

第七章 内排序



大纲

- **7.1 基本概念**

- **7.2 三种 $O(n^2)$ 的简单排序**

 - 插入排序

 - 直接插入排序

 - 二分法插入排序

 - 冒泡排序

 - 选择排序

- **7.3 Shell排序**



大纲（续）

- **7.4 基于分治法的排序**
 - 快速排序
 - 归并排序
- **7.5 堆排序**
- **7.6 分配排序和基数排序**
- **7.7 各种排序算法的理论和实验时间代价**
- **7.8 排序问题的下限**



7.1 基本概念

- **记录(Record):** 结点, 进行排序的基本单位
- **关键码(Key):** 唯一确定记录的一个或多个域
- **排序码(Sort Key):** 作为排序运算依据的一个或多个域
- **序列(Sequence):** 线性表, 由记录组成的集合



7.1基本概念（续）

- **排序(Sorting)** — 将序列中的记录按照排序码特定的顺序排列起来，即排序码域的值具有不减（或不增）的顺序。



排序问题

- 给定一个序列 $\mathbf{R} = \{r_1, r_2, \dots, r_n\}$, 其排序码分别为 $\mathbf{k} = \{k_1, k_2, \dots, k_n\}$
- 排序的目的就是将 \mathbf{R} 中的记录按照特定的顺序重新排列, 形成一个新的有序序列 $\mathbf{R}' = \{r'_1, r'_2, \dots, r'_n\}$
 - 相应排序码为 $\mathbf{k}' = \{k'_1, k'_2, \dots, k'_n\}$
 - 其中 $k'_1 \leq k'_2 \leq \dots \leq k'_n$ 或 $k'_1 \geq k'_2 \geq \dots \geq k'_n$, 前者称为不减序, 后者称为不增序。



7.1 基本概念（续）

- **内排序(Internal Sorting):** 整个排序过程中所有的记录都可以直接存放在内存中
- **外排序(External Sorting):** 内存无法容纳所有记录，排序过程中还需要访问外存



7.1 基本概念（续）

- “正序”序列：待排序序列正好符合排序要求
- “逆序”序列：把待排序序列逆转过来，正好符合排序要求



7.1 基本概念（续）

- “稳定的” (stable) 排序算法：
如果存在多个具有相同排序码的记录，经过排序后这些记录的相对次序仍然保持不变。



排序算法的分类

- 简单排序
 - 插入排序(**Insert sort**)
 - 直接选择排序(**Selection sort**)
 - 冒泡排序(**Bubble sort**)
- **Shell排序(Shell sort)**



排序算法的分类（续）

- 分治排序
 - 快速排序(**Quicksort**)
 - 归并排序(**Mergesort**)
- 堆排序(**Heapsort**)
- 分配排序 (**Binsort**)



排序算法的衡量标准

- 时间代价：记录的比较和交换次数
- 空间代价
- 算法的复杂度



总的排序类

```
template <class Record,class Compare>  
class Sorter{    //总排序类  
protected:  
    //交换数组中的两个元素  
    static void swap(Record Array[],int i,int j);  
  
public:  
    //对数组Array进行排序  
    void Sort(Record Array[],int n);  
    //输出数组内容  
    void PrintArray(Record array[], int n);  
};
```



Compare类

- **Compare**类是用来比较记录的排序码，把它单独定义成模板参数，是为了方便针对不同类型的排序码进行比较。为了简便起见，我们只讨论整数排序的例子。



int_intCompare 类

```
class int_intCompare{
```

```
//比较两个整型记录大小
```

```
public:
```

```
static bool lt(int x,int y)      {return x<y;}
```

```
static bool eq(int x,int y)     {return x==y;}
```

```
static bool gt(int x,int y)     {return x>y;}
```

```
static bool le(int x,int y)     {return x<=y;}
```

```
static bool ge(int x,int y)     {return x>=y;}
```

```
};
```



swap函数

```
template <class Record,class Compare>  
void Sorter<Record,Compare>::  
swap(Record Array[],int i,int j)  
{ //交换数组中的两个元素  
    Record TempRecord = Array[i];  
    Array[i] = Array[j];  
    Array[j] = TempRecord;  
}
```




PrintArray函数

```
template <class Record,class Compare>  
void Sorter<Record,Compare>::  
PrintArray(Record Array[], int n)  
{ //输出数组内容  
    for(int i=0;i<n;i++)  
        cout<<Array[i]<<" ";  
    cout<<endl;  
}
```



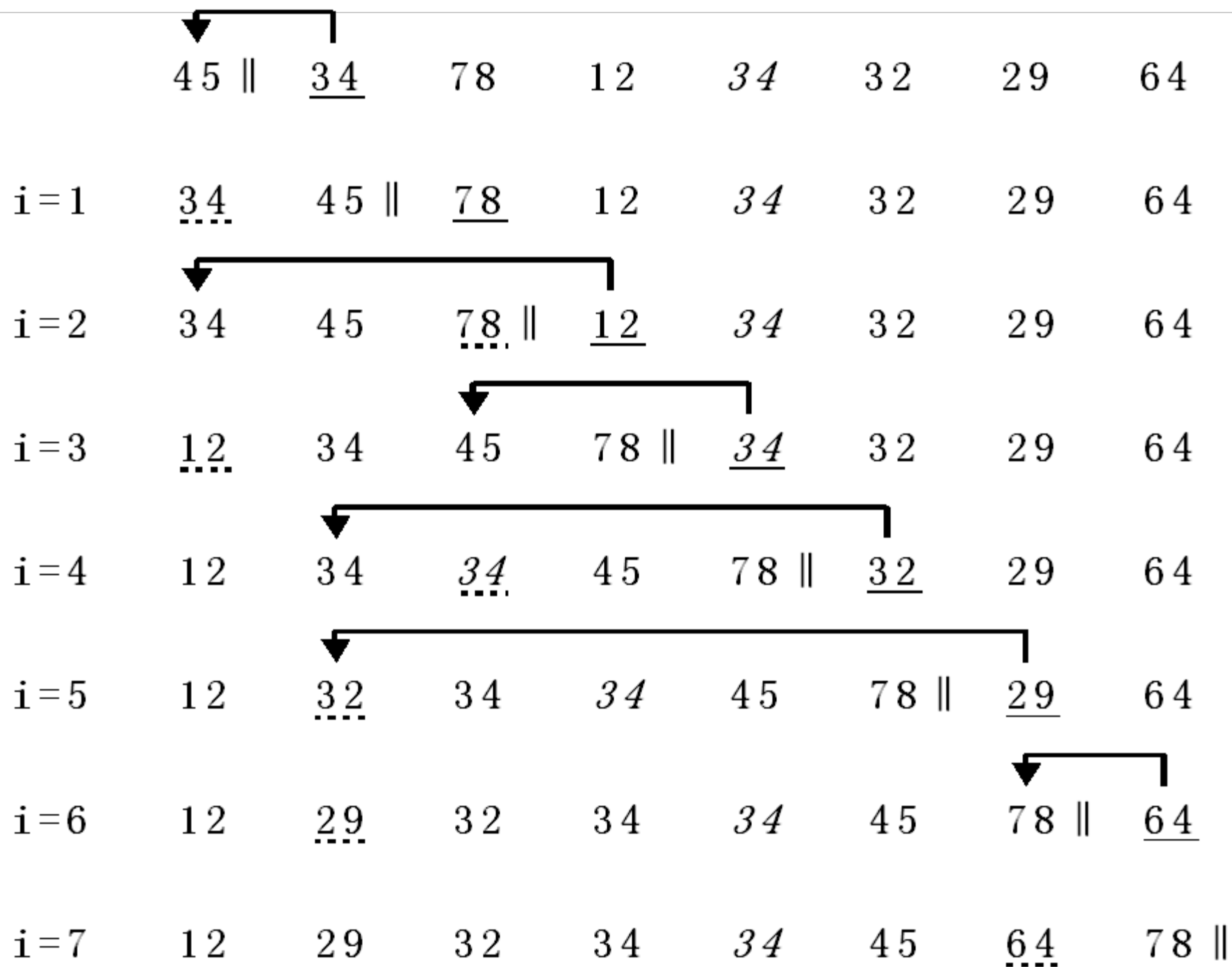
7.2 三种 $O(n^2)$ 的简单排序

- 插入排序(**Insert Sort**)
- 冒泡排序(**Bubble Sort**)
- 选择排序 (**Selection Sort**)



7.2.1 插入排序

- 算法思想：
 - 逐个处理待排序的记录，每个新记录都要与前面那些已排好序的记录进行比较，然后插入到适当的位置。





插入排序类

```
template <class Record,class Compare>  
class InsertSorter:public  
    Sorter<Record,Compare>{};
```



直接插入排序

```
template <class Record,class Compare>  
class StraightInsertSorter:public  
    InsertSorter<Record,Compare>  
{ //直接插入排序类  
public:  
    void Sort(Record Array[],int n);  
};
```

```

template <class Record,class Compare>
void StraightInsertSorter<Record,Compare>::
Sort(Record Array[], int n)
{ //Array[]为待排序数组，n为数组长度
  for (int i=1; i<n; i++) // 依次插入第i个记录
  { //依次与前面的记录进行比较，发现逆置就交换
    for (int j=i;j>0;j--){
      if (Compare::lt(Array[j], Array[j-1]))
        swap(Array, j, j-1);
      else break; //此时i前面记录已排序
    }
  }
}

```



算法分析

- 稳定
- 空间代价: $\Theta(1)$
- 时间代价:
 - 最佳情况: $n-1$ 次比较, 0 次交换, $\Theta(n)$
 - 最差情况: 比较和交换次数为
$$\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$$
 - 平均情况: $\Theta(n^2)$



优化的插入排序算法

```
template <class Record,class  
    Compare>  
class ImprovedInsertSorter:public  
    InsertSorter<Record,Compare>  
{ //优化的插入排序类  
public:  
    void Sort(Record Array[],int n);  
};
```



```
template <class Record,class Compare>  
void
```

```
ImprovedInsertSorter<Record,Compare>::
```

```
Sort(Record Array[], int n)
```

```
{ //Array[]为待排序数组， n为数组长度
```

```
Record TempRecord; // 临时变量
```

```
// 依次插入第i个记录
```

```
for (int i=1; i<n; i++)
```

```
{
```

```
TempRecord=Array[i];
```

```
//从i开始往前寻找记录i的正确位置
```

```
int j = i-1;
```



```
//将那些大于等于记录i的记录后移
while ((j>=0) &&
(Compare::lt(TempRecord, Array[j])))
{
    Array[j+1] = Array[j];
    j = j - 1;
}
//此时j后面就是记录i的正确位置，回填
Array[j+1] = TempRecord;
}
```

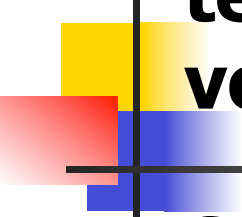


二分法插入排序

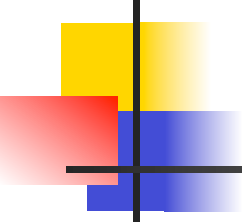
- 算法思想：
 - 在插入第 i 个记录时，前面的记录已经是有序的了，可以用二分法查找第 i 个记录的正确位置。



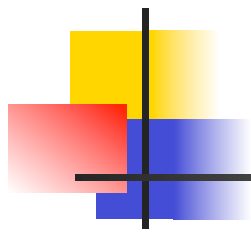
```
template <class Record,class  
    Compare>  
class BinaryInsertSorter:public  
    InsertSorter<Record,Compare>  
{ //二分法插入排序类  
public:  
    void Sort(Record Array[],int n);  
};
```



```
template <class Record,class Compare>
void
BinaryInsertSorter<Record,Compare>::
Sort(Record Array[], int n)
{ //Array[]为待排序数组， n为数组长度
    Record TempRecord;           //临时变量
    //记录已排好序序列的左、右、中位置
    int left,right,middle;
    //依次插入第i个记录
    for (int i=1;i<n;i++)
    {
        //保存当前待插入记录
        TempRecord = Array[i];
```



```
//记录已排好序序列的左右位置
left = 0; right = i-1;
//开始查找待插入记录的正确位置
while(left <= right)
{
    //中间位置
    middle = (left+right)/2;
    //如果待插入记录比中间记录小,
    // 就在左一半中查找,
    // 否则在右一半中查找
    if
    (Compare::lt(TempRecord,Array[mi
ddle]))
        right = middle-1;
```



```
        else left = middle+1;
    }
    //将前面所有大于当前待插入记录的记录后移
    for(int j = i-1; j >= left; j --)
        Array[j+1] = Array[j];
    //将待插入记录回填到正确位置
    Array[left] = TempRecord;
}
}
```



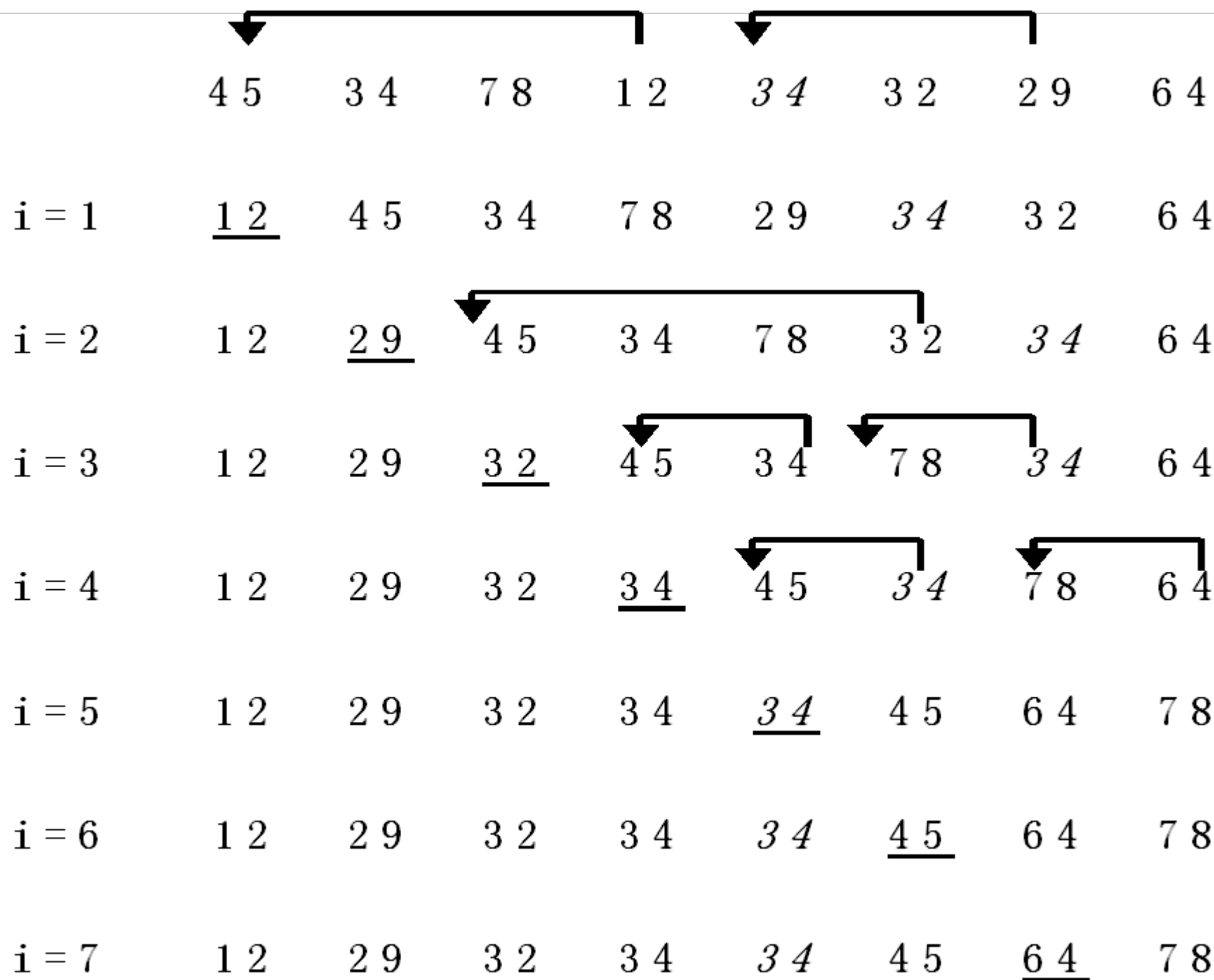

算法分析

- **稳定**
- 空间代价: $\Theta(1)$
- 时间代价:
 - 插入每个记录需要 $\Theta(\log i)$ 次比较
 - 最多移动 $i+1$ 次，最少2次（移动临时记录）
- 因此最佳情况下总时间代价为 $\Theta(n \log n)$ ，最差和平均情况下仍为 $\Theta(n^2)$



7.2.2 冒泡排序

- 算法思想：
 - 不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序。





冒泡排序类

```
template <class Record,class  
    Compare>  
class BubbleSorter:public  
    Sorter<Record,Compare>  
{ // 冒泡排序类  
public:  
    void Sort(Record Array[],int n);  
};
```



冒泡排序算法

```
template <class Record,class Compare>  
void BubbleSorter<Record,Compare>::  
Sort(Record Array[], int n)  
{ //冒泡排序，Array[]为待排数组，n为数组长度  
    //第i个记录冒泡  
    for (int i=1; i<n; i++)  
        //依次比较相邻记录，如果发现逆置，就交换  
        for (int j=n-1; j>=i; j--)  
            if (Compare::lt(Array[j],  
                Array[j-1]))        swap(Array, j, j-1);  
}
```



算法分析

- 稳定
- 空间代价: $\Theta(1)$
- 时间代价 : $\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = \Theta(n^2)$
 - 比较次数 : $\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = \Theta(n^2)$
 - 交换次数最多为 $\Theta(n^2)$, 最少为0, 平均为 $\Theta(n^2)$ 。
 - 最大, 最小, 平均时间代价均为 $\Theta(n^2)$ 。



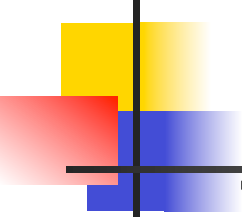
优化的冒泡排序

- 改进：检查每次冒泡过程中是否发生过交换，如果没有，则表明整个数组已经排好序了，排序结束。
- 避免不必要的比较

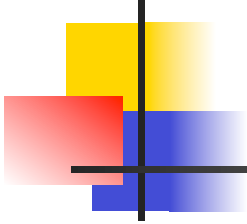


优化的冒泡排序

```
template <class Record,class  
    Compare>  
class ImprovedBubbleSorter:public  
    Sorter<Record,Compare>  
{ //优化的冒泡排序类  
public:  
    void Sort(Record Array[],int n);  
};
```

```
template <class Record,class Compare>  
void  
    ImprovedBubbleSorter<Record,Compare  
    >::  
Sort(Record Array[], int n)  
{ //Array[]为待排序数组，n为数组长度  
    bool NoSwap;        // 是否发生交换的标志  
    for (int i=1; i<n; i++)  
    {  
        NoSwap = true;    // 标志初始为真  
        for (int j=n-1; j>=i; j--)
```



```
if (Compare::lt(Array[j], Array[j-1]))
{
    //如果发生了交换,
    //标志变为假
    swap(Array, j, j-1);
    NoSwap = false;
}
// 如果没发生过交换,
// 表示已排好序, 结束算法
if (NoSwap)
    return;
}
```



算法分析

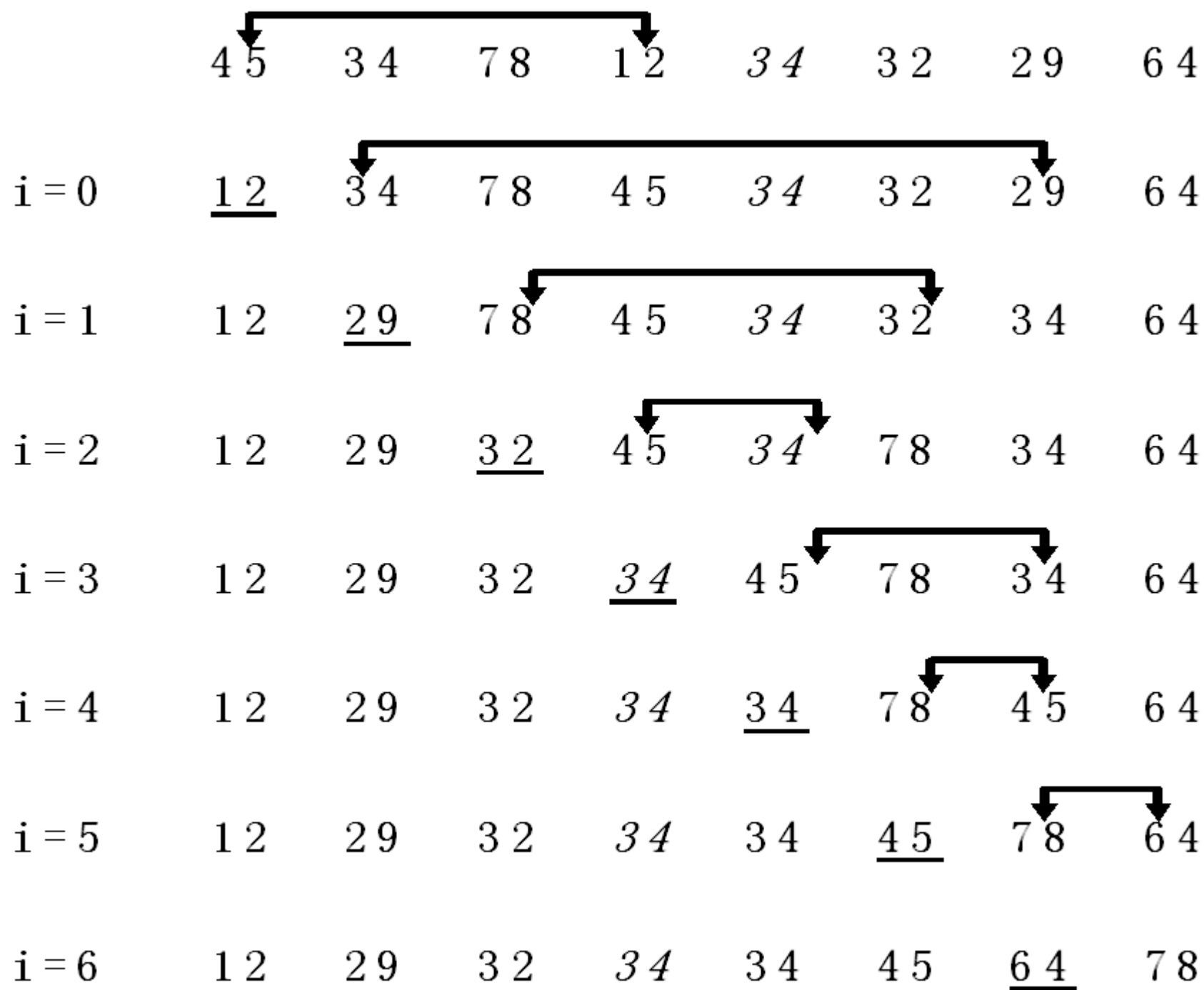
- 稳定
- 空间代价为 $\Theta(1)$
- 时间代价：
 - 最小时间代价为 $\Theta(n)$ ：最佳情况下只运行第一轮循环
 - 其他情况下时间代价仍为 $\Theta(n^2)$



7.2.3 直接选择排序

- 算法思想:

- 找出剩下的未排序记录中的最小记录，然后直接与数组中第 i 个记录交换，比冒泡排序减少了移动次数





直接选择排序

```
template <class Record,class  
    Compare>  
class StraightSelectSorter:public  
    Sorter<Record,Compare>  
{ // 直接选择排序类  
public:  
    void Sort(Record Array[],int n);  
};
```



```
template <class Record,class Compare>
```

```
void
```

```
StraightSelectSorter<Record,Compare
```

```
>::
```

```
Sort(Record Array[], int n)
```

```
{ //Array[]为待排序数组，n为数组长度
```

```
// 依次选出第i小的记录，
```

```
// 即剩余记录中最小的那个
```

```
for (int i=0; i<n-1; i++)
```

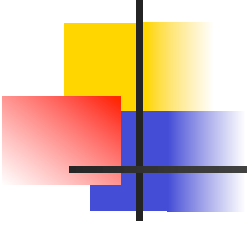
```
{
```

```
// 首先假设记录i就是最小的
```

```
int Smallest = i;
```

```
// 开始向后扫描所有剩余记录
```

```
for (int j=i+1;j<n; j++)
```



```
    // 如果发现更小的记录，记录它的位置
    if (Compare::lt(Array[j],
Array[Smallest]))
        Smallest = j;
    //将第i小的记录放在数组中第i个位置
    swap(Array, i, Smallest);
}
}
```

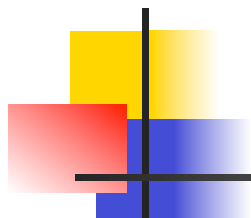



算法分析

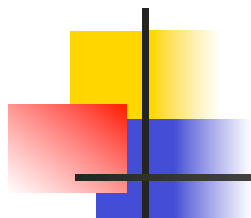
- 不稳定
- 空间代价: $\Theta(1)$
- 时间代价 :
 - 比较次数: $\Theta(n^2)$, 与冒泡排序一样
 - 交换次数: $n-1$
 - 总时间代价: $\Theta(n^2)$

7.2.4 简单排序算法的时间代价对比

比较次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$



移动次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	0	$\Theta(n)$	$\Theta(n)$	0	0	$\Theta(n)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$



总代价	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$



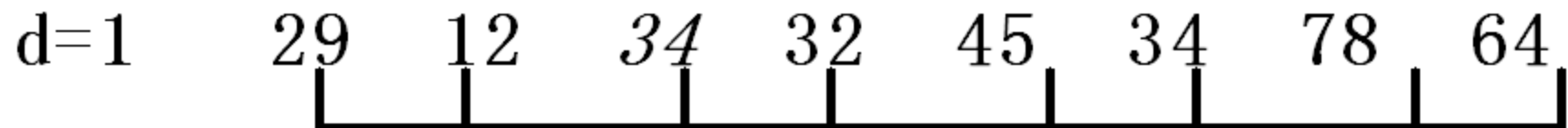
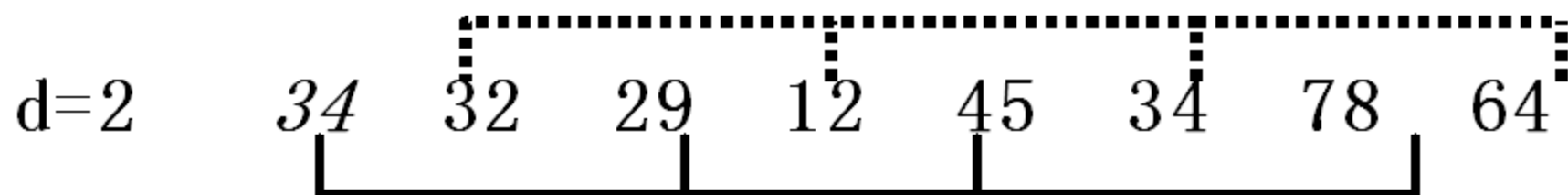
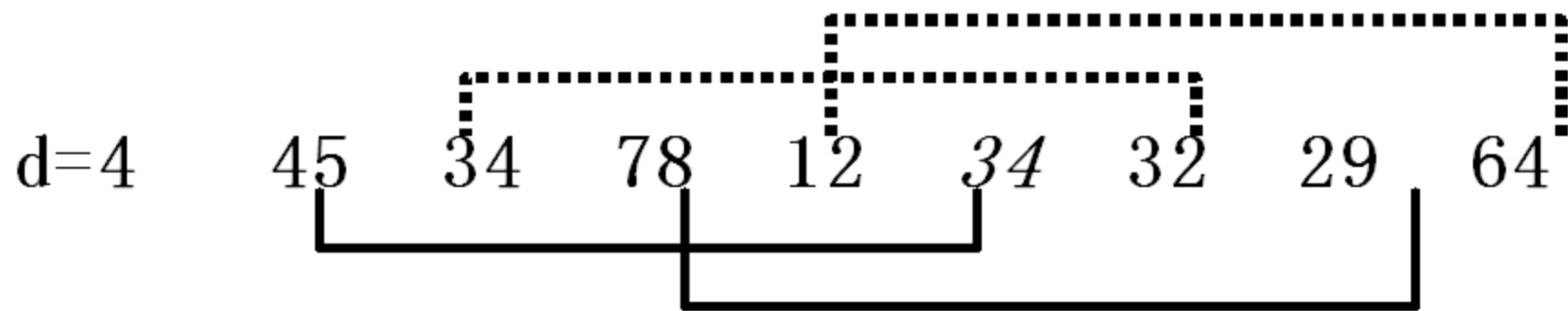
7.3 Shell排序

- 直接插入排序的两个性质：
 - 在最好情况（序列本身已是有序的）下时间代价为 $\Theta(n)$
 - 对于短序列，直接插入排序比较有效
- **Shell**排序有效地利用了直接插入排序的这两个性质。



Shell排序

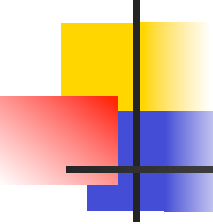
- 算法思想：
 - 先将序列转化为若干小序列，在这些小序列内进行插入排序；
 - 逐渐扩大小序列的规模，而减少小序列个数，使得待排序序列逐渐处于更有序的状态，
 - 最后对整个序列进行扫尾直接插入排序，从而完成排序。



12 29 32 34 34 45 64 78

“增量每次除以2递减”的 Shell排序

```
template <class Record,class Compare>
class ShellSorter:public
    Sorter<Record,Compare>
{ //Shell排序类
private:
    // 针对变化的增量而修改的插入排序算法，参数
    // delta表示当前的增量
    void ModifiedInsertSort(Record Array[],int
        n,int delta);
public:
    void Sort(Record Array[],int n);
};
```

```
template <class Record,class Compare>  
Void ShellSorter<Record,Compare>::Sort  
(Record Array[], int n)  
{ //Shell排序, Array[]为待排序数组,  
  // n为数组长度  
  // 增量delta每次除以2递减  
  for (int delta=n/2; delta>0; delta/=2)  
    //分别对delta个子序列排序  
    for (int j=0; j<delta; j++)  
      ModifiedInsertSort(&Array[j], n-  
j, delta);  
}
```

针对变化的增量而修改的插入排序算法

```
template <class Record,class Compare>
void ShellSorter<Record,Compare>::
ModifiedInsertSort(Record Array[],int n, int
    delta)
{ // 参数delta表示当前的增量
  // 对子序列中第i个记录排序
  for (int i=delta; i<n; i+=delta)
    for (int j=i; j>=delta; j-=delta){
      if (Compare::lt(Array[j], Array[j-
delta]))
        swap(Array, j, j-delta);
      else break;
    }
}
```



算法分析

- 不稳定
- 空间代价: $\Theta(1)$
- 时间代价: $\Theta(n^2)$
- 选择适当的增量序列, 可以使得时间代价接近 $\Theta(n)$ 。



增量序列的选择

- 增量每次除以**2**递减，时间代价为 $\Theta(n^2)$
- 增量序列为 $\{2^k-1, 2^{k-1}-1, \dots, 7, 3, 1\}$ 时，时间代价为 $\Theta(n^{3/2})$
- 选取其他增量序列还可以更进一步减少时间代价



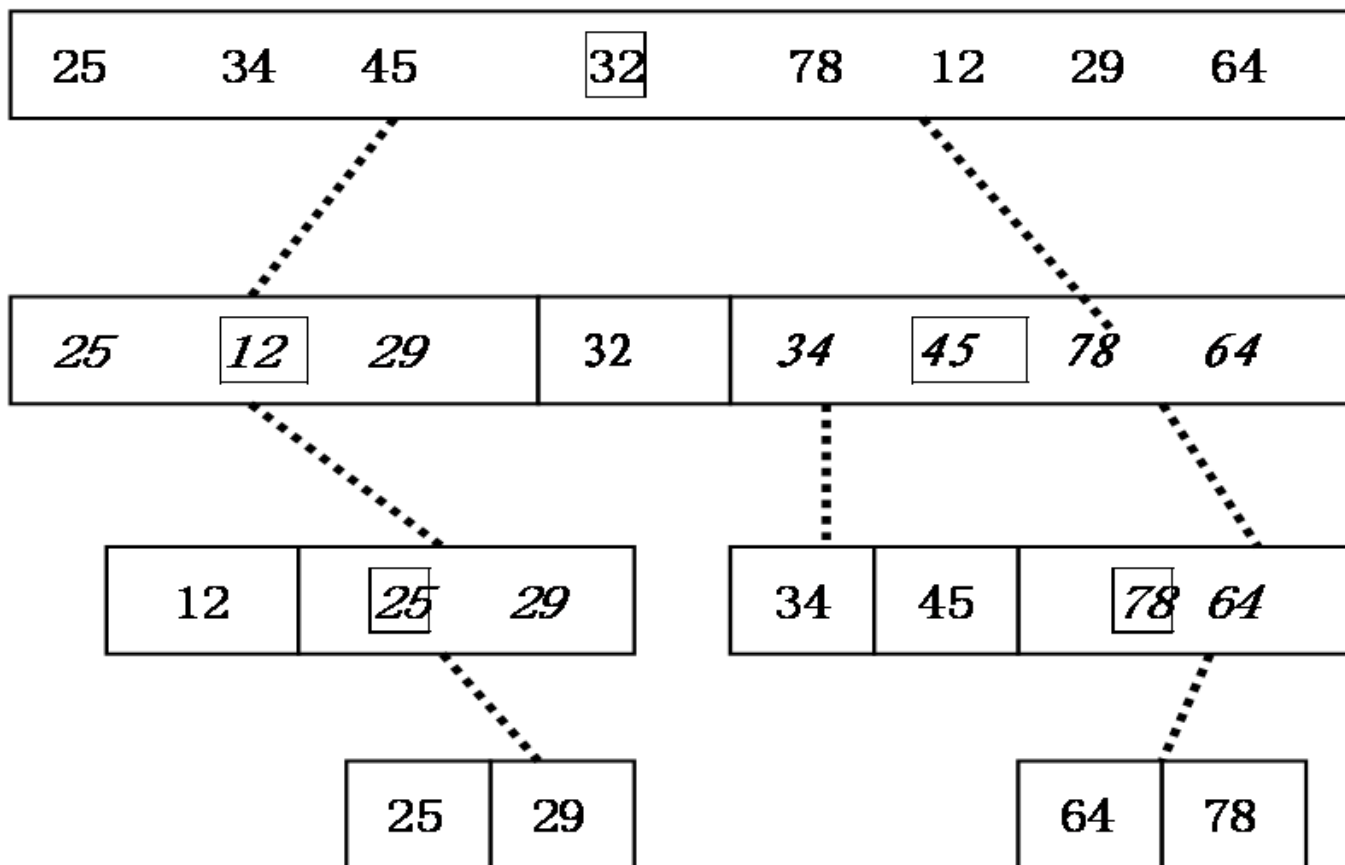
7.4 基于分治法的排序

- 将原始数组分为若干个子部分然后分别进行排序
- 两种算法
 - 快速排序
 - 归并排序



7.4.1 快速排序

- 算法思想：
 - 选择轴值（pivot）
 - 将序列划分为两个子序列**L**和**R**，使得**L**中所有记录都小于或等于轴值，**R**中记录都大于轴值
 - 对子序列**L**和**R**递归进行快速排序



最终排序结果:

12 25 29 32 34 45 64 78



轴值选择

- 尽可能使**L**，**R**长度相等
- 选择策略：
 - 选择最左边记录
 - 随机选择
 - 选择平均值



分割过程 (Partition)

- 整个快速排序的关键
- 分割后使得**L**中所有记录位于轴值左边，**R**中记录位于轴值右边，即轴值以位于正确位置

25 34 45 32 78 12 29 64

25	34	45	64	78	12	29	32
i							j

Diagram illustrating the comparison of elements at indices i and j in an array. The array contains the values 25, 34, 45, 64, 78, 12, 29, and 32. The element 34 is at index i and the element 32 is at index j . A curved arrow points from the element at j to the element at i , indicating a comparison or swap operation.


25 34 45 64 78 12 29 34

i j

Diagram illustrating a swap operation in an array. The array contains the values 25, 29, 45, 64, 78, 12, 29, 34. The element 29 at index i (the second position) is being swapped with the element 29 at index j (the seventh position). A curved arrow points from index j to index i , indicating the swap direction.

25 29 45 64 78 12 45 34

i j



25 29 12 64 78 12 45 34

i j

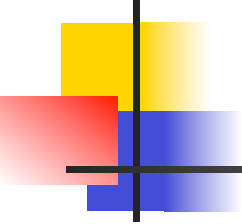
25	29	12	64	78	64	45	34
			i j				

25 29 12 32 78 64 45 34

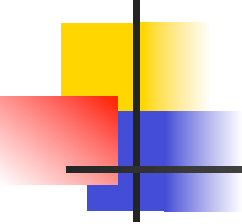


快速排序算法

```
template <class Record,class Compare>  
class QuickSorter:public  
    Sorter<Record,Compare>  
{ //快速排序类  
    private:  
        //选择轴值，返回轴值下标  
        int SelectPivot(int left, int right);  
        //分割,返回轴值位置  
        int Partition(Record Array[], int left, int right);  
    public:  
        void Sort(Record Array[],int left,int right);  
};
```



```
template <class Record,class Compare>  
void QuickSorter<Record,Compare>::  
Sort(Record Array[], int left,int right)  
{ //Array[]为待排序数组，i,j分别为数组两端  
// 如果子序列中只有0或1个记录，就不需排序  
if (right <= left)  
    return;  
//选择轴值  
int pivot = SelectPivot(left, right);  
//分割前先将轴值放到数组末端  
swap(Array, pivot, right);
```



```
// 对剩余的记录进行分割
pivot = Partition(Array, left, right);
//对轴值左边的子序列进行递归快速排序
Sort(Array, left, pivot-1);
//对轴值右边的子序列进行递归快速排序
Sort(Array, pivot + 1, right);
}
```



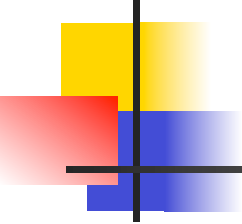
轴值选择函数

```
template <class Record,class Compare>  
int QuickSorter<Record,Compare>::  
SelectPivot(int left,int right)  
{ //参数left,right分别表示序列的左右端下标  
  //选择中间纪录作为轴值  
  return (left+right)/2;  
}
```

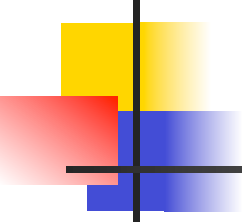


分割函数

```
template <class Record,class Compare>  
int QuickSorter<Record,Compare>::  
Partition(Record Array[], int left,int right)  
{ //分割函数，分割后轴值已到达正确位置  
  Record TempRecord; //存放轴值的临时变量  
  int i = left;           // i为左指针，j为右指针  
  int j = right;  
  //将轴值存放在临时变量中  
  TempRecord = Array[j];  
  //开始分割，i,j不断向中间移动，直到相遇
```

```
while (i != j)
{
    //i指针右移，直到找到一个大于等于轴值的记录
    while (Compare::lt(Array[i], TempRecord)
&& (j>i))
        i++;
    // 如果i,j未相遇就将逆序元素换到右边空闲位置
    if (i<j)
    {
        Array[j] = Array[i];
        j--;           //j指针向左移动一步
    }
}
```



```
//j指针左移，直到找到一个小于等于轴值的记录
while (Compare::gt(Array[j], TempRecord)
&& (j > i))
    j--;
//如果i,j未相遇就将逆序元素换到左边空闲位置
if (i < j)
{
    Array[i] = Array[j];
    i++;          //i指针向右移动一步
}
}
//把轴值回填到分界位置i上
Array[i] = TempRecord;
return i;        //返回分界位置i
}
```



算法分析


- 最差情况：
 - 时间代价: $\Theta(n^2)$
 - 空间代价: $\Theta(n)$
- 最佳情况：
 - 时间代价: $\Theta(n \log n)$
 - 空间代价: $\Theta(\log n)$
- 平均情况：
 - 时间代价: $\Theta(n \log n)$
 - 空间代价: $\Theta(\log n)$



算法分析（续）

- 不稳定
- 优化：当快速排序的子数组小于某个长度时，不继续递归，直接进行插入排序。
- 经验表明，最好的组合方式是当 n （子数组的长度）小于等于**16**时就使用插入排序

优化的快速排序



```
#define THRESHOLD 16  
template <class Record,class Compare>  
class ImprovedQuickSorter:public  
    Sorter<Record,Compare>  
{ //优化的快速排序类  
private:  
    //选择轴值，返回轴值下标  
    int SelectPivot(int left, int right);  
    //分割,返回轴值位置  
    int Partition(Record Array[], int left, int right);  
    void DoSort(Record Array[],int left,int right);  
public:  
    void Sort(Record Array[],int left,int right);  
};
```



```
template <class Record,class Compare>  
void  
    ImprovedQuickSorter<Record,Compare>::Sort  
    (Record Array[], int left,int right)  
{ //优化的快速排序, Array[]为待排序数组, i,j分别  
    //为数组两端  
    //先对序列进行递归排序  
    DoSort(Array,left,right);  
    //最后对整个序列进行一次直接插入排序  
    ImprovedInsertSorter<Record,Compare>  
    insert_sorter;  
    insert_sorter.Sort(Array,right-left+1);  
}
```



```
template <class Record,class Compare>
```

```
void
```

```
    ImprovedQuickSorter<Record,Compare>::DoSort(R  
    ecord Array[], int left,int right)
```

```
{ //优化的快速排序，Array[]为待排序数组，i,j分别为数组  
  //两端
```

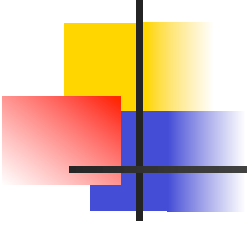
```
  // 如果序列中只有0或1个记录，就不用排序
```

```
  if (right <= left)      return;
```

```
  //如果序列长度小于等于阈值(16最佳),停止分割跳出递归
```

```
  if (right-left+1 > THRESHOLD)
```

```
{
```



```
//选择轴值
int pivot = SelectPivot(left, right);
// 将轴值放在数组末端
swap(Array, pivot, right);
// 对剩余的记录进行分割
pivot = Partition(Array, left, right);
//对分割出的子序列进行递归快速排序
//对轴值左右分别进行排序
DoSort(Array, left, pivot-1);
DoSort(Array, pivot+1, right);
```

```
}
```

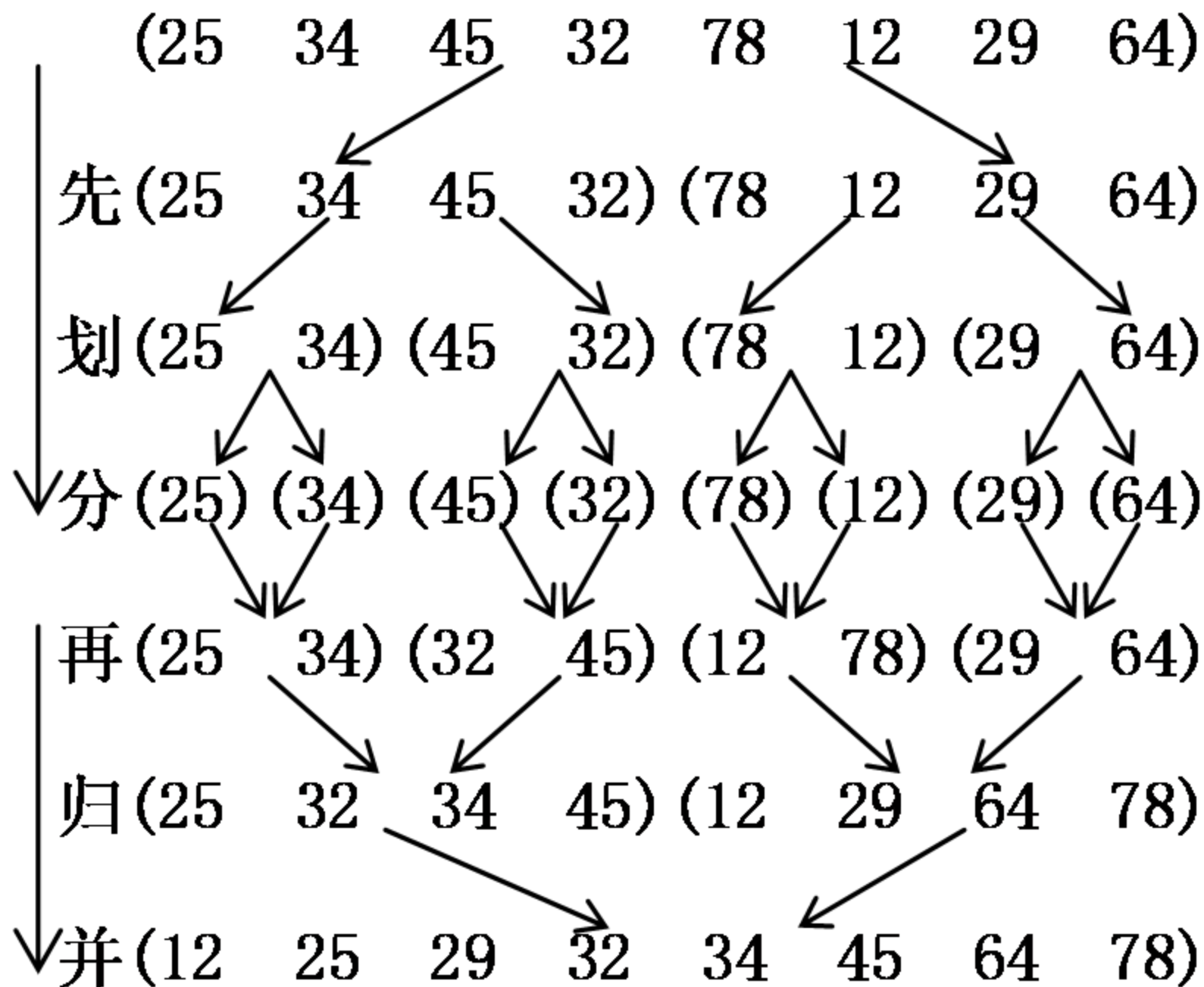
```
}
```




7.4.2 归并排序

■ 算法思想


- 简单地将原始序列划分为两个子序列
- 分别对每个子序列递归排序
- 最后将排好序的子序列合并为一个有序序列，即归并过程



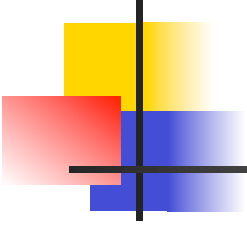


两路归并排序

```
template <class Record,class Compare>  
class TwoWayMergeSorter:public  
    Sorter<Record,Compare>  
{ //两路归并排序类  
private:  
    //归并过程  
    void Merge(Record Array[], Record  
    TempArray[],int left,int right,int middle);  
public:  
    void Sort(Record Array[], Record  
    TempArray[],int left, int right);  
};
```



```
template <class Record,class Compare>  
void TwoWayMergeSorter<Record,Compare>::  
Sort(Record Array[], Record TempArray[],int left,  
int right)  
{ //Array为待排序数组, left, right分别为数组两端  
// 如果序列中只有0或1个记录, 就不用排序  
if (left < right)  
{  
    //从中间划分为两个子序列  
int middle=(left+right)/2;
```



```
//对左边一半进行递归
Sort(Array, TempArray,left,middle);
//对右边一半进行递归
Sort(Array, TempArray,middle+1,right);
// 进行归并
Merge(Array, TempArray,left,right,middle);

}
}
```



归并函数

```
template <class Record,class Compare>  
void  
    TwoWayMergeSorter<Record,Compare  
    >::  
Merge(Record Array[], Record  
    TempArray[],int left,int right,int middle)  
{ //归并过程  
    // 将数组暂存入临时数组  
    for (int j=left; j<=right; j++)  
  
        TempArray[j] = Array[j];
```



```
int index1=left;    //左边子序列的起始位置  
int index2=middle+1; //右子序列起始位置  
int i=left;          //从左开始归并  
while ((index1 <= middle)&&(index2 <=  
right))  
{  
    //取较小者插入合并数组中  
    if (Compare::le(TempArray[index1],  
TempArray[index2]))  
        Array[i++] = TempArray[index1++];
```



else

Array[i++] = TempArray[index2++];

}

//处理剩余记录

while (index1 <= middle)

Array[i++] = TempArray[index1++];

while (index2 <= right)

Array[i++] = TempArray[index2++];

}



算法分析

- 空间代价： $\Theta(n)$
- 总时间代价： $\Theta(n \log n)$
- 不依赖于原始数组的输入情况，最大、最小以及平均时间代价均为 $\Theta(n \log n)$ 。



算法分析（续）

- 稳定
- 优化：
 - 同优化的快速排序一样，对基本已排序序列直接插入排序
 - **R.Sedgewick**优化：归并时从两端开始处理，向中间推进



优化的归并排序

```
#define THRESHOLD 16
template <class Record,class Compare>
class ImprovedTwoWayMergeSorter:public
    Sorter<Record,Compare>
{ //优化的两路归并排序类
private:
    void Merge(Record Array[], Record
        TempArray[],int left,int right,int middle); //归并过程
public:
    void Sort(Record Array[], Record TempArray[], int
        left, int right);
};
```



// 优化的两路归并排序, **Array[]**为待排序数组, **left**,
right分别为数组两端

template <class Record,class Compare>

void

**ImprovedTwoWayMergeSorter<Record,Compare>::Sort(Record Array[],Record TempArray[],
int left, int right)**

{

DoSort(Array,TempArray,left,right);

//最后对整个序列进行一次直接插入排序

**ImprovedInsertSorter<Record,Compare>
insert_sorter;**

insert_sorter.Sort(&Array[left],right-left+1);

}

```
template <class Record,class Compare>
```

```
void
```

```
ImprovedTwoWayMergeSorter<Record,Compare>::  
DoSort(Record Array[],Record TempArray[], int left,  
int right) {
```

```
//如果序列长度大于阈值(16最佳), 跳出递归
```

```
if (right <= left)      return;
```

```
    if (right-left+1 > THRESHOLD) {
```

```
        int middle=(left+right)/2;    //从中间划分
```

```
        Sort(Array, TempArray ,left,middle);
```

```
        Sort(Array, TempArray ,middle+1,right);
```

```
        Merge(Array, TempArray ,left,right,middle);
```

```
    }
```

```
    else {
```

```
        ImprovedInsertSorter<Record,Compare>
```

```
insert_sorter;
```

```
        insert_sorter.Sort(&Array[left],right-left+1);
```

```
    }
```

```
}
```



```
template <class Record,class Compare>
```

```
void
```

```
ImprovedTwoWayMergeSorter<Record,Compare>::Merge(Record Array[],Record  
TempArray[],int left,int right,int middle)
```

```
{ //归并过程
```

```
int index1,index2;    //两个子序列的起始位置
```

```
int i,j,k ;
```

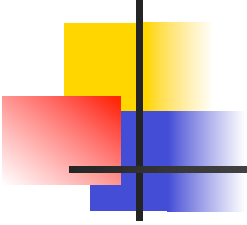
```
for (i=left; i<=middle; i++)//复制左边的子序列
```

```
TempArray[i] = Array[i];
```

```
//复制右边的子序列，但顺序颠倒过来
```

```
for (j=1; j<=right-middle; j++)
```

```
TempArray[right-j+1] = Array[j+middle];
```

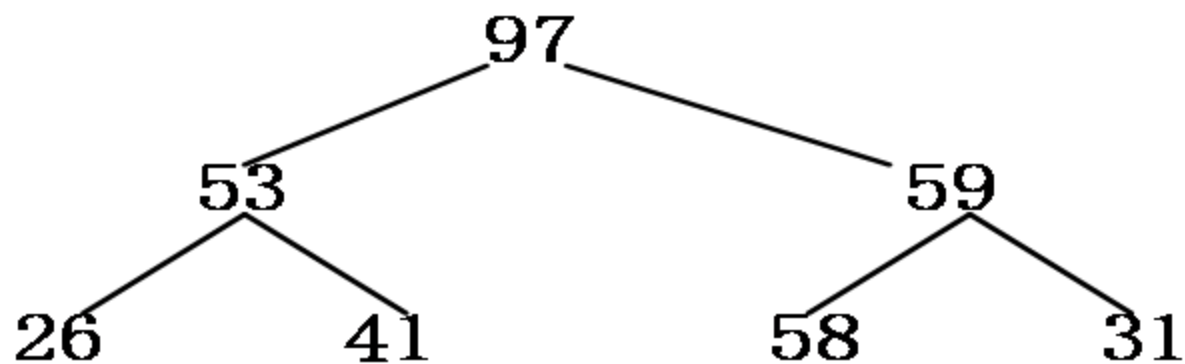


```
// 开始归并
for (index1=left,index2=right,k=left;
    k<=right; k++)
    if (Compare::lt(TempArray[index1],
        TempArray[index2]))
        Array[k] = TempArray[index1++];
    else
        Array[k] = TempArray[index2--];
}
```



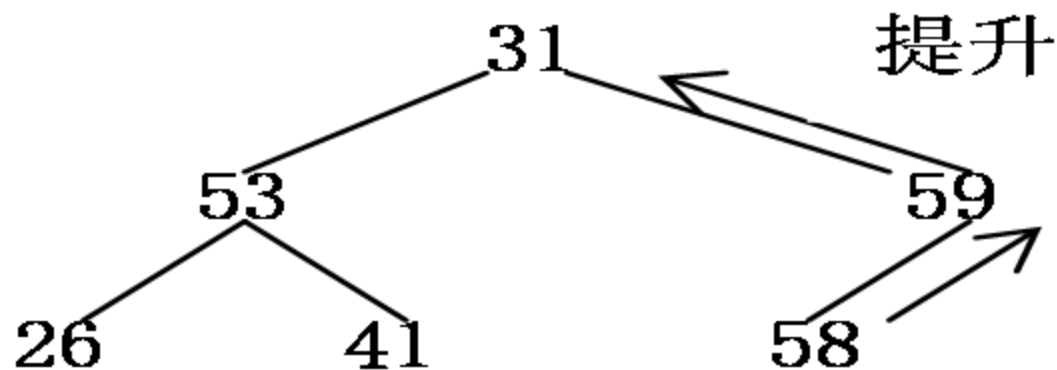
7.5 堆排序

- 直接选择排序：直接从剩余记录中线性查找最大记录
- 堆排序：基于最大值堆来实现，效率更高

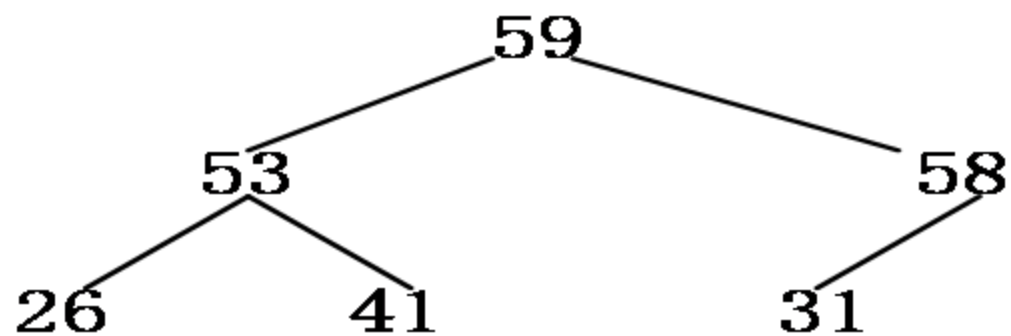


(a)

97 53 59 26 41 58 31

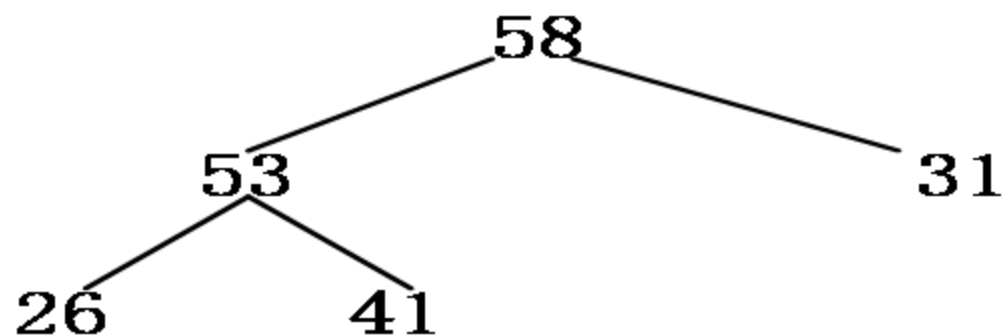


31 53 59 26 41 58 | 97



(b)

59 53 58 26 41 31 | 97



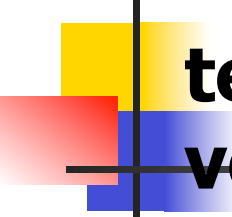
(c)

58 53 31 26 41 | 59 97



堆排序算法

```
template <class Record,class  
    Compare>  
class HeapSorter:public  
    Sorter<Record,Compare>  
{ //堆排序类  
public:  
    void Sort(Record Array[],int n);  
};
```



```
template <class Record,class Compare>
void HeapSorter<Record,Compare>::
Sort(Record Array[], int n)
{ //堆排序，Array[]为待排数组，n为数组长度
  Record record;
  MaxHeap< Record > max_heap =
  MaxHeap< Record >(Array,n,n); //建堆
  // 依次找出剩余记录中的最大记录，即堆顶
  for (int i=0; i<n; i++)
    max_heap.Removemax(record);
}
```



算法分析

- 建堆: $\Theta(n)$
- 删除堆顶: $\Theta(\log n)$
- 一次建堆, n 次删除堆顶
- 总时间代价为 $\Theta(n \log n)$
- 空间代价为 $\Theta(1)$



7.6 分配排序和基数排序

- 不需要进行比较
- 需要事先知道记录序列的一些具体情况



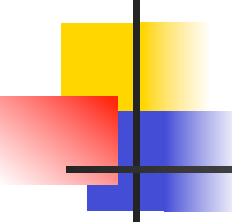
7.6.1 桶式排序

- 事先知道序列中的记录都位于某个小区间段[0, **m**)内
- 将具有相同值的记录都分配到同一个桶中，然后依次按照编号从桶中取出记录，组成一个有序序列

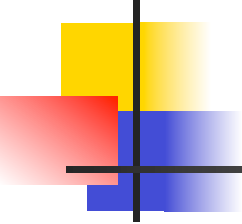


桶式排序算法

```
template <class Record>  
class BucketSorter:public  
    Sorter<Record,Compare>  
{ //桶式排序类  
public:  
    void Sort(Record Array[],int n,int  
        max);  
};
```

```
//桶式排序，Array[]为待排序数组，数组长度为n,所  
//有记录都位于区间[0,max)上  
template <class Record>  
void BucketSorter<Record>::Sort(Record  
    Array[], int n,int max)  
{  
    int* TempArray=new Record[n]; //临时数组  
    int* count=new int[max];//小于等于i的元素个数  
    int i;  
    for (i=0;i<n;i++)  
        TempArray[i]=Array[i];  
    //所有计数器初始都为0  
    for (i=0;i<max;i++)  
        count[i]=0;
```



```
//统计每个取值出现的次数
for (i=0;i<n;i++)
    count[Array[i]]++;
//统计小于等于i的元素个数
for (i=1;i<max;i++)
    count[i]=count[i-1]+count [i];
//按顺序输出有序序列
for (i=n-1;i>=0;i--)
    Array[--count[TempArray[i]]] =
    TempArray[i];
}
```



算法分析

- 时间代价：
 - 统计计数时： $\Theta(n+m)$
 - 输出有序序列时循环 n 次
 - 总的时间代价为 $\Theta(m+n)$
 - 适用于 m 相对于 n 很小的情况



算法分析（续）

- 空间代价：
 - **m**个计数器，长度为**n**的临时数组， $\Theta(m+n)$
- 稳定



7.6.2 基数排序

- 桶式排序只适合**m**很小的情况
- 当**m**很大时，可以将一个记录的值即排序码拆分为多个部分来进行比较——基数排序



基数排序

- 假设长度为**n**的序列

$$\mathbf{R} = \{ \mathbf{r}_0, \mathbf{r}_1, \dots, \mathbf{r}_{n-1} \}$$

记录的排序码**K**包含**d**个子排序码

$$\mathbf{K} = (\mathbf{k}_{d-1}, \mathbf{k}_{d-2}, \dots, \mathbf{k}_1, \mathbf{k}_0),$$

则称**R**对排序码 $(\mathbf{k}_{d-1}, \mathbf{k}_{d-2}, \dots, \mathbf{k}_1, \mathbf{k}_0)$ 有序就是
对于任意两个记录**R_i**, **R_j** ($0 \leq i < j \leq n-1$),
都满足

$$(\mathbf{k}_{i,d-1}, \mathbf{k}_{i,d-2}, \dots, \mathbf{k}_{i,1}, \mathbf{k}_{i,0}) \leq (\mathbf{k}_{j,d-1}, \mathbf{k}_{j,d-2}, \dots, \mathbf{k}_{j,1}, \mathbf{k}_{j,0})$$

其中**k_{d-1}**称为最高排序码, **k₀**称为最低排序码。



例子

- 例如：如果我们要对**0到9999**之间的整数进行排序
- 将四位数看作是由四个排序码决定，即千、百、十、个位，其中千位为最高排序码，个位为最低排序码。基数 **$r=10$** 。
- 可以按千、百、十、个位数字依次进行**4**次桶式排序
- **4**趟分配排序后，整个序列就排好序了。



基数排序分为两类

- 高位优先法（**MSD, Most Significant Digit first**）
- 低位优先法（**LSD, Least Significant Digit first**）



高位优先法 (MSD, Most Significant Digit first)

- 先对高位 k_{d-1} 进行桶式排序，将序列分成若干个桶中
- 然后对每个桶再按次高位 k_{d-2} 进行桶式排序，分成更小的桶；
- 依次重复，直到对 k_0 排序后，分成最小的桶，每个桶内含有相同的排序码($k_{d-1}, k_{d-2}, \dots, k_1, k_0$)；
- 最后将所有的桶依次连接在一起，成为一个有序序列。
- 这是一个分、分、...、分、收的过程。



低位优先法 (LSD, Least Significant Digit first)

- 从最低位 k_0 开始排序;
- 对于排好的序列再用次低位 k_1 排序;
- 依次重复, 直至对最高位 k_{d-1} 排好序后, 整个序列成为有序的。
- 这是一个分、收; 分、收; ...; 分、收的过程。
- 比较简单, 计算机常用

97 53 88 59 26 41 58 31 22

0 1 2 3 4 5 6 7 8 9

		26 22	31	41	53 59 58			88	
--	--	-------	----	----	----------	--	--	----	--

0 1 2 3 4 5 6 7 8 9

		22				26			
--	--	----	--	--	--	----	--	--	--

22 26 31 41 53 58 59 88 97

(a) 高位优先

97 53 88 59 26 41 58 31 22

0 1 2 3 4 5 6 7 8 9

	41 31	22	53			26	97	88 58	59
--	-------	----	----	--	--	----	----	-------	----

41 31 22 53 26 97 88 58 59

0 1 2 3 4 5 6 7 8 9

		22 26	31	41	53, 58, 59			88	97
--	--	--------------	----	----	------------	--	--	----	----

22 26 31 41 53 58 59 88 97

(b) 低位优先



基数排序的实现

- 基于顺序存储
- 基于链式存储



基于顺序存储的基数排序

- 只讨论**LSD**
- 原始输入数组**R**的长度为**n**
- 基数为**r**
- 排序码个数为**d**

初始数组内容: 97 53 88 59 26 41 58 31 22

 0 1 2 3 4 5 6 7 8 9

第一趟: count

0	2	1	1	0	0	1	1	2	1
---	---	---	---	---	---	---	---	---	---

 0 1 2 3 4 5 6 7 8 9

按 count 分配桶:

0	2	3	4	4	4	5	6	8	9
---	---	---	---	---	---	---	---	---	---

收集:

41	31	22	53	26	97	88	58	59
----	----	----	----	----	----	----	----	----

0 1 2 3 4 5 6 7 8 9

第二趟: count

0	0	2	1	1	3	0	0	1	1
---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9

按count 分配桶:

0	0	2	3	4	7	7	7	8	9
---	---	---	---	---	---	---	---	---	---

收集:

22	26	31	41	53	58	59	88	97
----	----	----	----	----	----	----	----	----

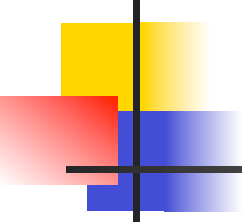
最终排序结果:

22 26 31 41 53 58 59 88 97

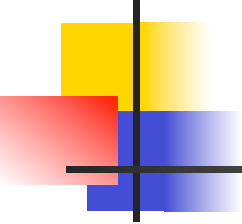


基于数组的基数排序

```
template <class Record>  
class RadixSorter:public  
    Sorter<Record,Compare>  
{ // 基数排序类  
public:  
    void Sort(Record Array[],int n,int  
        min,int max);  
};
```



```
template <class Record>  
void RadixSorter<Record>::Sort(Record Array[],  
    int n,int d, int r)  
{ //n为数组长度，d为排序码数，r为基数  
    //临时数组  
    Record *TempArray =new Record[n];  
    int* count= new int[r];           //计数器  
    int i,j,k;  
    int Radix=1;           //取Array[j]的第i位排序码  
    for (i=1; i<=d; i++) // 分别对第i个排序码分配  
    {
```



```
for (j=0; j<r; j++)          // 初始计数器均为0
    count[j] = 0;
for (j=0; j<n; j++)// 统计每个桶中的记录数
{
    //取Array[j]的第i位排序码
    k=(Array[j] /Radix)%r;
    count[k]++;          //相应计数器加1
}
// 将TempArray中的位置依次分配给r个桶
for (j=1; j<r; j++)
    count[j] = count[j-1] + count[j];
```



```
// 将所有桶中的记录依次收集到TempArray中
for (j=n-1; j>=0; j--)
```

```
{
```

```
    //取Array[j]的第i位排序码
```

```
    k=(Array[j] / Radix)%r;
```

```
    //从第k个桶取出一个记录，计数器减1
```

```
    count[k]--;
```

```
    TempArray[count[k]] = Array[j];
```

```
}
```

```
// 将临时数组中的内容复制到Array中
```

```
for (j=0; j<n; j++)
```

```
    Array[j] = TempArray[j];
```

```
Radix*=r;
```

```
}
```

```
}
```



算法分析

- 空间代价：
 - 临时数组, n
 - r 个计数器
 - $\Theta(n+r)$



算法分析（续）

- 时间代价
 - 桶式排序: $\Theta(m+n)$
 - d 次桶式排序
 - $\Theta(d \cdot (n+r))$

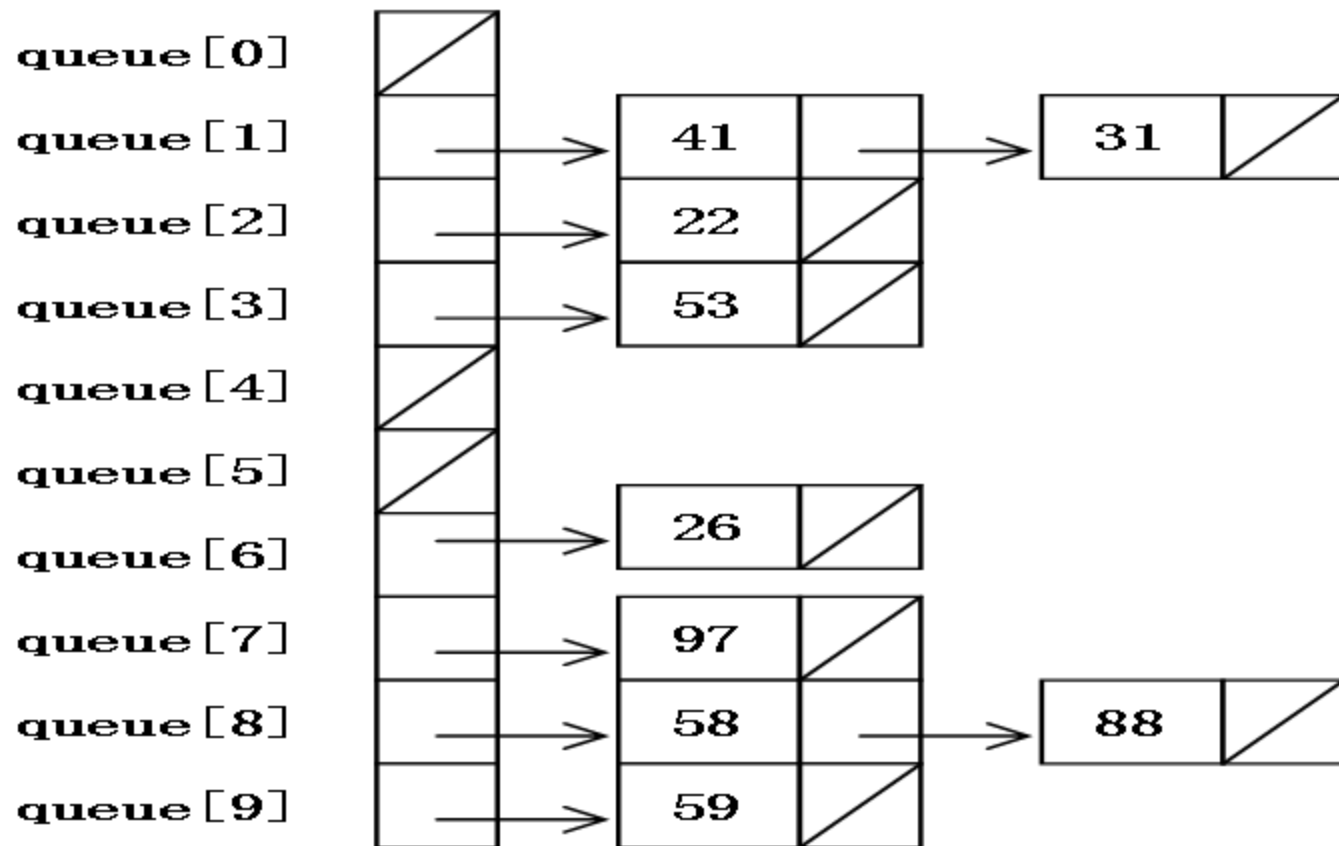


基于静态链的基数排序

- 将分配出来的子序列存放在 r 个(静态链组织的)队列中
- 链式存储避免了空间浪费情况

97	53	88	59	26	41	58	31	22
----	----	----	----	----	----	----	----	----

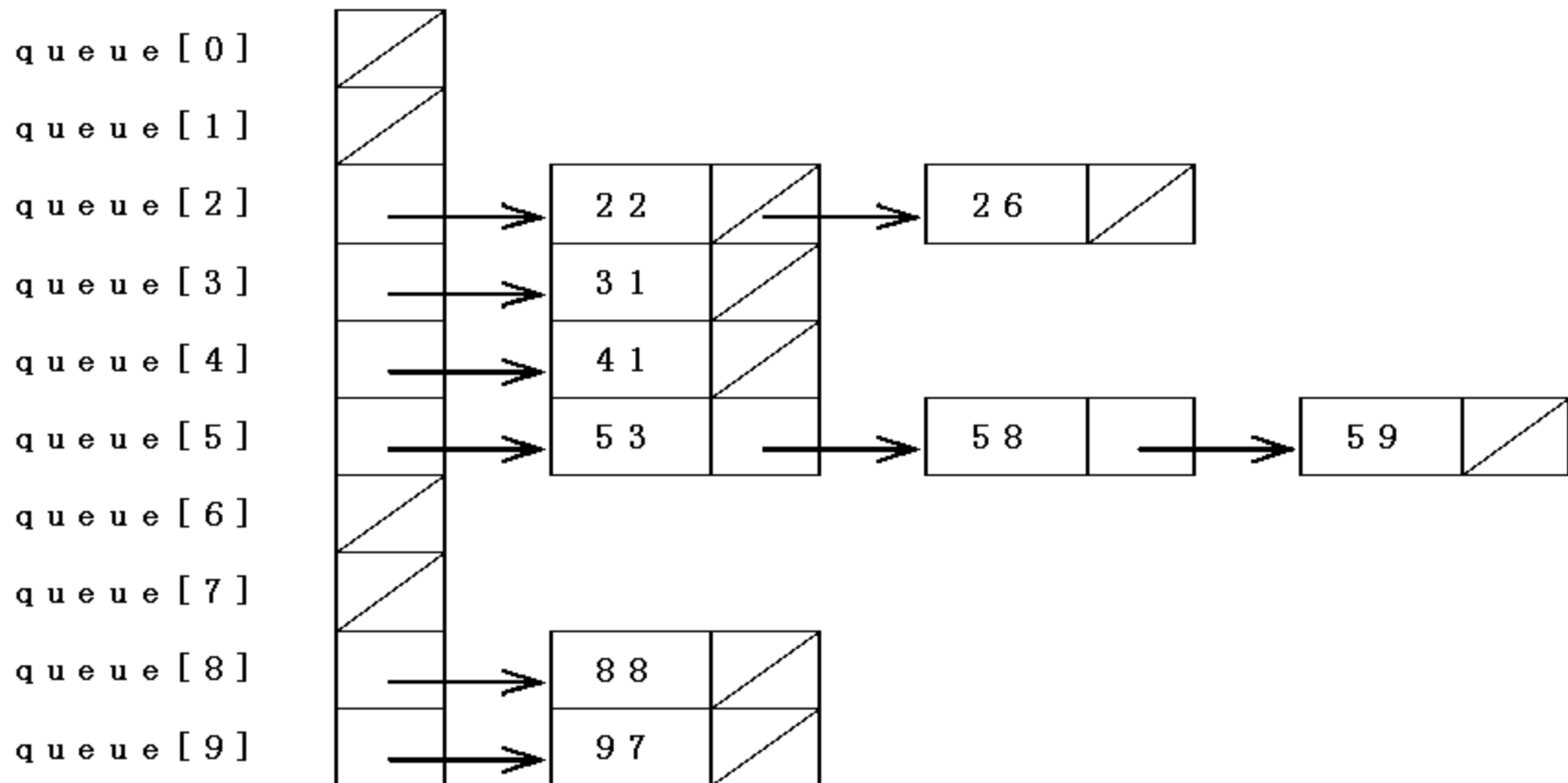
(a) 初始链表内容



(b) 第一趟分配结果

41	31	22	53	26	97	58	88	59
----	----	----	----	----	----	----	----	----

(c) 第一趟收集结果



(d) 第二趟分配结果

2 2	2 6	3 1	4 1	5 3	5 8	5 9	8 8	9 7
-----	-----	-----	-----	-----	-----	-----	-----	-----

(e) 第二趟收集结果

最终排序结果：

2 2 2 6 3 1 4 1 5 3 5 8 5 9 8 8 9 7



静态队列定义

```
// 结点类
class Node{
public:
    int key;    // 结点的关键码值
    int next;   // 下一个结点在数组中的下标
};
```

```
// 静态队列类
class StaticQueue{
public:
    int head;
    int tail;
};
```



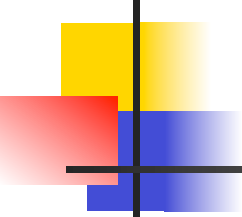
基于静态链的基数排序

```
template <class Record>  
class LinkRadixSorter  
{ //基于静态链的基数排序类  
private:  
    void Distribute(Record* Array,int first,int i,int  
        r, StaticQueue* queue);    //分配过程  
    void Collect(Record* Array, int& first, int i, int  
        r, StaticQueue* queue);    //收集过程  
    void PrintArray(Record* A, int first);//输出序列  
public:  
    void Sort(Record* Array,int n, int d, int r);  
};
```



静态链实现的基数排序

```
template <class Record>  
void LinkRadixSorter<Record>::Sort(Record*  
    Array, int n, int d, int r)  
{ //n为数组长度, d为排序码个数, r为基数  
  
    int first=0;    // first指向静态链中第一个记录  
    StaticQueue *queue;  
    //存放r个桶的静态队列  
    queue = new StaticQueue[r];  
    // 建链, 初始为next域指向下一个记录  
    for (int i=0; i<n; i++)  
        Array[i].next = i+1;
```



```
Array[n-1].next = -1; //链尾next为空  
cout<<"排序前: " <<endl;  
PrintArray(Array, 0); //输出原始序列  
//对第j个排序码进行分配和收集, 一共d趟  
for (int j=0; j<d; j++)  
{  
    Distribute(Array, first, j, r, queue);  
    Collect(Array, first, j, r, queue);  
}  
cout<<"排序后: " <<endl;  
PrintArray(Array, first); //输出排序后的结果  
delete[] queue;
```



分配过程

```
template <class Record>
void LinkRadixSorter<Record>::Distribute(
Record* Array,int first,int i,int r, StaticQueue*
queue)
{ //A中存放待排序记录, first为静态链中的第一个记
//录,i为第i个排序码, r为基数
//初始化r个队列
for (int j=0; j<r; j++)
    queue[j].head=-1;
//对整个静态链进行分配
while (first != -1)
```



{

//取第*i*位排序码数字

int k=Array[first].key;

for (int a=0;a<i;a++)

k= k/r;

k=k%r;

// 把Array[first]分配到第k个子序列中,

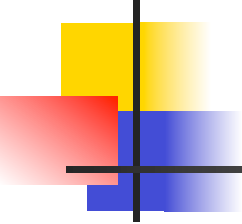
// 如果子序列为空,

// Array[first]就是第一个记录

if (queue[k].head == -1)

queue[k].head = first;

else // 否则加到子序列的尾部



```
first;      Array[queue[k].tail].next =  
             //first为子序列的尾部  
             queue[k].tail = first;  
             //继续分配下一个记录  
             first = Array[first].next;  
             }
```




收集过程

```
template <class Record>
```

```
void LinkRadixSorter<Record>::Collect  
(Record* Array, int& first, int i, int r,  
StaticQueue* queue)
```

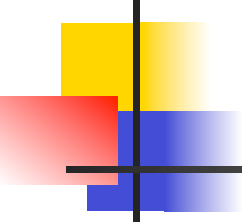
```
{ //A中存放待排序记录，first为静态链中的第一  
个记
```

```
//录，i为第i个排序码，r为基数
```

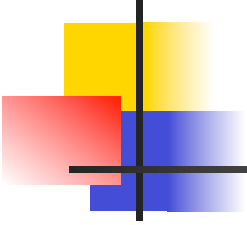
```
int last, k=0; //已收集到的最后一个记录
```

```
// 找到第一个非空队列
```

```
while (queue[k].head == -1) k++;
```



```
first = queue[k].head;  
last = queue[k].tail;  
while (k<r-1)           //继续收集下一个非空队列  
{  
    k++;  
    //找到下一个非空队列  
    while (k<r-1 && queue[k].head==-1) k++;  
    //将这个非空序列与上一个非空序列连接起来  
    if (queue[k].head!= -1)  
    {
```



```
    Array[last].next = queue[k].head;  
    //此时最后一个记录为该序列的尾部记录  
    last = queue[k].tail;  
    }  
  }  
  Array[last].next = -1;           //收集完毕  
}
```



输出序列中所有内容

```
template <class Record>
void LinkRadixSorter<Record>::
PrintArray(Record* Array, int first)
{ //first为静态链Array中第一个记录的下标
  int tmp;           //用来扫描整个链的指针
  tmp = first;
  while (tmp != -1)
  {
    cout << Array[tmp].key << " "; //输出记录
    tmp = Array[tmp].next;         }
  cout << '\n';
}
```



算法分析

- 空间代价
 - n 个记录空间
 - r 个子序列的头尾指针
 - $O(n + r)$
- 时间代价
 - 不需要移动记录本身，只需要修改记录的**next**指针
 - $O(d \cdot (n + r))$

7.7 各种排序算法的理论和实验时间代价

算法	最大时间	平均时间	最小时间	辅助空间代价	稳定性
直接插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
二分法插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$	稳定
冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	稳定
改进的冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
选择排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	不稳定

算法	最大时间	平均时间	最小时间	辅助空间代价	稳定性
Shell排序 (3)	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(1)$	不稳定
快速排序	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(\log n)$	不稳定
归并排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	稳定
堆排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	不稳定
桶式排序	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	稳定
基数排序	$\Theta(d \cdot (n + r))$	$\Theta(d \cdot (n + r))$	$\Theta(d \cdot (n + r))$	$\Theta(n+r)$	稳定



小结

- n 很小或基本有序时插入排序比较有效
- 综合性能快速排序最佳



测试环境

- 硬件环境:
 - **CPU: Intel P4 2.4G**
 - **内存: 512M DDR**
- 软件环境:
 - **windows xp**
 - **Visual C++ 6.0**



随机生成待排序数组

```
// 设置随机种子
inline void Randomize() { srand(1); }
// 返回一个0到n之间的随机整数值
inline int Random(int n)
    { return rand() % (n); }
// 产生随机数组
ELEM *sortarray = new ELEM[1000000];
for(int i=0;i<1000000;i++)
sortarray[i]=Random(32003);
```

#include <time.h> 时间测试
/* Clock ticks macro - ANSI version */
#define CLOCKS_PER_SEC 1000

clock_t tstart = 0; // Time at beginning
// Initialize the program timer
void Settime() { tstart = clock(); }
// The elapsed time since last Settime()
double Gettime()
{ return (double)((double)clock() -
(double)tstart)/
(double)CLOCKS_PER_SEC; }



排序的时间测试

```
Settime();  
    for (i=0; i<ARRAYSIZE; i+=listsize) {  
        sort<int,intintCompare>(&array[i],  
listsize);  
        print(&array[i], listsize);  
    }  
    cout << "Sort with list size " << listsize  
        << ", array size " << ARRAYSIZE  
        << ", and threshold " << THRESHOLD  
        << ": " << Gettime() << " seconds  
    \n";
```

数组规模	100	10K	1M	10K正序	10K逆序
直接插入排序	0.000175	1.7266	_____	0.00031	3.44187
优化插入排序	0.000092	0.8847	_____	0.00031	1.76157
二分插入排序	0.000036	0.1434	_____	0.00453	0.28297
冒泡排序	0.000271	2.7844	_____	1.74641	3.47484
优化冒泡排序	0.000269	2.7848	_____	0.00032	3.44063
选择排序	0.000180	1.7199	_____	1.72516	1.72812
Shell排序(2)	0.000072	0.0166	4.578	0.00484	0.00781
Shell排序(3)	0.000055	0.0153	4.125	0.00121	0.00687

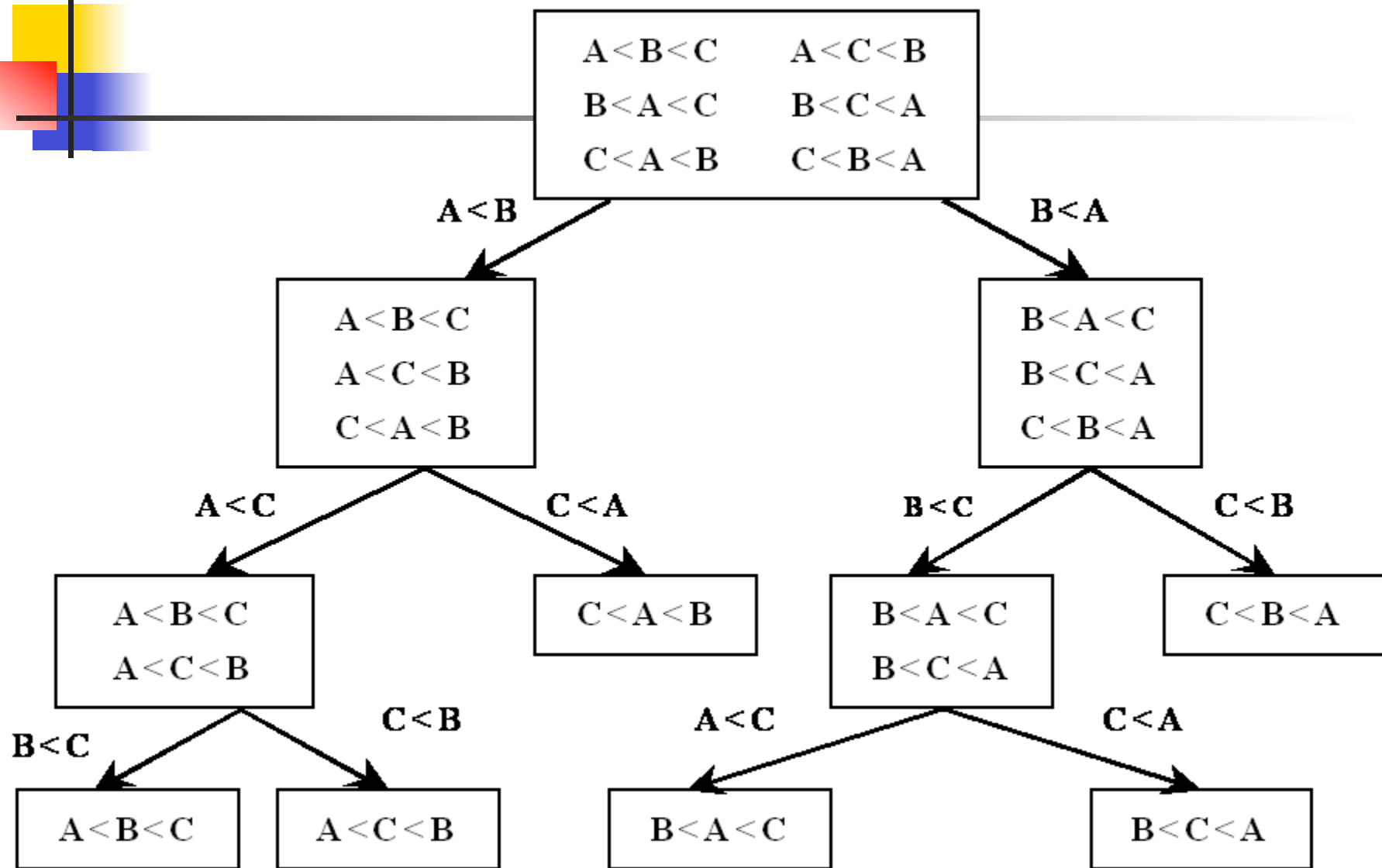
数组规模	100	10K	1M	10K正序	10K逆序
快速排序	0.000042	0.0068	1.001	0.00547	0.00547
优化快速排序	0.000031	0.0057	0.856	0.00408	0.00409
归并排序	0.000045	0.0070	1.105	0.00625	0.00546
优化归并排序	0.000033	0.0057	0.934	0.00531	0.00531
堆排序	0.000042	0.0075	1.562	0.00672	0.00688
基数排序/2	0.000294	0.0294	3.031	0.02937	0.02922
基数排序/4	0.000145	0.0147	1.515	0.01469	0.01469
基数排序/8	0.000095	0.0092	0.953	0.00922	0.00921



7.8 排序问题的下限

- 排序问题的时间代价在 $\Omega(n)$ (起码I/O时间) 到 $O(n \log n)$ (平均, 最差情况) 之间
- 基于比较的排序算法的下限也为 $\Theta(n \log n)$

用判定树模拟基于比较的排序





判定树(Decision Tree)

- 判定树中叶结点的最大深度就是排序算法在最差情况下需要的最大比较次数；
- 叶结点的最小深度就是最佳情况下的最小比较次数



基于比较的排序的下限

- 对 n 个记录，共有 $n!$ 个叶结点
- 判定树的深度为 $\log(n!)$
- 在最差情况下需要 $\log(n!)$ 次比较，即 $\Omega(n \cdot \log n)$
- 在最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较



排序问题的时间代价

- 在最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较，因此最差情况时间代价就是 $\Omega(n \cdot \log n)$ 。
- 那么排序问题本身需要的运行时间也就是 $\Omega(n \cdot \log n)$ 。
- 所有排序算法都需要 $\Theta(n \cdot \log n)$ 的运行时间，因此可以推导出排序问题的时间代价为 $\Theta(n \cdot \log n)$ 。



END!
