湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课程名称	算法设计与分析
章节、专题	分治方法
教学目标及基本要求	熟悉分治方法的基本思想和控制流程，能够使用分治方法求解搜索、排序和选择等问题。
教学重点	折半搜索，归并排序，快速排序
教学难点	归并排序，快速排序，选择算法
教学内容与时间分配	（1）分治方法的基本思想和控制流程（0.4课时） （2）折半搜索与集合的顺序表示（1.2课时） （3）归并排序与集合的基本运算（1.2课时） （4）快速排序与线性时间选择（1.2课时） （5）快速傅立叶变换（0.5课时） 共计4.5课时。
习　　题	第4.5.1节（基础知识题），第4.5.2节（实验题）。

第4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、线性时间选择以及集合的基本运算等。

- 分治方法的基本思想和控制流程
- 折半搜索与集合的顺序表示
- 归并排序与集合的基本运算
- 快速排序与线性时间选择
- 快速傅立叶变换

4.1 分治方法的基本思想和控制流程

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。如果分解后能够得到 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该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。

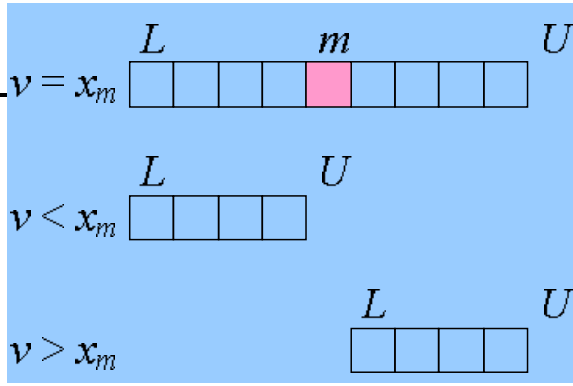
在分治方法中，使用的比较多的分解方法是将整个问题二分。

2. 控制流程

以二分为例说明。用DivideConquer(*X*, low, up)表示处理输入为*X*[low .. up)的问题。

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

其中，Small(*X*, low, up)用于判断规模为*n*=up - low的子问题是否小到很容易求解，Solution(*X*, low, up)用于直接求解小规模的子问题，Divide(*X*, low, up)是输入的分解函数，Combine(*x*, *y*)是解的合并函数。



4.2 折半搜索与集合的顺序表示

4.2.1 折半搜索

1. 问题

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. 算法描述

```
template<class T, class T2> // 在升序数组X中寻找v的位置
int search(T X[], int n, const T2 &v)
{   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))
```

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 $\Theta(\log n)$。

```
template<class T, class T2>
int search(T X[], int n, const T2 &v)
{   int low = 0, up = n;
    while(low < up)
    {   int m = (low + up) / 2;
        if(v == X[m]) return m;
        else if(v < X[m]) up = m;
        else low = m + 1;
    }
    return -1;
}
```

4.2.2 搜索元素的位置范围※

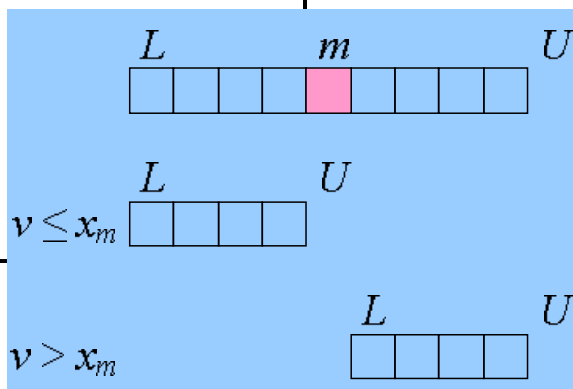
在升序数组X中寻找元素值范围[lower_value, upper_value)对应的元素位置范围[lower_bound, upper_bound)。通常，为了方便描述，将找到lower_bound的过程称为搜索下限，找到upper_bound的过程称为搜索上限。

1. 搜索下限

（1）问题描述。在升序数组X中寻找不小于v的第一个元素的位置。

（2）算法描述。

```
template<class T, class T2> // 寻找第一个不小于v的元素
int lower_bound(T X[], int n, const T2 &v)
{   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;
} // 耗时O(log(n))
```

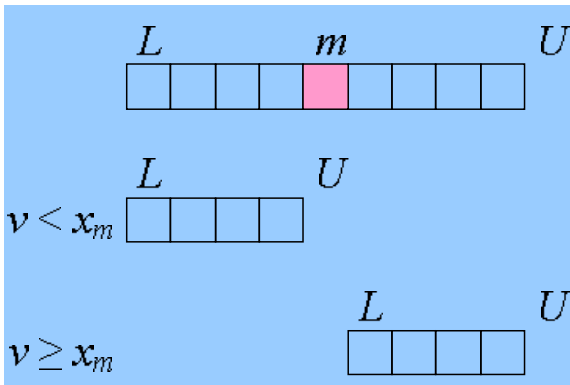


2. 搜索上限

（1）问题描述。在升序数组X中寻找大于v的第一个元素的位置。

（2）算法描述。

```
template<class T, class T2> // 寻找第一个大于v的元素
int upper_bound(T X[], int n, const T2 &v)
{   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;
} // 耗时O(log(n))
```



3. 复杂性分析

用 $T(n)$ 表示元素范围搜索所用的时间，则

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

其中，a 是一个常数。由简化Master定理可得 $T(n)=\Theta(\log n)$ 。

```
template<class T, class T2>
int lower_bound(T X[], int n, const T2 &v)
{   int low = 0, up = n;
    while(low < up)
    {   int m = (low + up) / 2;
        if(v <= X[m]) up = m;
        else low = m + 1;
    }
    return low;
}
```

4.2.3 集合的顺序表示※

1. 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且插入和删除操作比较少的情况。如果集合中的每2个元素都能比较大小，则可以考虑使用一个不含重复元素的升序数组buf来存储集合中的各个元素，其中，最小元素存放于buf[0]，次小元素存放于buf[1]，第i小元素存放于buf[i]。

C++ Boost和C++ 23中的flat_set类模板就是使用顺序表示实现的。

```
// Set.h
#pragma once
#include <vector>
#include <algorithm>
```

```
template<class T>
class Set
{   std::vector<T> buf; // 一维数组
```

```
public:
```

2. 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。

```
auto find(const T &v) const // 在集合中搜索v
{   // 找到第一个不小于v的元素(位置)
    auto p = std::lower_bound(buf.begin(), buf.end(), v);
    if(p != buf.end() and (*p) == v) return p; // 返回所在位置
    else return buf.end(); // end()表示未找到
} // O(log(n))
```

（2）插入元素。首先找到待插入位置，然后不重复地插入指定元素。这里特殊考虑了在尾部插入的情况。

```
void insert(const T &v) // 在当前集合中插入元素v
{   // 尝试插入尾部
    if(buf.empty() or buf.back() < v) // 若所有元素都小于v
    {   buf.push_back(v); // 插入尾部
        return;
    }
    // 找到第一个不小于v的元素(位置)
    auto p = std::lower_bound(buf.begin(), buf.end(), v);
    if(*p != v) buf.insert(p, v); // 只插入不重复元素
} // O(n)
```

（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
{   buf.erase(find(v));
} // O(n)
```

```
void clear() // 清空
{   buf.clear();
}
```

3. 其余常用操作

（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和结束位置。

```
auto begin() const { return buf.begin(); } // 首元素位置
auto end() const { return buf.end(); } // 结束位置
```

（2）大小与空集判断。

```
int size() const // 集合大小
{   return buf.size();
}
bool empty() const // 是否空集
{   return buf.empty();
}
};
```

4.3 归并排序与常用集合运算的实现

4.3.1 归并排序

【问题】已知含n个元素的数组X，将X中的元素升序排列。

1. 基本思想

归并排序算法采用分治方法，将要排序的数组分成两部分，先对各部分元素分别排序，然后将两部分元素按升序排列放在一个新数组中并复制回原数组。这一过程可能需要多次分解与合并，因此是一个递归过程。

2. 合并

将数组X[low .. up)的两个有序部分X[low .. m)和X[m .. up)合并到一个新数组中并复制回原数组X[low .. up)，使X[low .. up)有序。

（1）合并过程演示。如图4-1所示。

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4-1　合并过程演示

（2）算法描述。

```
#include "algorithm.h"
// 合并有序组X[low .. m)和X[m .. up), 使X[low .. up)有序
template<class T>
void Merge(T X[], int low, int m, int up)
{   int n = up - low; // 元素个数
    T W[n]; // 数组W用于记录合并结果
    int i = low, j = m, k = 0; // 左侧游标i, 右侧游标j, 结果游标k
    while(i < m and j < up) // 当两组都未取尽时
        if(X[i] < X[j]) // 左侧元素较小, 左侧元素加入结果
            W[k] = X[i], ++k, ++i;
        else // 右侧元素较小(含相等), 右侧元素加入结果
            W[k] = X[j], ++k, ++j;
    copy(X + i, X + m, W + k); // 左侧剩余部分添加到结果
    copy(X + j, X + up, W + k); // 右侧剩余部分添加到结果
    copy(W, W + n, X + low); // 合并结果复制回原数组
} // 耗时Θ(n)
```

容易看出，Merge的时间复杂度为$\Theta(n)$。

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

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> // 对数组X[low .. up)排序
void MergeSort(T X[], int low, int up)
{   if(up - low <= 1) return; // 最多一个元素
    int m = (low + up) / 2; // 分割点
    MergeSort(X, low, m); // 左侧排序
    MergeSort(X, m, up); // 右侧排序
    Merge(X, low, m, up); // 合并, 耗时O(n)
} // 耗时Θ(n*log(n))
```

4. 调用形式

由上述归并排序的主程序可以得到如下对数组*X*排序的归并排序算法。

```
template<class T> // 对数组X排序
void MergeSort(T X[], int n)
{   MergeSort(X, 0, n);
} // 耗时Θ(n*log(n))
```

5. 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。由简化Master定理可得 $T(n)=\Theta(n\log n)$ 。

【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 $\Theta(n\log n)$ 。

```
template<class T> // 对数组X[low .. up)排序
void MergeSort(T X[], int low, int up)
{   if(up - low <= 1) return; // 最多一个元素
    int m = (low + up) / 2; // 分割点
    MergeSort(X, low, m); // 左侧排序
    MergeSort(X, m, up); // 右侧排序
    Merge(X, low, m, up); // 合并，耗时O(n)
} // 耗时Θ(n*log(n))
```

4.3.2 几种常用集合运算的实现※

对于两个集合之间的运算，通常会考虑下列几种基本运算。

- 并。集合 A 与集合 B 的并集 $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$。
- 交。集合 A 与集合 B 的交集 $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$。
- 差。集合 A 与集合 B 的差集 $A - B = \{x \mid x \in A \text{ and } x \notin B\}$。
- 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。

1. 集合并

图4-4演示了集合并的计算过程。在合并时，相同的元素只保留一份。

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4-4　集合并演示

```
// Set.h续
template<class T> // 集合的并
auto operator+(const Set<T> &X, const Set<T> &Y)
{   auto I = X.begin(), Lx = X.end(), J = Y.begin(), Ly = Y.end(); // 源
    Set<T> W; // 结果
    while(I != Lx and J != Ly)
        if(*I < *J) // 保留X的元素
            W.insert(*I), ++I;
        else if(*J < *I) // 保留Y的元素
            W.insert(*J), ++J;
        else // 相同元素只保留一份
            W.insert(*I), ++I, ++J;
    while(I != Lx) W.insert(*I), ++I;
    while(J != Ly) W.insert(*J), ++J;
    return W;
} // O(|X|+|Y|)(末尾插入)
```

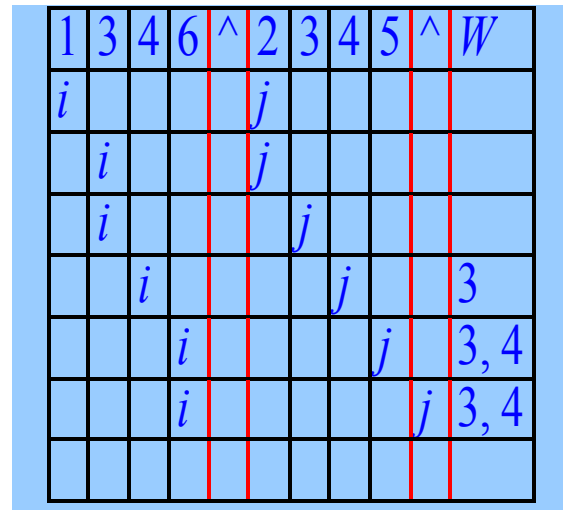
2. 集合交

图4-5演示了集合交的计算过程。根据元素比较结果，只保留相同的元素。

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4-5　集合交演示

```
template<class T> // 集合的交
auto operator*(const Set<T> &X, const Set<T> &Y)
{   auto I = X.begin(), Lx = X.end(), J = Y.begin(), Ly = Y.end(); // 源
    Set<T> W; // 结果
    while(I != Lx and J != Ly)
        if(*I < *J) ++I; // 不保留
        else if(*J < *I) ++J; // 不保留
        else W.insert(*I), ++I, ++J; // 只保留相同元素
    return W;
} // O(|X|+|Y|)(末尾插入)
```



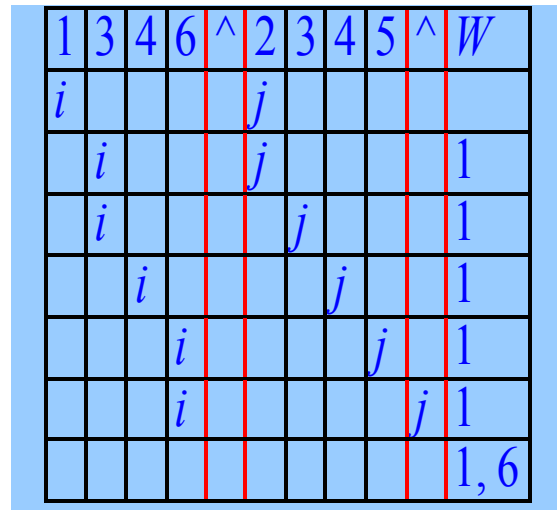
3. 集合差

图4-6演示了集合差的计算过程。根据元素比较结果，只保留仅在左侧出现的元素。

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4-6　集合差演示

```
template<class T> // 集合的差
auto operator-(const Set<T> &X, const Set<T> &Y)
{   auto I = X.begin(), Lx = X.end(), J = Y.begin(), Ly = Y.end(); // 源
    Set<T> W; // 结果
    while(I != Lx and J != Ly)
        if(*I < *J) W.insert(*I), ++I; // 只保留X的元素
        else if(*J < *I) ++J; // 不保留Y的元素
        else ++I, ++J; // 不保留相同元素
    while(I != Lx) W.insert(*I), ++I;
    return W;
} // O(|X|+|Y|)(末尾插入)
```



4. 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。图4-7演示了集合包含的判断过程。

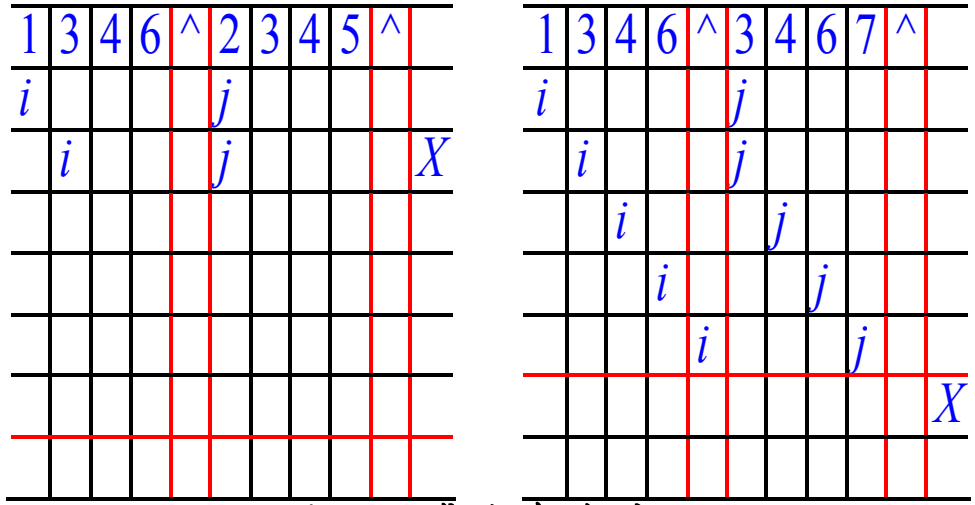
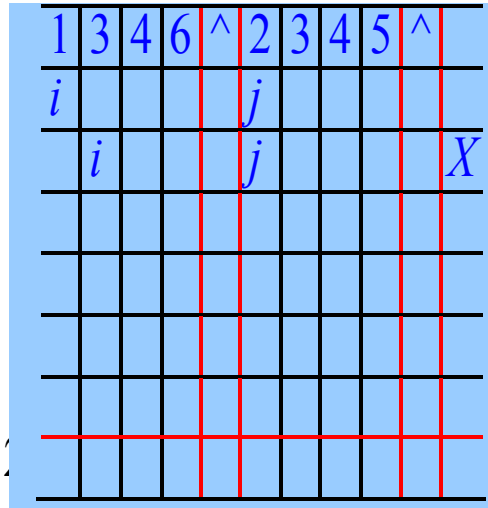


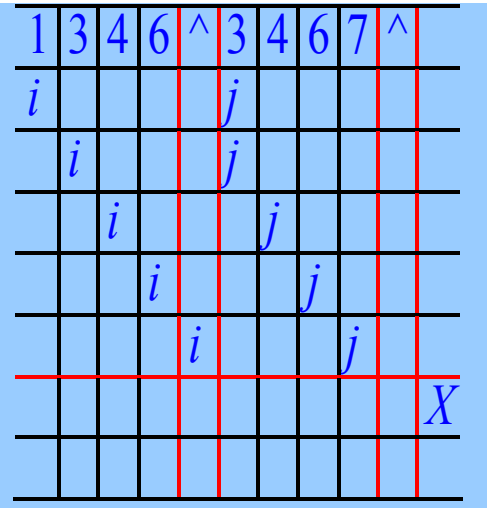
图4-7　集合包含演示

在图4-7左侧所示的比较过程中，出现了$Y[0]<X[1]$的情形，这说明$Y[0]$不在X中，这是因为如果$Y[0]=2$在X中，则只会与X中的2比较，不会与$X[1]=3$比较。

在图4-7右侧所示的比较过程完成后，Y中还有剩余元素$Y[3]$，这说明$Y[3]$不在X中，这是因为如果$Y[3]=7$在X中，则会与X中的7比较，不会出现在剩余部分。

```
template<class T> // 集合X包含了集合Y
bool includes(const Set<T> &X, const Set<T> &Y)
{   auto I = X.begin(), Lx = X.end(), J = Y.begin(), Ly = Y.end();
    while(I != Lx and J != Ly)
        if(*I < *J) ++I; // Y的当前元素可能在X中，与X的下一元素比较
        else if(*J < *I) return false; // Y的当前元素元素不在X中
        else ++I, ++J; // 考虑X和Y的下一元素
    return J == Ly; // Y是否检查完毕
} // O(|X|+|Y|)
```





5. 集合相等

集合X是否等于集合Y。首先比较大小，大小相同时对元素逐个比较。

```
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 I = X.begin(), L = X.end(), J = Y.begin();
    for(; I != L; ++I, ++J) // 逐个比较
        if(*I != *J) return false; // 对应元素不相同
    return true;
} // O(|X|)
```

```
template<class T> // 两集合不相等
bool operator!=(const Set<T> &X, const Set<T> &Y)
{   return not(X == Y);
} // O(|X|)
```

m	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

4.4 快速排序与线性时间选择

4.4.1 快速排序

另一种使用分治方法的排序算法是快速排序，是由计算机科学家C.A.R.Hoare提出的。

该算法首先将原数组划分成两个区间，使得左侧区间中的元素都小于右侧区间中的元素，然后分别对这两个区间中的元素排序。

一种典型的划分方法是从原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（划分元素不需要参与比较），小于划分元素的元素放在左侧区间里，大于或等于划分元素的元素放在右侧区间里，右侧区间的开始位置通常称为划分位置。

因为划分元素应该位于右侧区间，所以，若使用尾部元素作为划分元素，则划分元素已经位于右侧区间，划分时不需要移动。

在划分完成以后，因为划分元素是右侧区间的最小元素，在数组排序后该元素应该位于右侧区间的开始位置（划分位置），所以，在划分完成后应该将划分元素调整到划分位置。

1. 划分

（1）算法描述。下面给出划分算法的C++描述，保存在文件Partition.h中。

```
// Partition.h
#pragma once
#include "algorithm.h"
```

```
template<class T> // 划分，待划分区间为[low, up)，左侧存放小元素
int Partition(T X[], int low, int up)
{   int key = up - 1, m = low; // 划分元素选用尾部元素，用m记录划分位置
    for(int i = low; i < key; ++i) // 小元素调整到左侧(划分元素不参与比较)
        if(X[i] < X[key])
            swap(X[i], X[m]), ++m; // 左侧元素增加，划分位置后移
    swap(X[key], X[m]); // 将划分元素调整到划分位置
    return m; // 返回划分位置
} // 耗时O(n)
```

容易看出，若令n=up-low，则每一次调用Partition(low, up)只涉及n个元素，最多进行n次元素比较。

m	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

（2）举例说明。设原数组为{6, 7, 9, 8, 4, 3, 2, 9, 6}，low = 0，up = 9，则划分元素为key = 6，划分过程如图4-2所示，划分位置为3。

m	*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4-2　划分过程举例

容易看出，在划分完成后，将划分元素调整到划分位置可以使得划分位置就是原数组排序后划分元素所在的位置。

2. 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"
template<class T> // 对范围[low, up)排序
void QuickSort(T X[], int low, int up)
{   if(up - low <= 1) return; // 最多一个元素
    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)
```

3. 调用形式

由上述快速排序的主程序可以得到如下对数组*X*排序的快速排序算法。

```
template<class T> // 对数组X[n]排序
void QuickSort(T X[], int n)
{   QuickSort(X, 0, n);
} // 平均耗时O(n*log(n)), 最坏耗时O(n*n)
```

4.4.2 快速排序的复杂性

要分析QuickSort的时间复杂性，只需分析它的元素比较次数，因为其他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)只涉及up - low个元素，最多进行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，只涉及 n 个元素；第1层调用Partition两次，只涉及 $n-1$ 个元素（不涉及第0层选定的划分元素）；第 k 层调用Partition只涉及 $n-k$ 个元素。所以QuickSort算法在最坏情况下总的元素比较次数不超过 $\sum\{n-k\}_{k=0}^{n-1}=n(n+1)/2$，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。

根据以上假定可知，划分元素在位置m的概率是$1/n$，其中m=0, 1, …, n-1（如图4-3所示）。

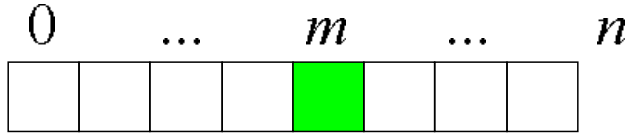


图4-3　划分位置

设$C(n)$是对$X[0..n)$排序需要的元素比较次数，则

$$\begin{aligned} C(n) &= (1/n)\sum\{C(m)+C(n-m-1)\}_{m=0}^{n-1}+n \\ &= (2/n)[C(0)+C(1)+\ldots+C(n-1)]+n \end{aligned} \tag{1}$$

其中，n是第0层调用划分程序需要的元素比较次数。显然，$C(0)=C(1)=0$。

上课时请勿吃喝，请勿讲话，请勿使用电话，请勿随意进出和走动。

将(1)式两边乘以 n，得

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

用 $n-1$ 代换(2)式中的 n 得

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

用(2)式减去(3)式，得

$$nC(n)-(n-1)C(n-1)=2C(n-1)+2n-1$$

从而

$$nC(n)=(n+1)C(n-1)+2n-1\le(n+1)C(n-1)+2(n+1)$$

两边除以 $n(n+1)$，可得

$$C(n)/(n+1)\le C(n-1)/n+2/n \tag{4}$$

$$C(n)=(2/n)[C(0)+C(1)+\ldots+C(n-1)]+n \tag{1}$$

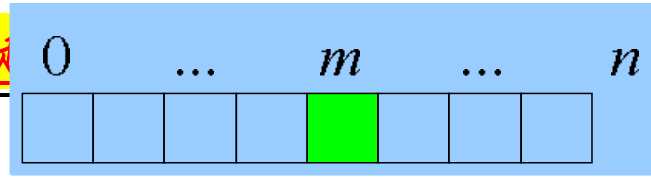
反复用(4)式代换 $C(n-1)$，$C(n-2)$，...，得到

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

所以，$C(n)\le 2(n+1)\ln n \le 4n\ln n = O(n\log n)$。

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。

$$C(n)/(n+1)\le C(n-1)/n+2/n \qquad (4)$$

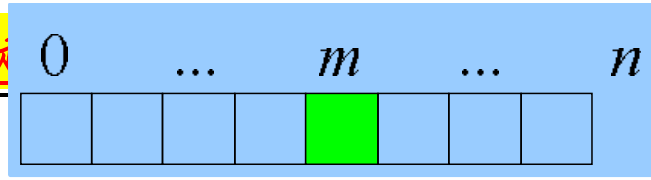


4.4.3 选择算法

1. 问题描述与分析

已知数组$X[0..n)$，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号元素。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$O(n)$的算法，通常称为线性时间选择算法。为此，考察Partition算法。如图4-3所示，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小的元素；如果$k<m$，则第k小的元素在新数组$X[0..m)$中；如果$k>m$，则第k小的元素在新数组$X[m+1..n)$中。



2. 算法描述

在数组$X[n]$中寻找第$k(0\le k<n)$小的元素，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
template<class T> // 在数组X[n]中找第k(0 <= k < n)小的元素，即X[k]
auto Select(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)小的元素的位置
        else if(k < m) up = m; // 左侧继续寻找
        else low = m + 1; // 右侧继续寻找
    }
}
```

3. 测试☼

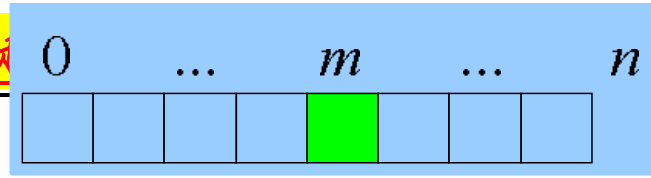
```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << Select(X, n, 4) << endl; // 6
    cout << to_string(X, n) << endl; // 4 3 2 6 6 7 8 9 9
}
```

4.4.4 选择算法的复杂性

要分析Select的时间复杂性，只需分析它的元素比较次数，因为其他运算的频率计数和元素比较次数有相同的数量级。

1. 最坏情况

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。在最坏情况下，第 k 小的元素总是在元素较多的子数组中，下一次划分涉及的元素至少减少1个。将过程Select按照划分来分层，第0层调用涉及 n 个元素；第1层调用最多涉及 $n-1$ 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Select算法在最坏情况下总的元素比较次数不超过 $n+(n-1)+(n-2)+\ldots+1=n(n+1)/2 \le n^2$，即Select最坏情况下的时间复杂性为 $O(n^2)$。



2. 平均情况

根据与快速排序相同的假定可知，对有n个元素的数组进行划分时，划分元素在i位置的概率为$1/n$，且下一次划分涉及的元素不超过$\max(i, n-i-1)$个，其中，$i=0, 1, ..., n-1$。

设$C(n)$是选择算法作用于含n个元素的数组时所需的平均元素比较次数，显然，$C(0)=C(1)=0$，且$C(n)\leq n^2$。

由第0层调用划分程序需要的元素比较次数不超过n和$C(n)$的含义可得

$$C(n) \leq n + \left(\sum \{C(\max(i, n-i-1)\}_{i=0}^{n-1} \right) \Big/ n$$

$$\leq n + \left(2\sum \{C(i)\}_{i=\lfloor n/2 \rfloor}^{n-1} \right) \Big/ n$$

下面使用数学归纳法证明当 $n \geq 1$ 时， $C(n) \leq 8n$ 。

0, 1,2,3,4　4,3,2,3,4
0,1,2,3,4,5　5,4,3,3,4,5

容易验证，当 $1 \le n < 8$ 时，$C(n) \le n^2 \le 8n$，所以只需考虑 $n \ge 8$ 的情形。假设当 $1 \le n < m$（$m \ge 8$）时，$C(n) \le 8n$，则当 $n = m$ 时，有

$$
\begin{aligned}
C(m) &\le m + \left(2\sum\left\{C(i)\right\}_{i=\lfloor m/2 \rfloor}^{m-1}\right)\Big/ m \\
&\le m + \left(16\sum\left\{i\right\}_{i=\lfloor m/2 \rfloor}^{m-1}\right)\Big/ m \qquad \left\{\because\ C(i) \le 8i\right\} \\
&= m + 16\left(m - 1 + \lfloor m/2 \rfloor\right)\left(m - 1 - \lfloor m/2 \rfloor + 1\right)/2/m \\
&\le m + 8\left(m - 1 + m/2\right)\left(m/2 + 1\right)/m \quad \left\{\because\ m - 1 - \lfloor m/2 \rfloor \le m/2\right\} \\
&\le 7m + 8 \\
&\le 8m \qquad \left\{\because\ m \ge 8\right\}
\end{aligned}
$$

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】“概率方法”一章介绍了快速排序和线性时间选择算法的随机化版本，如果不准备完整阅读“概率方法”一章，则可在了解相关概念后选择补充阅读下列小节并选作相关练习题。

- 9.1　概率算法的大致类型（选择阅读关于舍伍德算法的描述）
- 9.3.1　快速排序和线性时间选择
- 9.3.2　随机洗牌算法

4.5 快速傅立叶变换※

4.5.1 基本概念与性质

1. 单位根

给定复数 ω 和正整数 n，若 n 是使得 $\omega^n = 1$ 的最小正整数，则称 ω 为一个 n 次单位根。显然，$\omega = e^{-2\pi i/n}$ 和 $\omega = e^{2\pi i/n}$ 都是一个 n 次单位根（i 是虚数单位）。

容易证明，n 次单位根 ω 有下列性质。

（1）周期性。序列 $\ldots,\omega^{-2},\omega^{-1},\omega^0,\omega^1,\omega^2,\ldots$ 的周期是 n。

（2）不重复性。$\omega^0,\omega^1,\ldots,\omega^{n-1}$ 各不相同。

（3）共轭对称性。$\omega^{n-k} = \mathrm{conj}(\omega^k)$，$k$ 是整数。

（4）相反性。若 n 是偶数，则 $\omega^{n/2} = -1$，从而 $\omega^{k+n/2} = -\omega^k$。

（5）低次单位根。若正整数 k 能够整除 n，则 ω^k 是一个 n/k 次单位根。

2. 离散傅立叶变换的表达式

设 ω 为一个 n 次单位根，$f(x)$ 是一个以 n 为周期的离散函数（其中 $x \in Z$），则离散傅立叶变换的通用表达式可以写成

$$F(u) = \sum_{x=0}^{n-1} f(x)\omega^{ux} \quad , \quad u \in Z$$

容易证明，离散傅立叶变换具有下列性质。

- 周期性。$F(u+n) = F(u)$。由此可知，只需对 $u = 0,1,...,n-1$ 计算 $F(u)$。
- 共轭对称性。如果 $f(x)$ 是实数函数，则 $F(n-u) = \text{conj}(F(u))$。

3. 直接计算的算法描述

若直接使用 $F(u)=\sum_{x=0}^{n-1} f(x)\omega^{ux}$ 计算 $F(0),F(1),\ldots,F(n-1)$，则可获得下列参考程序描述的算法。

```
template<class Complex>
void MyDFT(Complex X[], int n, Complex w, Complex Y[])
{   // X[i]表示f(i), w是n次单位根, Y[u]表示F(u)
    Complex P[n]; //  保存w ^ i
    P[0] = 1; // w ^ 0
    for(int i = 1; i < n; ++i)
        P[i] = w * P[i - 1]; // w ^ i
    for(int u = 0; u < n; ++u)
    {   Y[u] = X[0];
        for(int x = 1; x < n; ++x)
            Y[u] += X[x] * P[u * x % n];
    }
}
```

容易看出，该算法耗时 $O(n^2)$。

4.5.2 快速算法

这里介绍一种耗时 $O(n\log n)$ 的快速方法。该方法要求 n 是2的正整数幂，若 n 不满足该要求，则可令 $n_e=2^{\lceil\log_2 n\rceil}$，并使用下述方法将 $f(x)$ 扩充为 $f_e(x)$ 后对 $f_e(x)$ 使用该方法进行计算。

- 若 $f(x)$ 是实数函数，则令

$$f_e(x)=\begin{cases}f(x) & x=1,2,\ldots,n-1\\ 0 & x=n,\ldots,n_e-1\end{cases}$$

- 若 $f(x)$ 不是实数函数，则令

$$f_e(x)=\begin{cases}f(x) & x=1,2,\ldots,n-1\\ \mathrm{conj}(f(n_e-x)) & x=n,\ldots,n_e-1\end{cases}$$

当 n 是偶数时，$\omega^{k+n/2}=-\omega^k$，ω^2 是 $n/2$ 次单位根

1. 算法构造

为了方便，令 $m=n/2$。

$$\begin{aligned}F(u)&=\sum_{x=0}^{n-1}f(x)\omega^{ux}\\&=\sum_{x=0}^{m-1}f(2x)\omega^{u(2x)}+\sum_{x=0}^{m-1}f(2x+1)\omega^{u(2x+1)}\\&=\sum_{x=0}^{m-1}f(2x)(\omega^2)^{ux}+\omega^u\sum_{x=0}^{m-1}f(2x+1)(\omega^2)^{ux}\end{aligned}$$

设 $f_0(x)=f(2x)$，$f_1(x)=f(2x+1)$，$\widehat{\omega}=\omega^2$，$F_0(u)=\sum_{x=0}^{m-1}f_0(x)\widehat{\omega}^{ux}$，

$F_1(u)=\sum_{x=0}^{m-1}f_1(x)\widehat{\omega}^{ux}$，则

$$\begin{aligned}F(u)&=F_0(u)+\omega^uF_1(u)\\F(u+m)&=F_0(u+m)+\omega^{u+m}F_1(u+m)\\&=F_0(u)-\omega^uF_1(u)\end{aligned}$$

$$F(u) = F_0(u) + \omega^u F_1(u)$$
$$F(u+m) = F_0(u) - \omega^u F_1(u)$$

2. 算法描述

可以使用下列参考程序描述。

```
template<class Complex>
void MyFFT(Complex X[], int n, Complex w, Complex Y[])
{   // X[i]表示f(i), w是n次单位根, Y[u]表示F(u)
    if(n <= 1) { Y[0] = X[0]; return; }
    int m = n / 2;
    Complex X0[m], X1[m], Y0[m], Y1[m]; // f(2x)和f(2x+1)的值, F0和F1
    for(int i = 0; i < m; ++i) X0[i] = X[2 * i], X1[i] = X[2 * i + 1];
    MyFFT(X0, m, w * w, Y0), MyFFT(X1, m, w * w, Y1);
    Complex wp = 1; // w ^ u, u = 0
    for(int u = 0; u < m; ++u)
    {   Y[u] = Y0[u] + wp * Y1[u], Y[u + m] = Y0[u] - wp * Y1[u];
        wp *= w; // w^(u + 1) = w ^ u×w
    }
}
```

【复杂性】设 $T(n)$ 是当输入规模为 n 时的耗时，则由算法描述容易看出，$T(n) = 2T(n/2) + \Theta(n)$。由简化Master定理可知，$T(n) = \Theta(n \log n)$。

4.5.3 一个快速计算示例

1. 实际应用中的离散傅立叶变换

在实际应用中，通常将离散傅立叶正变换定义为

$$F(u)=\sum_{x=0}^{n-1} f(x)\omega^{ux}\ ,\quad u\in Z$$

离散傅立叶逆变换定义为

$$f(x)=(1/n)\sum_{u=0}^{n-1} F(u)\,\mathrm{conj}(\omega)^{ux}\ ,\quad x\in Z$$

$$F(u)=\sum_{x=0}^{n-1} f(x)\omega^{ux}$$

2. 测试程序

```
#include<complex> // 复数
#include"algorithm.h"
typedef complex<double> Complex;
const Complex PI_I = 3.14159265358979323846264338327951;
ostream &operator<<(ostream &out, const Complex &X)
{   return out << noshowpos << real(X) << showpos << imag(X) << 'i';
}
int main()
{   int n = 5, m = 1 << (int)ceil(log2(n));
    Complex X[m] = {1, 2, 3, 4, 5, 0, 0, 0}, Y[m];
    auto w = exp(-2.0 * PI_I / (double)m);
    MyFFT(X, m, w, Y); // 正变换
    cout << to_string(Y, m, ", ") << endl;
    MyFFT(Y, m, conj(w), X); // 逆变换
    for(auto &e : X) e /= m; // 缩放结果
    cout << to_string(X, m, ", ") << endl;
}
```

$$f(x)=(1/n)\sum_{u=0}^{n-1} F(u)\,\mathrm{conj}(\omega)^{ux}$$

15+0i, -5.41421-7.24264i, 3+2i, -2.58579-1.24264i, 3+0i, -2.58579+1.24264i, 3-2i, -5.41421+7.24264i
1+0i, 2-1.11022e-16i, 3+1.11022e-16i, 4+2.22045e-16i, 5+0i, 1.11022e-16+1.11022e-16i, 0-1.11022e-16i, 2.22045e-16-2.22045e-16i

4.5 练习题

4.5.1 基础知识题

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，并分析其时间复杂性。

4. 编写程序实现快速排序算法，并证明在最坏情况下快速排序的运行时间为$\Theta(n^2)$。

5. 编写程序实现线性时间选择算法，并证明在最坏情况下线性时间选择算法的运行时间为$\Theta(n^2)$。

4.5.2 实验题

1. 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。

2. 用大小分别为1000、2000、3000、4000、5000、6000、7000、8000、9000和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。

3. 实际使用的排序方法通常混合使用了快速排序和插入排序。当问题实例的规模很大时使用快速排序，规模较小时使用插入排序。请使用一种程序设计语言编写一个混合快速排序和插入排序的混合排序程序。

4. 请使用一种程序设计语言编写一个混合归并排序和插入排序的混合排序程序。

5. 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。

6. 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。

4.5.3 进阶训练题

编写一个子程序，用于将一个数组根据指定条件分成两个区间，左侧区间存放原数组中满足指定条件的元素，右侧区间存放原数组中不满足指定条件的元素，返回右侧区间的开始位置。其中，划分条件由子程序的某个参数指定。