



(1) (2)

图5-1 二叉比较树

由图5-1(1)可见，一次成功的搜索至多做4次比较，而一次不成功的搜索做3次或4次比较。即当 $2^3 \le 9 < 2^4$ 时，元素比较的次数最多为4。一般地，当 $2^{k-1} \le n < 2^k$ 时，一次成功的搜索至多做 k 次比较，而一次不成功的搜索做 $k-1$ 或 k 次比较，元素比较的次数最多为 k 。

实际上，任何以比较为基础的搜索算法的执行过程都可以用一棵二叉比较树来描述。例如，对于9个元素的有序数组，顺序搜索的二叉比较树如图5-1(2)所示。

2. 二叉树的一个性质

【引理】深度为 k 的二叉树最多有 2^k-1 个顶点。

3. 搜索算法的时间下界

【定理1】设数组 $A[1..n]$ 的元素满足 $A[1]<A[2]<\ldots<A[n]$。则以比较为基础判断是否有 $x\in A[1..n]$ 的任何算法在最坏情况下所需的最少比较次数 $k\geq\lceil\log(n+1)\rceil$。

【证明】通过考察模拟求解搜索问题的各种可能算法的比较树可知，在最坏情况下，k 是所有顶点的最深层次，即二叉树的深度。根据二叉树的性质，深度为 k 的二叉树最多有 2^k-1 个顶点，因此 $n\leq 2^k-1$，从而 $k\geq\lceil\log(n+1)\rceil$。【证毕】

由定理可见，任何一种以比较为基础的搜索算法，在最坏情况下所用的时间都不可能低于 $\Theta(\log n)$，因此，也就不可能存在最坏情况下所需时间比折半搜索数量级还低的算法。

4. 问题的复杂性

如果一个问题 P 的任何算法在最坏情况下所需的最少时间 $t(n)=\Omega(f(n))$，则称问题 P 的时间下界为 $f(n)$ 或问题 P 的时间复杂性为 $f(n)$。例如，以比较为基础的搜索问题的时间复杂性为 $\log n$。问题的空间下界可以类似定义。

下界理论就是专门研究问题的复杂性下界的，与 NP 完全问题的研究有着紧密联系，是复杂性理论的基础。

5.3 排序算法

问题：已知n个元素的数组X，将X中元素按不降顺序排列。

5.3.1 归并排序算法

该算法采用分治方法，将要排序的数组分成两部分，先对每部分进行排序，然后将两部分的元素按不降顺序放在一个新数组中。这一过程可能需要多次分解和合并，因此是一个递归过程。

1. 合并两个有序组

```
template<class T> // 将有序组X[m]和Y[n]合并到W
void Merge(T X[], int m, T Y[], int n, T W[])
{   int i = 0, j = 0, k = 0; // X组游标i, Y组游标j, 结果游标k
    while(i < m && j < n) // 当两组都未取尽时
        if(X[i] <= Y[j]) W[k] = X[i], ++k, ++i;
        else W[k] = Y[j], ++k, ++j;
    while(i < m) W[k] = X[i], ++k, ++i; // 将X剩余部分复制到W尾部
    while(j < n) W[k] = Y[j], ++k, ++j; // 将Y剩余部分复制到W尾部
} // 耗时O(|X|+|Y|)
```

2. 合并过程举例说明

如图5-2所示。容易看出，Merge的时间复杂度为 $\Theta(|X|+|Y|)$。

1	3	5	^	2	4	6	7	8	^	W
i				j						[]
	i			j						1
	i				j					1, 2
		i			j					1, 2, 3
		i				j				1, 2, 3, 4
			i			j				1, 2, 3, 4, 5
										1, 2, 3, 4, 5, 6, 7, 8

1	3	5	7	8		2	4	6	^	W
i						j				[]
	i					j				1
	i						j			1, 2
		i					j			1, 2, 3
		i						j		1, 2, 3, 4
			i					j		1, 2, 3, 4, 5
			i						j	1, 2, 3, 4, 5, 6
										1, 2, 3, 4, 5, 6, 7, 8

图5-2　合并过程举例

3. 归并排序主程序

在合并程序的基础上，可以给出如下归并排序算法。

```
template<class T> // 借用临时数组W对数组X[n]排序
void MergeSort(T X[], int n, T W[])
{   if(n <= 1) return;
    int m = n / 2; // 求分割点
    T *Y = X + m; // Y[n-m]组
    MergeSort(X, m, W); // 将X[m]组排序
    MergeSort(Y, n - m, W); // 将Y[n-m]组排序
    Merge(X, m, Y, n - m, W); // 合并两个有序组
    for(int i = 0; i < n; ++i) X[i] = W[i]; // 已排序部分复制到X
} // 耗时O(n*log(n))
```

4. 复杂性分析

用 $T(n)$ 表示归并排序所用的时间，则

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

其中，a 是一个常数。根据简化Master定理可以得到 $T(n)=\Theta(n\log n)$ 。

【注】仔细观察该算法的执行过程可以发现，该算法的最好、最坏和平均时间复杂性都是 $\Theta(n\log n)$ 。

5. 测试*

① 调用方法。可使用如下形式调用。

```
template<class T> // 对数组X[n]排序
void MergeSort(T X[], int n)
{   T W[n]; // 借用临时数组W
    MergeSort(X, n, W);
}
```

② 测试程序。使用数组{6, 7, 9, 8, 4, 3, 2, 9, 6}。

```
#include "algorithm.h"
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << to_string(X, n) << endl;
    MergeSort(X, n);
    cout << to_string(X, n) << endl;
}
```

```
6 7 9 8 4 3 2 9 6
2 3 4 6 6 7 8 9 9
```

5.3.2 快速排序算法

另一个使用分治方法的排序算法是快速排序，由计算机科学家C. A. R. Hoare提出。

1. 基本策略

将数组区间[low, up)分解成子区间[low, p)和[p, up)，使得[low, p)中的元素都小于[p, up)中的元素，然后分别对这两个数组中的元素排序。一种典型的分解方法是从数组中选定一个元素作为划分元素，然后将数组中的所有元素与划分元素进行比较，小于划分元素的放在子区间[low, p)里，大于或等于划分元素的放在子区间[p, up)里。这个过程叫做划分。在划分时，使用尾部元素作为划分元素可以避免移动划分元素。

2. 划分程序

需要多次使用，保存在文件Partition.h中。

```
// Partition.h
#pragma once
#include <algorithm>
using namespace std;
```

```
template<class T>
int Partition(T X[], int low, int up) // 划分程序
{   // 待划分区间为[low, up), 左侧存放小元素
    int key = up - 1; // 使用尾部元素作为划分元素以避免移动划分元素
    for(int i = low; i < up; ++i) // 将小元素交换到左侧
        if(X[i] < X[key])
        {   swap(X[i], X[low]);
            ++low;
        }
    swap(X[key], X[low]); // 划分元素调整到右侧开始位置
    return low; // 划分位置
} // 耗时O(n)
```

3. 划分过程举例说明

设原数组为{6, 7, 9, 8, 4, 3, 2, 9, 6}，划分元素为key = 6，low = 0，up = 9，则划分过程如图5-3所示。容易看出，每一次调用Partition(low, up)涉及up - low个元素，进行up - low次元素比较。

L	i		0	1	2	3	4	5	6	7	8
0	4		6	7	9	8	4	3	2	9	6
1	5		4	7	9	8	6	3	2	9	6
2	6		4	3	9	8	6	7	2	9	6
3	8		4	3	2	8	6	7	9	9	6
			4	3	2	6	6	7	9	9	8

图5-3　划分过程举例

4. 算法主程序

在划分程序的基础上，可以给出如下快速排序算法。

```
#include "Partition.h"
#include "algorithm.h"
```

```
template<class T> // 对范围[low, up)排序
void QuickSort(T X[], int low, int up)
{   if(low >= up) return; // 没有元素
    int m = Partition(X, low, up); // 划分位置
    QuickSort(X, low, m); // 左侧排序
    QuickSort(X, m + 1, up); // 右侧排序
} // 平均耗时O(n*log(n)), 最坏耗时O(n*n)
```

5. 复杂性分析

要分析QuickSort，只需计算出它的元素比较次数即可。因为其它运算的频率计数和元素比较次数有相同的数量级。

① 最坏时间复杂性。每一次调用Partition(low, up)涉及up - low个元素，至多进行up - low次元素比较。将过程QuickSort按照划分来分层，第0层调用Partition一次，涉及 n 个元素；第1层调用Partition两次，涉及 $n-1$ 个元素，因为在第0层选定的划分元素不在其中；第 k 层调用Partition最多涉及 $n-k$ 个元素。因为 $0 \le k < n$，所以 QuickSort 算法在最坏情况下总的元素比较次数不超过 $n+(n-1)+(n-2)+\ldots+1=n(n+1)/2$，即QuickSort最坏情况下的时间复杂性为 $O(n^2)$。

② 平均时间复杂性。需要作如下假定。

- 待排序数组没有重复元素。
- 在待划分数组中按均匀分布随机选取划分元素。

根据以上假定，划分元素在位置 m（即将 $X[1\ ..\ n]$ 划分为 $X[1\ ..\ m-1]$ 和 $X[m+1\ ..\ n]$）的概率是 $1/n$，其中 $m=1,2,...,n$。设 $C(n)$ 是对 $X[1\ ..\ n]$ 排序时所需的元素比较次数，则

$$C(n)=\left[(1/n)\sum_{m=1}^{n}(C(m-1)+C(n-m))\right]+n \tag{1}$$

其中，n 是第一次划分时所需要的元素比较次数。显然，$C(0)=C(1)=0$。将(1)式两边乘以 n，得

$$nC(n)=2[C(0)+C(1)+...C(n-2)]+C(n-1)+n^2 \tag{2}$$

用 $n-1$ 代换(2)式中的 n 得

$$(n-1)C(n-1)=2[C(0)+C(1)+...C(n-3)]+C(n-2)+(n-1)^2 \tag{3}$$

用(2)式减去(3)式，得

$$\begin{aligned} nC(n)-(n-1)C(n-1) &= C(n-2)+C(n-1)+(2n-1) \\ &\le 2C(n-1)+2n \end{aligned} \tag{4}$$

由(4)式和 $(n+1)/n \le n/(n-1)$ 可得

$$\begin{aligned} C(n)/n &\le (n+1)C(n-1)/n^2+2/n \\ &\le C(n-1)/(n-1)+2/n \end{aligned} \tag{5}$$

反复用(5)式代换 $C(n-1)$、$C(n-2)$、...，得到

$$\begin{aligned} C(n)/n &\le C(n-1)/(n-1)+2/n \\ &\le C(n-2)/(n-2)+[2/(n-1)+2/n] \\ &\le \dots \\ &\le C(1)/1+[2/2+2/3+\dots+2/n] \\ &= 2(1/2+1/3+\dots+1/n) \\ &\le 2\int_1^n \frac{1}{x}dx \\ &= 2\ln n \end{aligned}$$

所以，$C(n) \le 2n\ln n$，从而，$C(n)=O(n\log n)$。

由以上分析可知，快速排序与归并排序具有相同的平均时间复杂性。但实际表现有所不同，快速排序一般要比归并排序用时少。

【注】在快速排序中，如果划分元素是随机选取的，则几乎可以保证不会出现最坏情况，几乎都在平均时间内完成。算法的改装在概率算法一章中介绍。

6. 测试*

① 调用方法。可使用如下形式调用。

```
template<class T> // 对数组X[n]排序
void QuickSort(T X[], int n)
{   QuickSort(X, 0, n);
}
```

② 测试程序。使用数组{6, 7, 9, 8, 4, 3, 2, 9, 6}。

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << to_string(X, n) << endl;
    QuickSort(X, n);
    cout << to_string(X, n) << endl;
}
```

```
6 7 9 8 4 3 2 9 6
2 3 4 6 6 7 8 9 9
```

5.3.3 以比较为基础的排序时间的下界

1. 二叉比较树

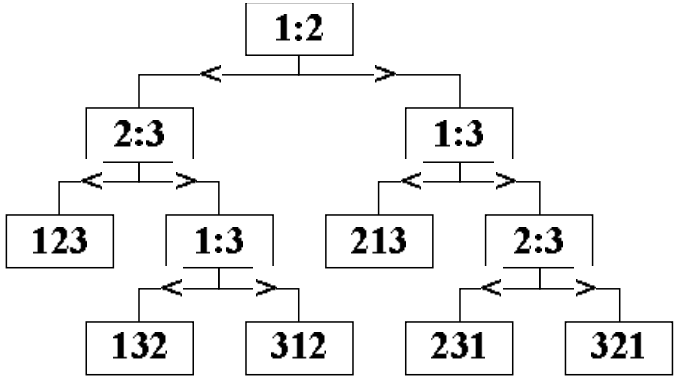
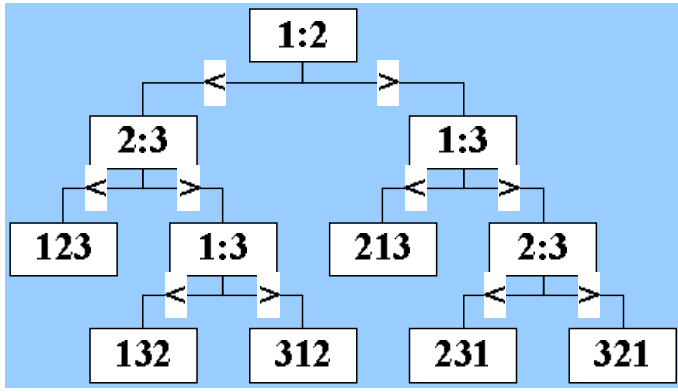


图5-4 n=3时的比较树

类似于估计搜索算法的时间下界，也可以用树来模拟排序算法（如图5-4所示），考虑最坏情况下的时间下限。假定数组中没有重复元素。在树的内部顶点上，算法执行一次比较，并根据比较的结果移向某一个孩子。由于每两个元素 $A[i]$ 和 $A[j]$ 的比较只有两种可能： $A[i]<A[j]$ 或 $A[i]>A[j]$，所以这棵树是一棵二叉树。当 $A[i]<A[j]$ 时进入左分支，当 $A[i]>A[j]$ 进入右分支。外部顶点表示算法终止。从根到外顶点的每一条路径分别与一种唯一的排列相对应。由于 n 个不同元素的不同排列共有 $n!$ 个，因此比较树至多有 $n!$ 个外部顶点。



2. 二叉树的一个性质

【引理】在二叉树的第 i 层上最多有 2^{i-1} 个结点（$i \geq 1$）。

3. 排序算法的时间下界

通过观察比较树可知，由根到外顶点的路径描述了该外顶点对应排列的生成过程，路径的长度（边数）就是经历的比较次数。因此，比较树中最长路径的长度 k（此时比较树的深度为 $k+1$）就是算法在最坏情况下所做的比较次数。要得到所有以比较为基础的排序算法在最坏情况的时间下界，只需求出这些算法对应的比较树的最小高度。根据二叉树的性质，如果比较树的深度是 $k+1$，则该二叉树的外顶点（最多位于第 $k+1$ 层）至多是 2^k 个。于是，最坏情况下的最少比较次数 k 满足 $n! \leq 2^k$。易知，当 $n \geq 4$ 时，有 $n/2-1 \geq n/4$ 和 $\log(n/2) \geq (1/2)\log n$ 成立。从而，由 $n! \geq n(n-1)\ldots(n/2) \geq (n/2)^{n/2-1}$ 可得

$$k \geq \log(n!) \geq (n/2-1)\log(n/2) \geq (1/8)n\log n$$

这说明，以比较为基础的排序问题的时间复杂性为 $n\log n$。

5.3.4 实际使用的排序方法举例

1. 排序

```
#include "InsertionSort.h" // 插入排序
#include "Partition.h" // 划分程序
#include "algorithm.h"
```

```
const int __SORT__MAX = 4; // VC 6.0规定_SORT_MAX=16
```

```
template<class T>
void Sort(T X[], int low, int up)
{   int sz = up - low;
    if(sz <= __SORT__MAX) // 插入排序
        InsertionSort(X + low, sz);
    else
    {   int m = Partition(X, low, up); // 划分位置
        Sort(X, low, m); // 左侧排序
        Sort(X, m + 1, up); // 右侧排序
    }
}
```

2. 测试*

```
template<class T>
void Sort(T X[], int n) // 对数组X[n]排序
{   Sort(X, 0, n);
}
```

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << to_string(X, n) << endl;
    Sort(X, n);
    cout << to_string(X, n) << endl;
}
```

```
6 7 9 8 4 3 2 9 6
2 3 4 6 6 7 8 9 9
```

5.4 选择算法

5.4.1 问题描述和分析

已知数组 $A[0..(n-1)]$，试确定其中第 k 小的元素（ $0 \le k < n$ ）。最容易想到的算法是首先将数组排序，然后从排好序的数组中选取第 k 个元素。但这样的算法在最坏情况下至少是 $n\log n$ 。

实际上，可以设计出在平均情况下时间复杂度为 n 的算法。为此，考察Partition算法。假设在一次划分中，划分元素 v 处于第 m 位置。如果 $k=m$，则划分元素 v 就是第 k 小元素；如果 $k<m$，则第 k 小元素在新数组 $A[1..(m-1)]$ 中；如果 $k>m$，则第 k 小元素在新数组 $A[(m+1)..(n-1)]$ 中。

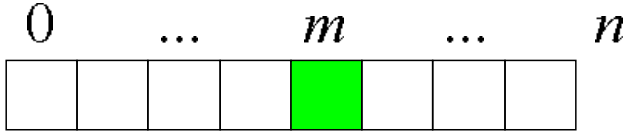
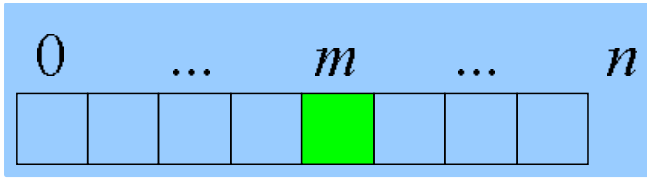


图5-5　选择算法分析



5.4.2 采用划分的选择算法

在数组X[n]中找第k($0 \le k < n$)小元素，即X[k]，其余元素按划分元素为X[k]的划分规则存放。

```
#include "Partition.h" // 划分程序
#include "algorithm.h"

template<class T> // 在数组X[n]中找第k(0 <= k < n)小元素, 即X[k]
const T &PartSelect(T X[], int n, int k)
{   int low = 0, up = n; // 待查找区间为[low, up)
    for(;;)
    {   int m = Partition(X, low, up); // 划分元素位置
        if(k == m) return X[m]; // m是第m(0 <= m < n)小元素的位置
        else if(k < m) up = m; // 左侧继续寻找
        else low = m + 1; // 右侧继续寻找
    }
}
```

5.4.3 复杂性分析

1. 最坏情况

在最坏情况下，每次调用Partition(i, j)耗时 $O(j-i)$。而下一次划分涉及的元素至少减少1个。在最初的划分中，$i=0$，$j=n$，因此，至多需要 n 次划分。所以，PartSelect在最坏情况下的时间复杂度为 $O(n^2)$。

2. 平均情况

设 $T(n)$ 是选择算法作用于含 n 个元素的数组的平均耗时，显然，$T(n)$ 是单调递增的。因为 $T(n)=O(n^2)$，所以存在 $c_0>0$ 和 $n_0\geq 1$，使得当 $n\geq n_0$ 时，$T(n)\leq c_0 n^2$。

设PartSelect中循环体执行一次（包含进行一次划分）需耗时 $C(n)$。易知 $C(n)=O(n)$，即存在 $c_1>0$ 和 $n_1\geq 1$，使得当 $n\geq n_1$ 时，$C(n)\leq c_1 n$。

令 $c=\max(c_0,c_1,T(n_0),T(n_1))$，则当 $n\geq 1$ 时，$C(n)\leq cn$，$T(n)\leq cn^2$。

根据与快速排序相同的假定，可知对含有 n 个元素的数组进行划分时，划分元素在 i 位置的概率为 $1/n$，$i=1,...,n$。

而且，在最坏情况下，第 k 小元素总是在元素较多的子数组中。

综合上述分析，可得

$$
\begin{aligned}
T(n) &\le C(n) + (1/n)\sum_{i=1}^{n} T(\max(i-1, n-i)) \\
&\le C(n) + (2/n)T(\lfloor n/2 \rfloor) + (2/n)\sum_{i=\lfloor n/2 \rfloor+1}^{n-1} T(i) \\
&\le cn + (2/n)c(n/2)^2 + (2/n)\sum_{i=\lfloor n/2 \rfloor+1}^{n-1} T(i) \\
&\le 2cn + (2/n)\sum_{i=\lfloor n/2 \rfloor+1}^{n-1} T(i)
\end{aligned}
$$

证明当 $n \ge 2$ 时，$T(n) \le 8cn$。首先，$T(2) \le 4c + T(1) \le 5c \le 8cn$。其次，设当 $2 \le n < m$ 时，$T(n) \le 8cn$。从而

$$
\begin{aligned}
T(m) &\le 2cm + (2/m)\sum_{i=\lfloor m/2 \rfloor+1}^{m-1} T(i) \\
&\le 2cm + (16c/m)\sum_{i=\lfloor m/2 \rfloor+1}^{m-1} (i) \qquad \{\because\ T(i) \le 8ci\} \\
&= 2cm + (16c/m)(m + \lfloor m/2 \rfloor)(m - \lfloor m/2 \rfloor - 1)/2 \\
&\le 2cm + (8c/m)(m + m/2)(m - m/2) \\
&= 8cm
\end{aligned}
$$

因此，选择算法可以在 $O(n)$ 平均时间内从 n 个元素中找出第 k 小的元素。

【注】在选择算法中，如果划分元素是随机选取的，则几乎可以保证不会出现最坏情况，几乎都在平均时间内完成。算法的改装在概率算法一章中介绍。

5.4.4 测试程序*

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << to_string(X, n) << endl;
    cout << PartSelect(X, n, 4) << endl;
    cout << to_string(X, n) << endl;
}
```

```
6 7 9 8 4 3 2 9 6
6
4 3 2 6 6 7 8 9 9
```

5.5 集合的顺序表示*

5.5.1 集合的性质和基本运算

集合是由一些不同的对象构成的一个整体。集合中的每一个对象都称为该集合的一个元素。若对象 x 是集合 A 的元素，则记作 $x \in A$，否则记作 $x \notin A$。

集合中的元素有下列三个重要特性。

- 确定性。对象 x 是否是集合 A 的元素是确定的。
- 互异性。集合中的每个元素都只能在该集合中出现一次。
- 无序性。集合的元素之间没有先后之分，地位平等。

对于两个集合之间的运算，通常会考虑下列几种基本运算。

- 相等。两个集合的元素完全相同。
- 包含。集合 X 包含集合 Y 表示集合 Y 的所有元素都在集合 X 中。
- 并。集合 A 与集合 B 的并集 $A \cup B = \{x \mid x \in A \operatorname{or} x \in B\}$。
- 交。集合 A 与集合 B 的交集 $A \cap B = \{x \mid x \in A \operatorname{and} x \in B\}$。
- 差。集合 A 与集合 B 的差集 $A - B = \{x \mid x \in A \operatorname{and} x \notin B\}$。

5.5.2 表示方法

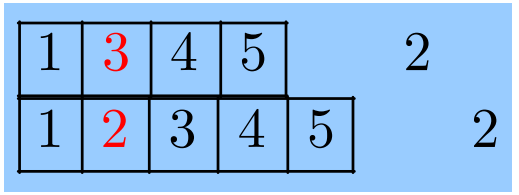
集合的存储结构有多种表示方法，这里只介绍集合的顺序表示，适用于集合中每2个元素都能比较大小的情况。如果集合中的每2个元素都能比较大小，则根据集合中元素的三个重要特性，可以考虑用一个有序的不含重复元素的一维数组buf[n]来表示集合，其中n是允许存入集合中元素的最大数目。集合中的第0号元素（最小元素）存放在buf[0]，第1号元素存放在buf[1]，第i号元素存放在buf[i]。

```
template<class T> //
class Set
{   T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 数组容量，实际元素个数
    // …
};
```

用这种表示方法有下列好处。

- 可以通过改装向有序数组中插入新元素的算法实现向集合中插入新元素的操作。
- 可以通过改装合并2个有序数组的算法实现集合的并、交、差和包含等运算。
- 可以通过改装折半搜索算法实现在集合中查找或删除某个元素的操作。

5.5.3 实质性操作



查找、插入和删除元素是集合的3个实质性操作。

1. 查找元素

使用折半搜索算法。

```
int find(const T &v) const
{   // 在集合中搜索v, 返回所在位置, -1表示未找到
    int low = 0, up = sz;
    while(low < up)
    {   int mid = (low + up) / 2;
        if(v == buf[mid]) return mid;
        else if(v < buf[mid]) up = mid;
        else low = mid + 1;
    }
    return -1; // 未找到v
} // O(log(sz))
```

2. 插入元素

可以通过改装向有序数组中插入新元素的算法实现向集合中插入新元素的操作。

```
void insert(const T &v) // 在当前集合中插入元素v
{   reserve(sz + 1); // 扩大容量
    int p = sz - 1; // 从尾部开始找到第一个不大于v的元素
    while(p >= 0 && buf[p] > v) --p;
    if(p >= 0 && buf[p] == v) return; // 重复元素
    for(int i = sz - 1; i > p; --i) // 移动元素
        buf[i + 1] = buf[i];
    buf[p + 1] = v, ++sz; // 插入
} // O(sz)
```

3. 删除元素

首先找到待删除元素的位置，然后删除该元素。

```
void erase(const T &v) // 从当前集合中删除元素v
{   int pos = find(v);
    if(pos < 0) return;
    --sz;
    for(int i = pos; i < sz; ++i) // 移动元素
        buf[i] = buf[i + 1];
} // O(sz)
```

5.5.4 一些其他基本操作

这里介绍集合的并、交、差和包含运算，这些运算都可以通过改装合并2个有序数组的算法实现。

1. 访问元素

使用指针访问，只读，begin()和end()分别得到首元素位置（就是最小元素的地址）和结束位置（地址）。

```
const T *begin() const { return buf; } // 首元素位置(地址)
const T *end() const { return buf + sz; } // 结束位置(地址)
```

2. 集合并

举例说明计算过程，如图5-6所示。

1	3	4	6	^	2	3	4	5	^	*W*
i					*j*					
	i				*j*					1
	i					*j*				1, 2
		i					*j*			1, 2, 3
			i					*j*		1, 2, 3, 4
			i						*j*	1, 2, 3, 4, 5
										1, 2, 3, 4, 5, 6

图5-6　集合并

```
template<class T> // 集合并, 即合并两个有序组
auto operator+(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    // begin()得到首元素位置(地址), end()得到结束位置(地址)
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) W.insert(*F1), ++F1; // 保留X中的元素
        else if(*F2 < *F1) W.insert(*F2), ++F2; // 保留Y中的元素
        else W.insert(*F1), ++F1, ++F2; // 只保留一份
    while(F1 != L1) W.insert(*F1), ++F1;
    while(F2 != L2) W.insert(*F2), ++F2;
    return W;
} // O(|X|+|Y|)(末尾插入)
```

3. 集合交

举例说明计算过程，如图5-7所示。

1	3	4	6	^	2	3	4	5	^	*W*
i					*j*					
	i				*j*					
	i					*j*				
		i					*j*			3
			i					*j*		3, 4
			i						*j*	3, 4

图5-7　集合交

```
template<class T> // 集合交
auto operator*(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) ++F1; // 不保留
        else if(*F2 < *F1) ++F2; // 不保留
        else W.insert(*F1), ++F1, ++F2; // 只保留相同的元素
    return W;
} // O(|X|+|Y|)(末尾插入)
```

4. 集合差

举例说明计算过程，如图5-8所示。

1	3	4	6	^	2	3	4	5	^	*W*
i					*j*					
	i				*j*					1
	i					*j*				1
		i					*j*			1
			i					*j*		1
			i						*j*	1
										1, 6

图5-8　集合差

```
template<class T> // 集合差
auto operator-(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) W.insert(*F1), ++F1; // 只保留X中的元素
        else if(*F2 < *F1) ++F2;
        else ++F1, ++F2; // 不保留相同的元素
    while(F1 != L1) W.insert(*F1), ++F1;
    return W;
} // O(|X|+|Y|)(末尾插入)
```

5. 集合包含

集合 X 是否包含集合 Y，即集合 Y 中的所有元素是否都在集合 X 中。举例说明判断过程，如图5-9所示。

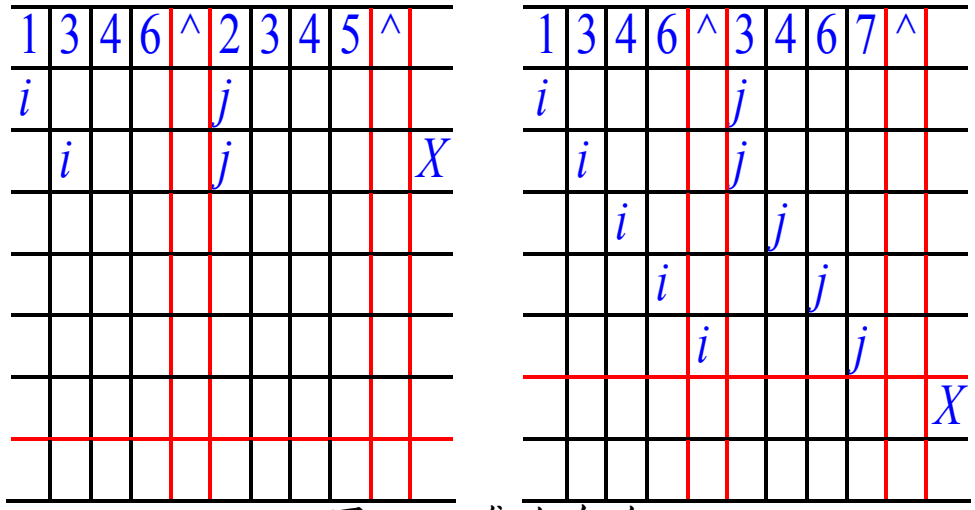


图5-9　集合包含

```
template<class T> // 一个集合是否包含另一集合
bool includes(const Set<T> &X, const Set<T> &Y)
{   auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) ++F1; // 与X中下一元素比较
        else if(*F2 < *F1) return false; // Y中有元素不在X中
        else ++F1, ++F2; // 考虑X和Y的下一元素
    return F2 == L2; // Y是否检查完毕
} // O(|X|+|Y|)
```

5.5.4 比较完整的C++实现

1. 预处理

```
// set.h
#pragma once
#include <algorithm> // copy_n, etc.
using namespace std;
using namespace rel_ops; // !=, <=, >, >=
```

2. 数据成员

```
template<class T> //
class Set
{   T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 数组容量，实际元素个数
```

3. 初始化

```
public:
    Set() // 空集合, 使用默认容量
    {   buf = new T[cap]; // 创建数组
    }
```

```
    Set(const Set &X): cap(X.cap), sz(X.sz) // 用其他集合初始化
    {   buf = new T[cap]; // 创建数组
        copy_n(X.buf, sz, buf); // 复制元素
    } // O(sz)
```

```
    ~Set()
    {   delete []buf; // 释放数组
    }
```

4. 复制集合

```
Set &operator=(const Set &X) // 复制集合
{   if(this == &X) return *this;
    reserve(X.sz); // 扩大容量
    copy_n(X.buf, X.sz, buf); // 复制元素
    sz = X.sz;
    return *this;
} // O(sz)
```

5. 查找元素

```
int find(const T &v) const
{   // 在集合中搜索v, 返回所在位置, -1表示未找到
    int low = 0, up = sz;
    while(low < up)
    {   int mid = (low + up) / 2;
        if(v == buf[mid]) return mid;
        else if(v < buf[mid]) up = mid;
        else low = mid + 1;
    }
    return -1; // 未找到v
} // O(log(sz))
```

6. 插入元素

```
void insert(const T &v) // 在当前集合中插入元素v
{   reserve(sz + 1); // 扩大容量
    int p = sz - 1; // 从尾部开始找到第一个不大于v的元素
    while(p >= 0 && buf[p] > v) --p;
    if(p >= 0 && buf[p] == v) return; // 重复元素
    for(int i = sz - 1; i > p; --i) // 移动元素
        buf[i + 1] = buf[i];
    buf[p + 1] = v, ++sz; // 插入
} // O(sz)
```

```
auto &operator<<(const T &v) // 方便使用的insert
{   insert(v);
    return *this;
}
```

7. 删除元素

```
void erase(const T &v) // 从当前集合中删除元素v
{   int pos = find(v);
    if(pos < 0) return;
    --sz;
    for(int i = pos; i < sz; ++i) // 移动元素
        buf[i] = buf[i + 1];
} // O(sz)
```

```
void clear() // 清空
{   sz = 0;
}
```

8. 大小

```
void reserve(int n) // 扩大容量, 新容量不小于n
{   if(cap >= n) return; // 无需扩容
    T *buf0 = buf; // 原数组
    while(cap < n) cap *= 2; // 计算新容量
    buf = new T[cap]; // 新数组
    copy_n(buf0, sz, buf); // 复制原有元素
    delete []buf0; // 释放原数组
}
```

```
int size() const // 集合大小
{   return sz;
}
```

```
bool empty() const // 是否空集
{   return sz <= 0;
}
```

9. 访问元素

使用指针访问，只读，begin()和end()分别得到首元素位置（就是最小元素的地址）和结束位置（地址）。

```
    const T *begin() const { return buf; } // 首元素位置(地址)
    const T *end() const { return buf + sz; } // 结束位置(地址)
};
```

10. 集合并

```
template<class T> // 集合并, 即合并两个有序组
auto operator+(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    // begin()得到首元素位置(地址), end()得到结束位置(地址)
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) W.insert(*F1), ++F1; // 保留X中的元素
        else if(*F2 < *F1) W.insert(*F2), ++F2; // 保留Y中的元素
        else W.insert(*F1), ++F1, ++F2; // 只保留一份
    while(F1 != L1) W.insert(*F1), ++F1;
    while(F2 != L2) W.insert(*F2), ++F2;
    return W;
} // O(|X|+|Y|)(末尾插入)
```

11. 集合交

```
template<class T> // 集合交
auto operator*(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) ++F1; // 不保留
        else if(*F2 < *F1) ++F2; // 不保留
        else W.insert(*F1), ++F1, ++F2; // 只保留相同的元素
    return W;
} // O(|X|+|Y|)(末尾插入)
```

12. 集合差

```
template<class T> // 集合差
auto operator-(const Set<T> &X, const Set<T> &Y)
{   Set<T> W;
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) W.insert(*F1), ++F1; // 只保留X中的元素
        else if(*F2 < *F1) ++F2;
        else ++F1, ++F2; // 不保留相同的元素
    while(F1 != L1) W.insert(*F1), ++F1;
    return W;
} // O(|X|+|Y|)(末尾插入)
```

13. 集合包含

```
template<class T> // 一个集合是否包含另一集合
bool includes(const Set<T> &X, const Set<T> &Y)
{   auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin(), L2 = Y.end();
    while(F1 != L1 && F2 != L2)
        if(*F1 < *F2) ++F1; // 与X中下一元素比较
        else if(*F2 < *F1) return false; // Y中有元素不在X中
        else ++F1, ++F2; // 考虑X和Y的下一元素
    return F2 == L2; // Y是否检查完毕
} // O(|X|+|Y|)
```

```
template<class T> // 一个集合是否是另一集合的真子集
bool operator<(const Set<T> &X, const Set<T> &Y)
{   return X.size() != Y.size() && includes(Y, X);
}
```

14. 集合相等

```
template<class T> // 两集合相等
bool operator==(const Set<T> &X, const Set<T> &Y)
{   if(X.size() != Y.size()) return false; // 大小不相等
    auto F1 = X.begin(), L1 = X.end(), F2 = Y.begin();
    while(F1 != L1 && *F1 == *F2) ++F1, ++F2; // 逐个比较
    return F1 == L1; // 全部检查完毕
} // O(max(|X|, |Y|))
```

5.5.5 C++ STL中的集合

在C++ STL中，使用类模板std::set表示集合。set使用与元素数目的对数成正比例的操作完成元素的查找、插入和删除。1.5.7节举例说明了set的使用方法。

5.5.6 C++ STL中的map

可以认为map是一种特殊的集合。模板类std::map<Key, T>描述可变长std::pair<const Key, T>类型元素序列。每个对的first分量是排序键且second分量是键的关联值。可以使用first分量作为下标访问序列中的元素，返回值为相应second分量的引用，当first分量无效时插入一个元素。map使用与元素数目的对数成正比例的操作完成元素的查找、插入和删除。1.5.8节举例说明了map的使用方法。

5.6 练习题

1、至少列出5种排序算法，并说明每种算法的最好时间复杂性、最坏时间复杂性和平均时间复杂性。

2、使用一种程序设计语言描述有序数组的折半搜索算法，并分析其时间复杂性。

3、编写程序实现归并排序算法和快速排序算法。

4、用大小分别为1000、2000、3000、4000、5000、6000、7000、8000、9000、10000的10个数组的排列来统计归并排序算法和快速排序算法的时间复杂性。

5、编写一个程序，使用分治方法实现在 n 个元素中同时寻找最大值和最小值。请分析该程序的时间复杂性。

6、证明在最坏情况快速排序的运行时间为 $\Theta(n^2)$。

7、编写一个混合快速排序和插入排序的混合排序程序。

8、编写一个混合归并排序和插入排序的混合排序程序。

9、编写一个子程序，用于将2个不含重复元素的有序数组X[m]和Y[n]合并成一个不含重复元素的有序数组W。

10、编写一个子程序，用于将一个整数数组分成两个区间，左侧区间存放原数组中的奇数，右侧区间存放原数组中的偶数，返回右侧区间的开始位置。