



第八章 内排序

张铭 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008.6（“十二五”国家级规划教材）

<http://jpk.pku.edu.cn/course/sjg/>

<https://www.icourse163.org/course/PKU-1002534001>

大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of Computer Programming

VOLUME 3

Sorting and Searching
Second Edition

DONALD E. KNUTH

8.1 基本概念

8.1 基本概念

- **序列** (Sequence): 线性表
 - 由记录组成
- **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
 - 有人翻译为 “键”
- **排序码** (Sort Key): 作为排序运算依据的一个或多个域

学号	姓名
2100015428	张无忌
2100013098	赵敏
2100013055	闫之翰
2100013185	肖滨琪
2100013085	郑锡宇
2000017767	赵敏
2100017794	张星博

什么是排序？

• 排序

- 将序列中的记录**按照排序码顺序排列**起来
- 排序码域的值具有**不减**（或**不增**）的顺序
- 排序码不重复，则**递增**或**递减**

• 内排序

- 整个排序过程在**内存**中完成

排序问题

- 给定一个**序列** $R = \{ r_0, r_1, \dots, r_{n-1} \}$
 - 其排序码分别为 $k = \{ k_0, k_1, \dots, k_{n-1} \}$
- 排序的目的：将记录**按排序码重排**
 - 形成新的有序序列 $R' = \{ r'_0, r'_1, \dots, r'_{n-1} \}$
 - 相应排序码为 $k' = \{ k'_0, k'_1, \dots, k'_{n-1} \}$
- 排序码的顺序
 - 其中 $k'_0 \leq k'_1 \leq \dots \leq k'_{n-1}$ ，称为**不减序**
 - 或 $k'_0 \geq k'_1 \geq \dots \geq k'_{n-1}$ ，称为**不增序**

正序 vs. 逆序

- “**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
- 例如，要求不升序列
 - 08 12 34 96
 - 96 34 12 08

正序!

逆序!

排序的稳定性

- 稳定

- 存在多个具有相同排序码的记录
- 排序后这些记录的**相对次序保持不变**

- 例如,

- 34 12 34' 08 96

- 08 12 34 34' 96

稳定!

- 稳定性的证明——**形式化证明**

8.1 基本概念

排序的稳定性

- 不稳定

- 存在多个具有相同排序码的记录
- 排序后这些记录的**相对次序可能变化**

- 例如,

- 34 12 34' 08 96
- 08 12 34' 34 96

不稳定!

- 不稳定性的证明——反例说明

排序算法的衡量标准

- **时间**代价：记录的**比较**和**移动**次数
- **空间**代价
- 算法本身的**繁杂程度**

45

34

78

12



思考

1. 排序算法的**稳定性**有何**意义**？
2. 为何需要考虑 “**正序**” 与 “**逆序**” 序列？



大纲

- 8.1 排序问题的基本概念
- **8.2 插入排序 (Shell 排序)**
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

8.2 插入排序

- 8.2.1 直接**插入**排序
- 8.2.2 Shell 排序



8.2 插入排序

插入排序动画

45 34 78 12 34 32 29 64

8.2 插入排序

插入排序算法

```
template <class Record>
void ImprovedInsertSort (Record Array[], int n){
//Array[] 为待排序数组, n 为数组长度
    Record TempRecord;           // 临时变量
    for (int i=1; i<n; i++){      // 依次插入第 i 个记录
        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) && (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;
    }
}
```

12

34

45

78

34'

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, $2(n-1)$ 次移动, $\Theta(n)$
 - 最差情况: $\Theta(n^2)$

- 比较次数

$$\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$$

- 移动次数

$$\sum_{i=1}^{n-1} (i+2) = (n-1)(n+4)/2 = \Theta(n^2)$$

- 平均情况: $\Theta(n^2)$



8.2.2 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**基本有序**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

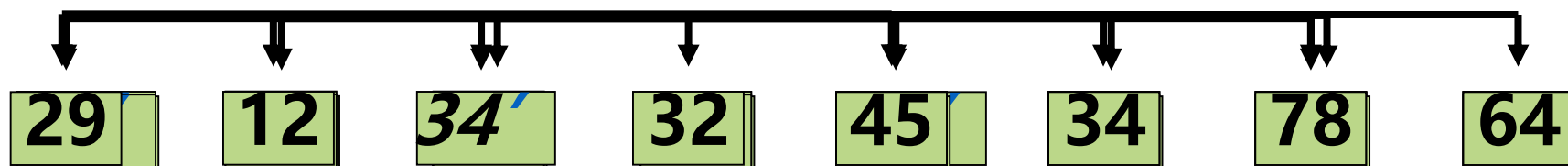


Shell排序算法思想

- 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**增加小序列的规模**，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

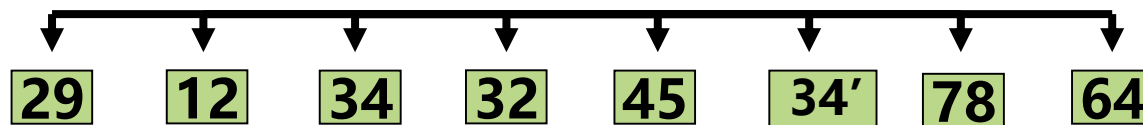
8.2.2 Shell排序

Shell排序动画



“增量每次除以2递减”的Shell 排序

```
template <class Record>
void ShellSort(Record Array[], int n) {
// Shell排序, Array[]为待排序数组, n为数组长度
    int i, delta;
// 增量delta每次除以2递减
    for (delta = n/2; delta>0; delta /= 2)
        for (i = 0; i < delta; i++)
            // 分别对delta个子序列进行插入排序
            // “&”传 Array[i]的地址, 数组总长度为n-i
            ModInsSort(&Array[i], n-i, delta);
// 如果增量序列不能保证最后一个delta间距为1
// 可以安排下面这个扫尾性质的插入排序
// ModInsSort(Array, n, 1);
}
```





针对增量而修改的插入排序算法

```
template <class Record> // 参数delta表示当前的增量
void ModInsSort(Record Array[], int n, int delta) {
    int i, j;
    for (i = delta; i < n; i += delta) // 第i个记录找插入位置
        // j以delta为步长向前寻找逆置对进行调整
        for (j = i; j >= delta; j -= delta) {
            if (Array[j] < Array[j-delta]) // 逆置对
                swap(Array, j, j-delta); // 交换
            else break;
        }
}
```

Shell 排序算法分析

- 不稳定
- 空间代价: $\Theta(1)$
- 时间代价
 - 增量每次除以2递减, $\Theta(n^2)$

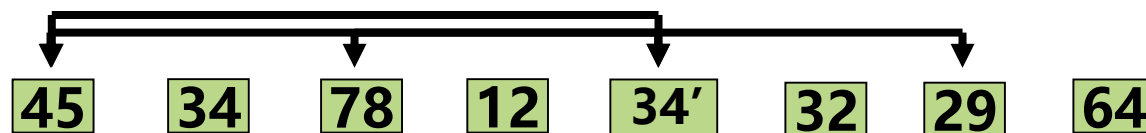
Shell 排序选择增量序列

- 增量每次除以2递减

- 效率仍然为 $\Theta(n^2)$

- 问题：选取的**增量**之间**并不互质**

- 间距为 2^{k-1} 的子序列，都是由那些间距为 2^k 的子序列组成的
 - 上一轮循环中在同一个子序列中，可能下一轮还在同一个子序列中，**重复处理**，导致处理效率不高





Hibbard 增量序列

- Hibbard 增量序列
 - $\{2^k - 1, 2^{k-1} - 1, \dots, 7, 3, 1\}$
- Shell(3) 和 Hibbard 增量序列的 Shell 排序的效率可以达到 $\Theta(n^{3/2})$
- 选取其他增量序列还可以更进一步减少时间代价

Shell最好的代价

- 呈 $2^p 3^q$ 形式的一系列整数：
- 1, 2, 3, 4, 6, 8, 9, 12
- 算法代价 $\Theta(n \log^2 n)$



思考

- 1. 插入排序的**变种**
 - 发现**逆序对直接交换**
 - **二分插入排序**: 查找待插入位置时, 采用二分法
- 2. Shell 排序中**增量作用**是什么? 增量除以2递减和增量除以3递减的序列, 哪个更好? 为什么?
- 3. Shell 排序的**每一轮子序列排序**可以用其他方法吗?



大纲

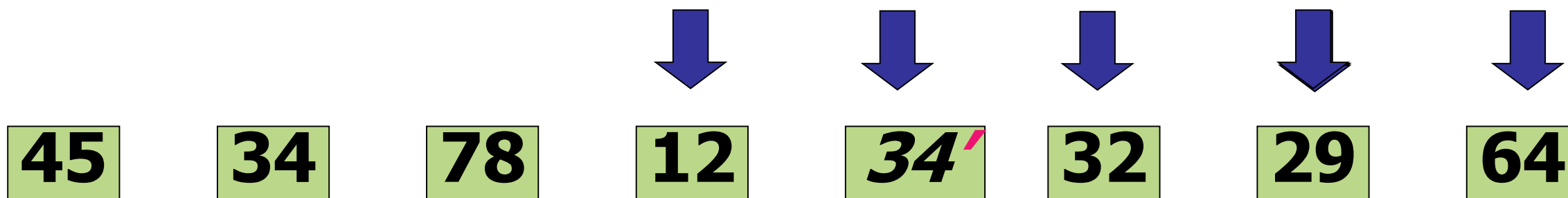
- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- **8.3 选择排序 (堆排序)**
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

8.3 选择排序

8.3 选择排序

- 8.3.1 **直接选择排序**
 - 依次选出剩下的未排序记录中的最小记录
- 8.3.2 **堆排序**
 - 堆排序：基于最大堆来实现

直接选择排序动画



8.3 选择排序

直接选择排序

```
template <class Record>
void SelectSort(Record Array[], int n) {
// 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 (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);
    }
}
```

直接选择排序性能分析

- 不稳定
- 空间代价: $\Theta(1)$
- 时间代价
 - 比较次数: $\Theta(n^2)$
 - 交换次数: $n-1$
 - 总时间代价: $\Theta(n^2)$

8.3.2 堆排序

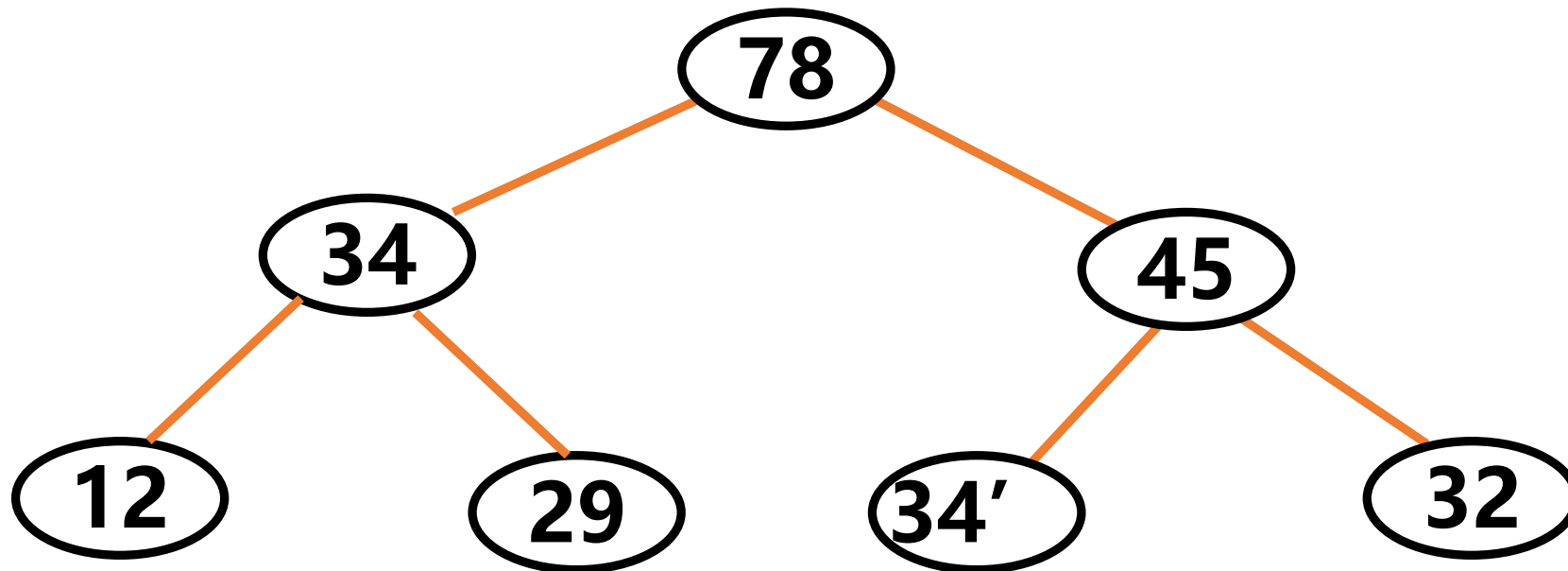
- **选择类的内排序**

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

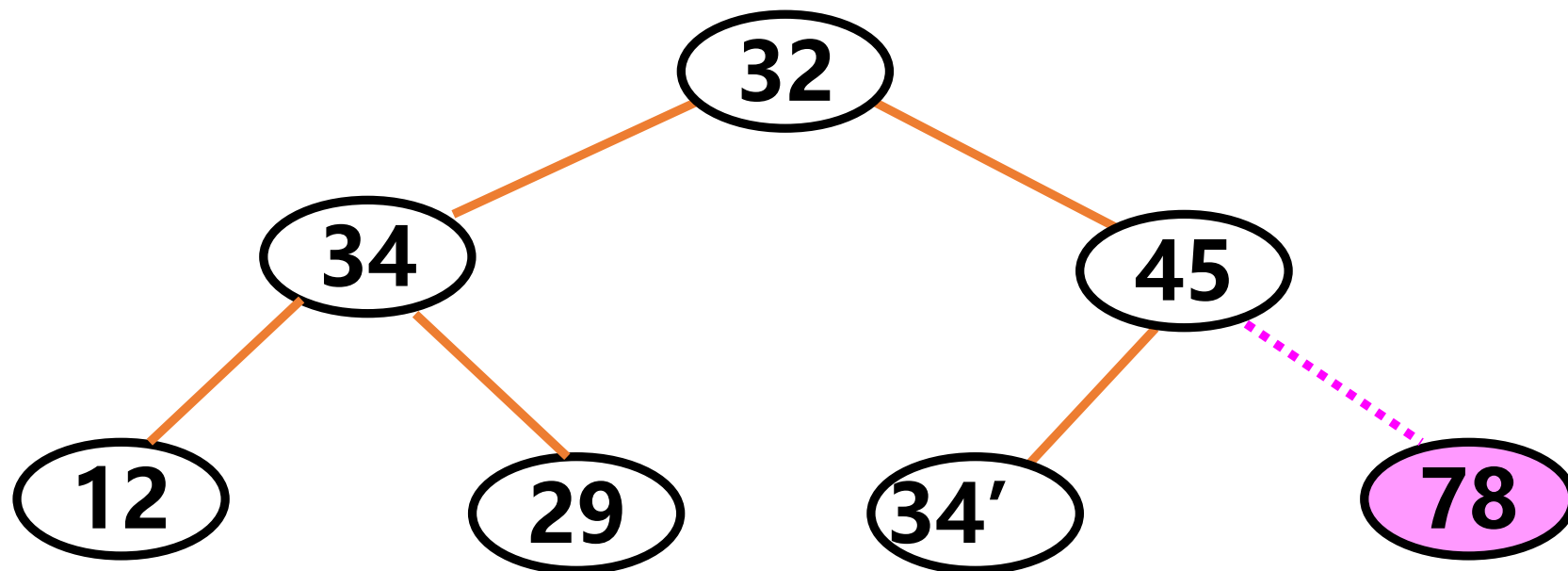
- **选择类的外排序**

- 置换选择排序
- 赢者树、败方树

最大堆排序过程示意图



最大堆排序过程示意图



堆排序算法

```
template <class Record>
void sort(Record Array[], int n){
    int i;
    // 建堆
    MaxHeap<Record> max_heap
        = MaxHeap<Record>(Array,n,n);
    // 算法操作n-1次, 最小元素不需要出堆
    for (i = 0; i < n-1; i++)
        // 依次找出剩余记录中的最大记录, 即堆顶
        max_heap.RemoveMax();
}
```

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 $\Theta(1)$

思考

- 直接选择排序为什么**不稳定**？怎么修改一下让它变稳定
- **改写堆排序**算法，发现**逆序对直接交换**

大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- **8.4 交换排序**
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

8.4 交换排序

- 8.4.1 冒泡排序
- 8.4.2 快速排序

8.4.1 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
- 检查**每轮冒泡过程中是否**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

8.4.1 冒泡排序

冒泡排序动画



8.4.1 冒泡排序

冒泡排序算法

```
template <class Record>
void BubbleSort(Record Array[], int n) {
    bool NoSwap;                // 是否发生了交换的标志
    int i, j;
    for (i = 0; i < n-1; i++) {
        NoSwap = true;         // 标志初始为真
        for (j = n-1; j > i; j--){
            if (Array[j] < Array[j-1]) { // 判断是否逆置
                swap(Array, j, j-1);    // 交换逆置对
                NoSwap = false;         // 发生了交换，标志变为假
            }
            if (NoSwap)          // 没发生交换，则已完成排好序
                return;
        }
    }
}
```

8.4.1 冒泡排序

算法分析

- **稳定**
- **空间**代价: $\Theta(1)$
- 时间代价分析
 - 比较次数
 - 最少: $\Theta(n)$
 - 最多: $\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = \Theta(n^2)$
- 交换次数最少为 0, 最多为 $\Theta(n^2)$, 平均为 $\Theta(n^2)$
- **时间**代价结论
 - 最大, **平均**时间代价均为 $\Theta(n^2)$
 - **最小**时间代价为 $\Theta(n)$: 最佳情况下只运行第一轮循环

8.4.2 快速排序

思考

- **冒泡**排序和**直接选择**排序哪个更优



第八章 内排序

张铭 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008.6（“十二五”国家级规划教材）

<http://jpk.pku.edu.cn/course/sjg/>

<https://www.icourse163.org/course/PKU-1002534001>



学习目标

· 经典的内排序算法思想

- 归纳法（插入）
- 分治法（归并、快排）
- 跨区虚拟数组操作（Shell）
- 地址排序的数据整理

· 算法时间复杂度分析方法

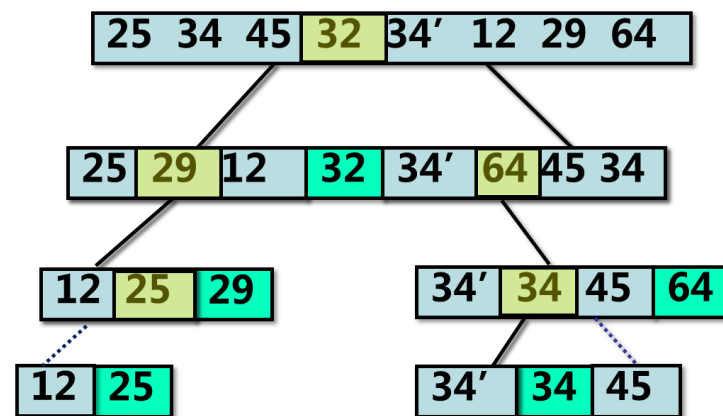
- 排序的基本操作：比较、移动
- 最好、最坏、平均情况排序算法
- 排序问题的决策树分析方法

大 纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- **8.4 交换排序**
 - 8.4.1 冒泡排序
 - **8.4.2 快速排序**
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

8.4.2 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
- 20世纪十大算法
 - Top 10 Algorithms of the Century
 - 1962 London 的 Elliot Brothers Ltd 的 Tony Hoare 提出的快速排序



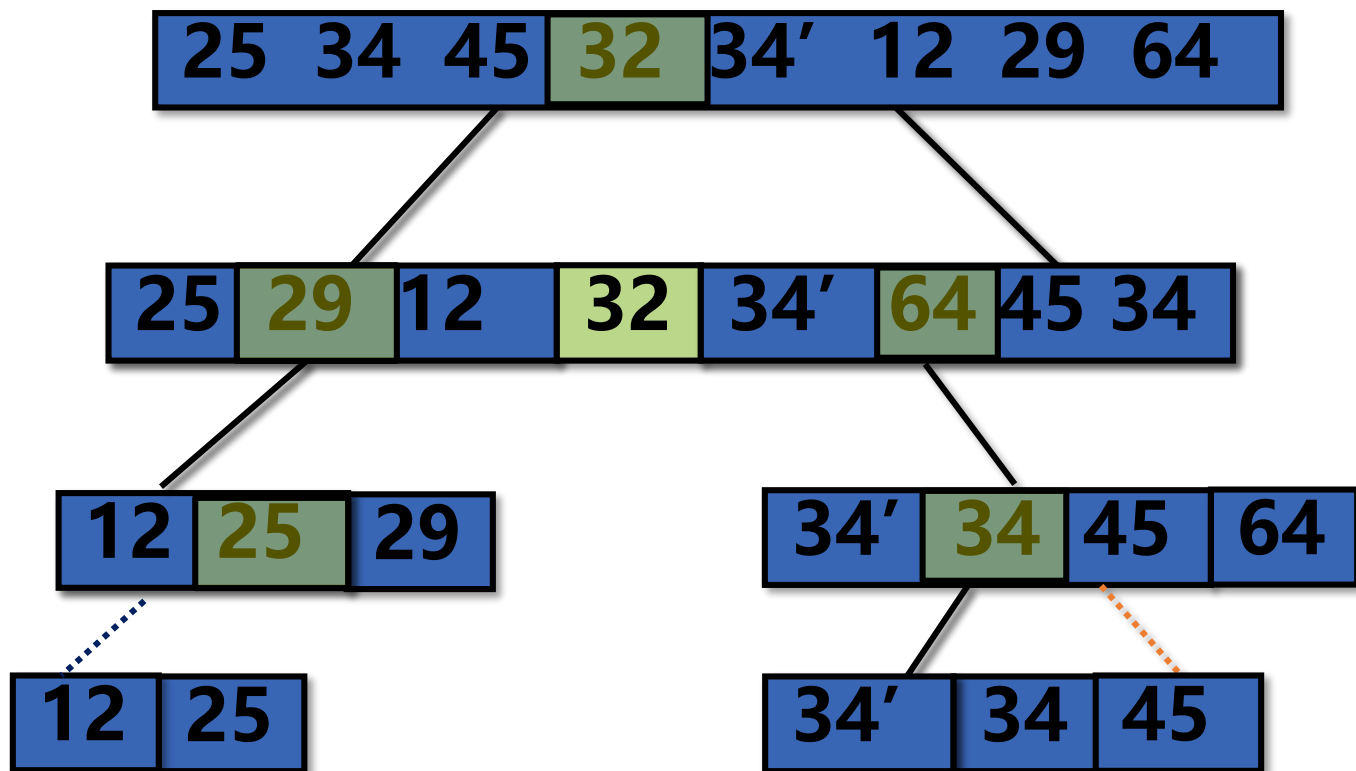


分治策略的基本思想

- 主要思想
 - **分**：划分子问题（子问题规模尽量均衡）
 - **治**：求解子问题（子问题不重叠）
 - **合**：综合子问题的解
- 分治策略的实例
 - BST 查找、插入、删除算法
 - 二分检索
 - 快速排序、归并排序



快速排序分治思想



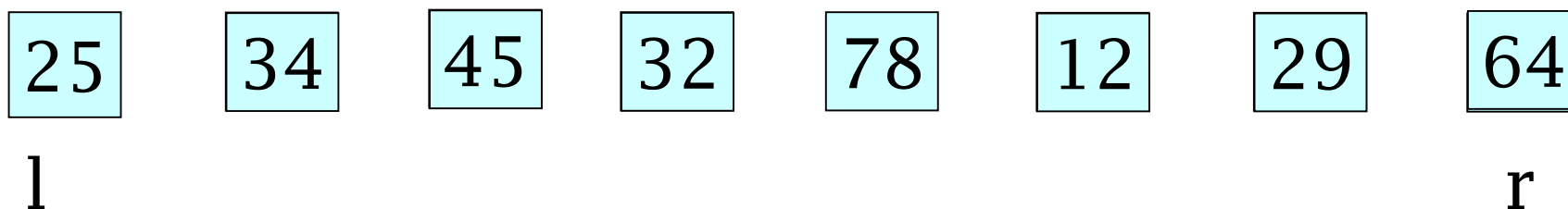
最终排序结果: 12 25 29 32 34' 34 45 64

轴值选择

- **L, R 长度尽可能接近**
- 选择策略：
 - 选择最左/最右
 - 随机选择 RQS
 - 选择平均值

8.4.2 快速排序

一次分割过程



- **选择轴值**并存储轴值
- 最后一个元素放到轴值位置
- 初始化**下标 l, r**，分别指向**头尾**
- i 递增直到遇到比轴值大的元素，将此元素覆盖到 j 的位置；j 递减直到遇到比轴值小的元素，将此元素覆盖到 i 的位置
- 重复上一步**直到 l==r**，将轴值放到 i 的位置，完毕

8.4.2 快速排序

快速排序算法

```
template <class Record>
void QuickSort(Record Array[], int left, int right) {
// Array[]为待排序数组，left,right分别为数组两端
    if (right <= left)           // 只有0或1个记录，就不需排序
        return;
    int pivot = SelectPivot(left, right);    // 选择轴值
    swap(Array, pivot, right);               // 轴值放到数组末端
    pivot = Partition(Array, left, right);    // 分割后轴值正确
    QuickSort(Array, left, pivot-1);          // 左子序列递归快排
    QuickSort(Array, pivot +1, right);        // 右子序列递归快排
}

int SelectPivot(int left, int right) {
// 选择轴值，参数left,right分别表示序列的左右端下标
    return (left+right)/2;                  // 选中间记录作为轴值
}
```

分割函数

```
template <class Record>
int Partition(Record Array[], int left, int right) {
// 分割函数，分割后轴值已到达正确位置
    int l = left;           // l 为左指针
    int r = right;          // r 为右指针
    Record TempRecord = Array[r]; // 保存轴值
    while (l != r) {        // l, r 不断向中间移动，直到相遇
        // l 指针向右移动，直到找到一个大于轴值的记录
        while (Array[l] <= TempRecord && r > l)
            l++;
        if (l < r) {        // 未相遇，将逆置元素换到右边空位
            Array[r] = Array[l];
            r--;            // r 指针向左移动一步
        }
    }
}
```

8.4.2 快速排序

```
// r 指针向左移动, 直到找到一个小于轴值的记录
while (Array[r] >= TempRecord && r > l)
    r--;
if (l < r) {                // 未相遇, 将逆置元素换到左空位
    Array[l] = Array[r];
    l++;                    // l 指针向右移动一步
}
} //end while
Array[l] = TempRecord; // 把轴值回填到分界位置 l 上
return l;              // 返回分界位置 l
}
```

时间代价

- 长度为 n 的序列，时间为 $T(n)$
 - $T(0) = T(1) = 1$
- 选择轴值时间为常数
- 分割时间为 cn
 - 分割后长度分别为 i 和 $n-i-1$
 - 左右子序列 $T(i)$ 和 $T(n-i-1)$
- 求解递推方程

$$T(n) = T(i) + T(n - i - 1) + cn$$

8.4.2 快速排序

最差情况

$$T(n) = T(n-1) + cn$$

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

...

$$T(2) = T(1) + c(2)$$

- 总的时间代价为：

$$T(n) = T(1) + c \sum_{i=2}^n i = \Theta(n^2)$$



最佳情况

$$T(n) = 2T(n/2) + cn$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c \log n$$

$$T(n) = cn \log n + n = \Theta(n \log n)$$

等概率情况

- 也就是说，轴值将数组分成长度为 0 和 $n-1$, 1 和 $n-2$, ... 的子序列的概率是相等的，都为 $1/n$
- $T(i)$ 和 $T(n-i-1)$ 的平均值都是

$$T(i) = T(n - i - 1) = \frac{1}{n} \sum_{k=0}^{n-1} T(k)$$

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n - k - 1)) = cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k)$$

$$nT(n) = (n+1)T(n-1) + 2cn - c$$

$$T(n) = \Theta(n \log n)$$

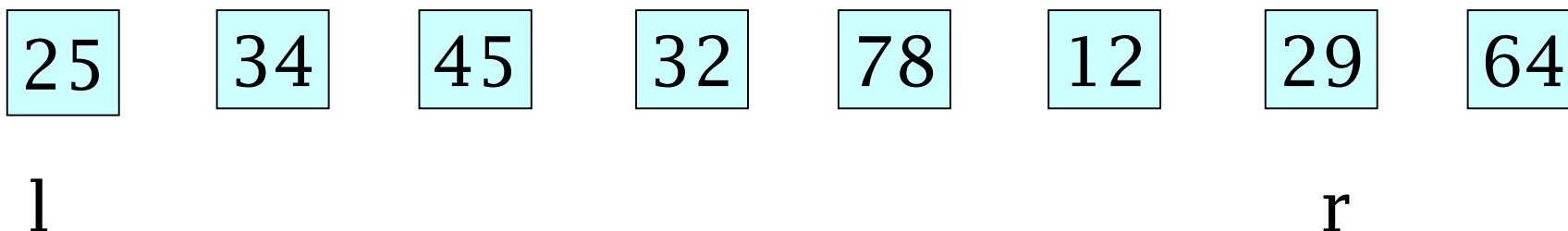


快速排序算法分析

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



另一种分割方案



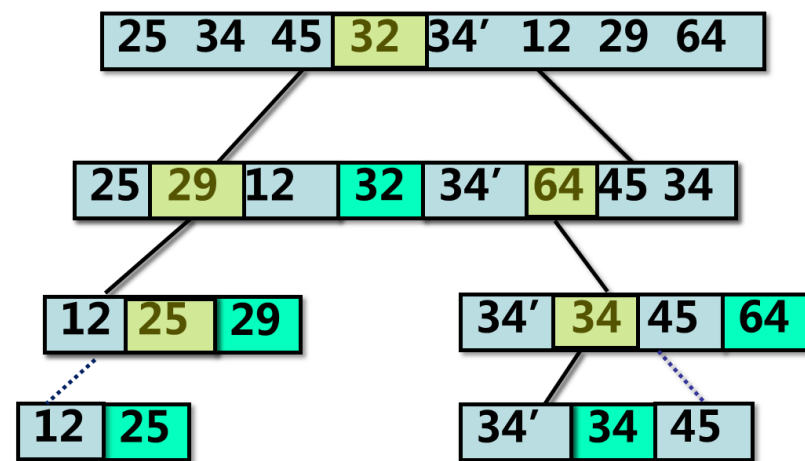
- **选择轴值**并与最后一个元素交换
- 初始化下标 **l, r**, 分别指向待分序列的**头尾**
- l 递增直到遇到比轴值大的元素, r 递减直到遇到比轴值小的元素, **两元素交换**
- 重复上面的步骤, **直到 $l > r$** , 将轴值(在最后)与位置 l 的元素交换

一个逆序对互相交换的分割方法

```
template <class Record>                // 初始时, 轴值 pivot 被放置在最右边
int partition(Record * Array, int l, int r) {
    int right = r;                       // 保留轴值的位置
    Record pivot = Array[r];
    l--;                                // 为了算法统一, 初始化 l = l - 1
    // 从两端向中间移动, 直到边界重合
    do { // 每个while 循环都是先拨动下标, 再进行比较
        while (Array[++l] < pivot);      // 左下标不需要判越界, 右移
        while ((r!=0) && (Array[--r] > pivot)); // 右下标左移, 注意要判下标不越界
        swap(array, l, r);
    } while (l < r);                     // l>=r 时才会停止
    swap(array, l, r);                   // 此算法有一次多余交换, 故应该换回来
    swap(array, l, right);               // 轴值入位
    return l;                            // 返回左下标值 l, 这就是轴值所在位置
}
```

思考

- **快速排序**为什么不**稳定**
- **快速排序**可能的**优化**
 - 轴值选择 RQS
 - 小子串不递归（阈值 28?）
 - 消除递归（用栈，队列?）





大 纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- **8.5 归并排序**
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

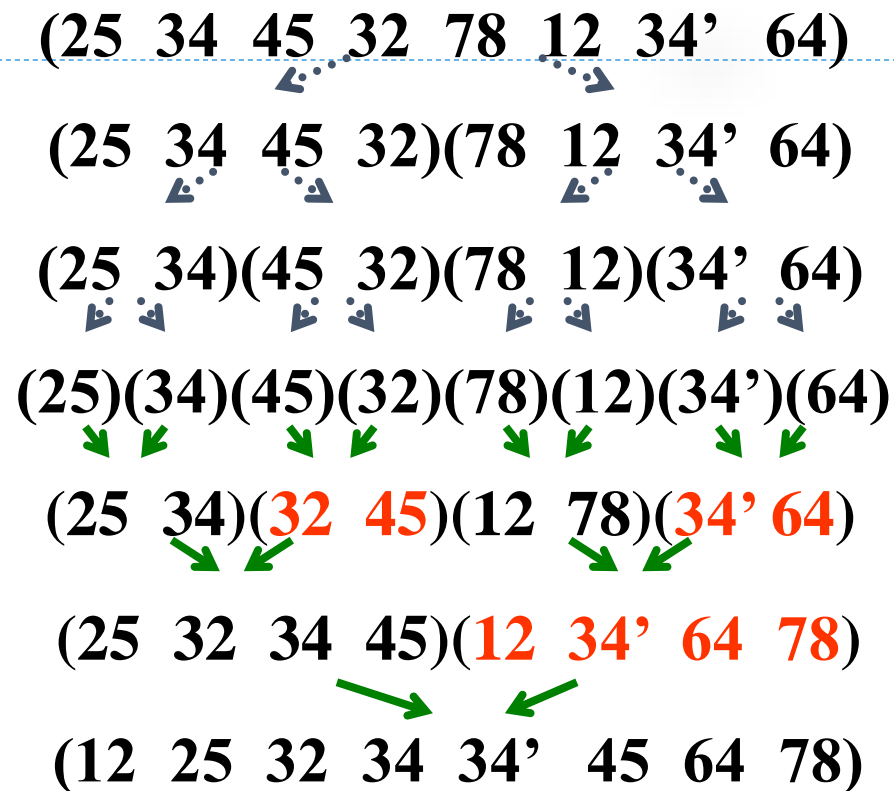
8.5 归并排序

归并排序思想

- 划分为**两个子序列**
- 对每个子序列**分别进行归并排序**
- 有序**子序列合并**

先划分

再归并



8.5 归并排序

两路归并排序

```
template <class Record>
void MergeSort(Record Array[], Record TempArray[], int left,
int right) {
    // Array为待排序数组, left, right两端
    int middle;
    if (left < right) { // 序列中只有0或1个记录, 不用排序
        middle = (left + right) / 2; // 平分为两个子序列
        // 对左边一半进行递归
        MergeSort(Array, TempArray, left, middle);
        // 对右边一半进行递归
        MergeSort(Array, TempArray, middle+1, right);
        Merge(Array, TempArray, left, right, middle); // 归并
    }
}
```

合并子序列的函数

```
// 两个有序子序列都从左向右扫描，归并到新数组
template <class Record>
void Merge(Record Array[], Record TempArray[], int left, int
right, int middle) {
    int i, j, index1, index2;
    // 将数组暂存入临时数组
    for (j = left; j <= right; j++)
        TempArray[j] = Array[j];
    index1 = left;           // 左边子序列的起始位置
    index2 = middle+1;       // 右边子序列的起始位置
    i = left;                // 从左开始归并
```

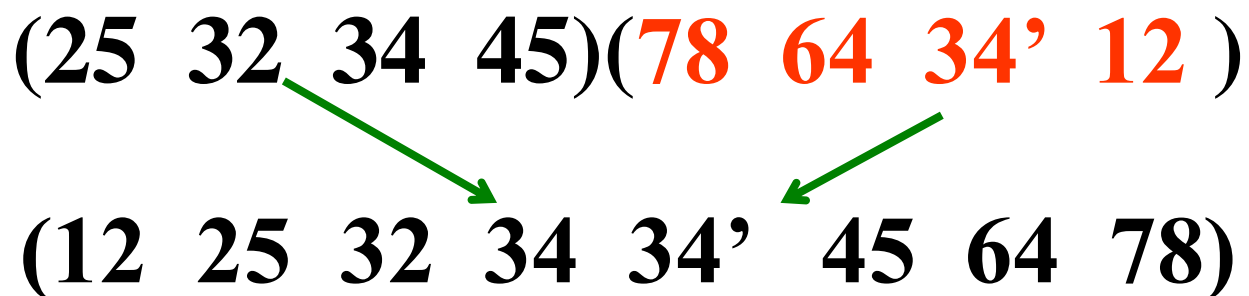
8.5 归并排序

```
while (index1 <= middle && index2 <= right) {  
    // 取较小者插入合并数组中  
    if (TempArray[index1] <= TempArray[index2])  
        Array[i++] = TempArray[index1++];  
    else Array[i++] = TempArray[index2++];  
}  
while (index1 <= middle) // 只剩左序列，可以直接复制  
    Array[i++] = TempArray[index1++];  
while (index2 <= right) // 与上个循环互斥，复制右序列  
    Array[i++] = TempArray[index2++];  
}
```

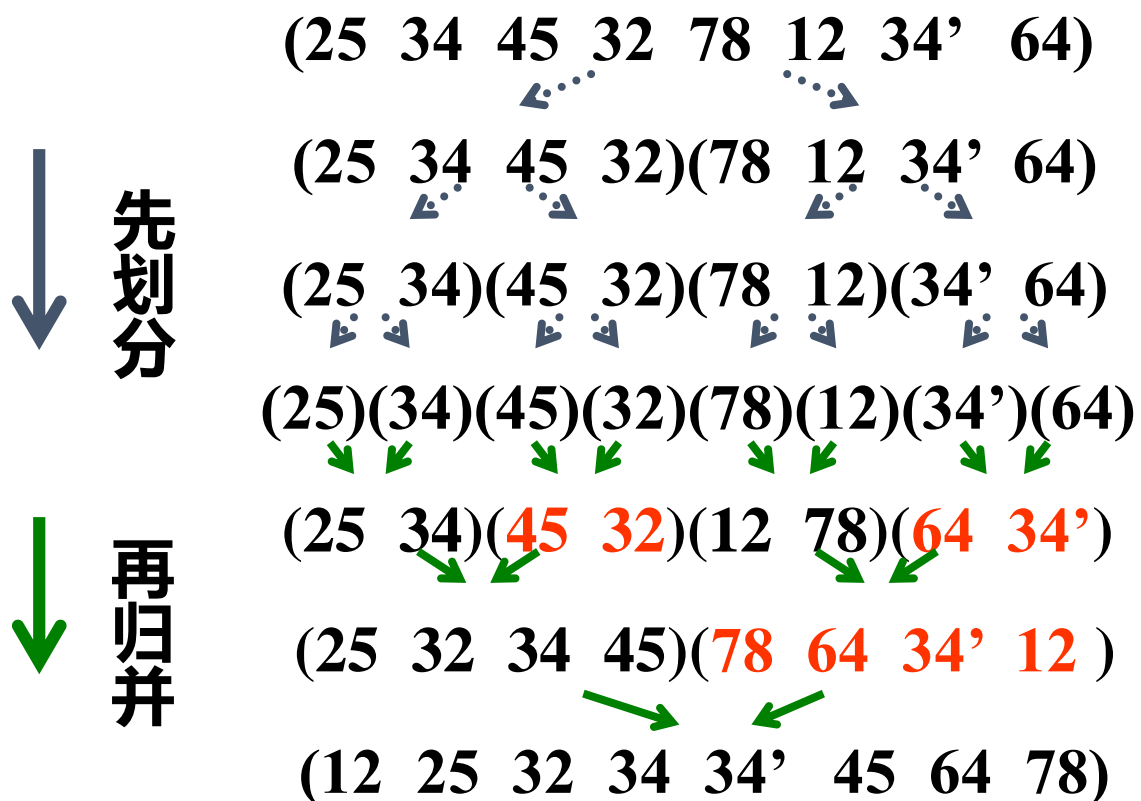
(25 32 34 45)(12 34' 64 78)
 ↘ ↙
(12 25 32 34 34' 45 64 78)

归并算法优化

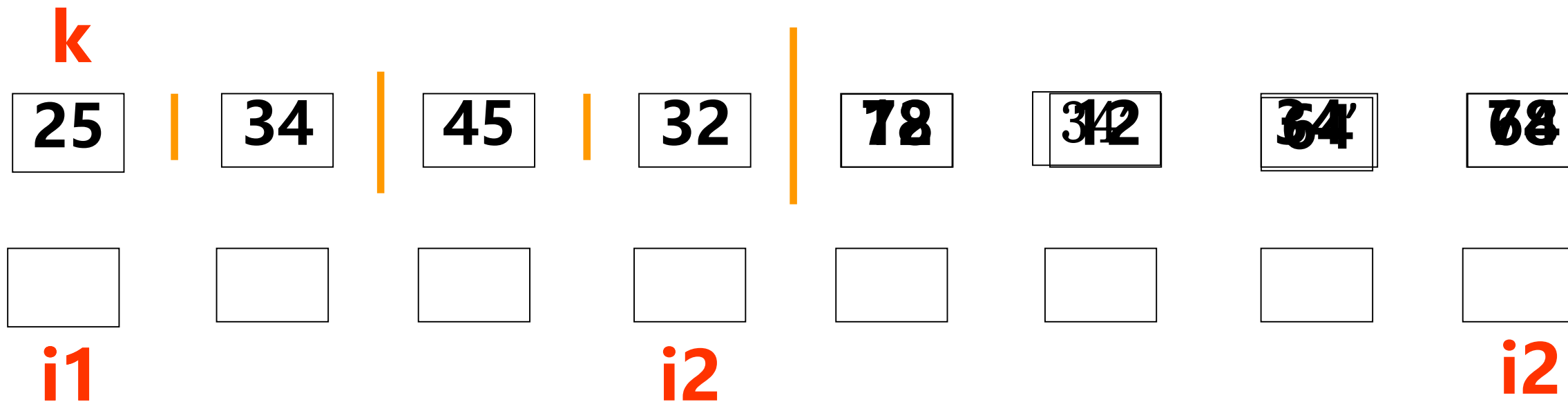
- 同优化的快速排序一样，对基本已排序序列直接插入排序
- R.Sedgwick 优化：先把**右子序列置逆**，归并时从两端开始处理，**向中间推进**，简化边界判断



R.Sedgwick优化归并思想



8.5 归并排序



R.Sedgewick优化归并



优化的归并排序（阈值28）

```
template <class Record>
void ModMergeSort(Record Array[], Record TempArray[], int left, int right) {
// Array为待排序数组，left，right两端
    int middle;
    if (right-left+1 > THRESHOLD) {    // 长序列递归
        middle = (left + right) / 2;    // 从中间划为两个子序列
        ModMergeSort(Array, TempArray, left, middle);    // 左
        ModMergeSort(Array, TempArray, middle+1, right);    // 右
        // 对相邻的有序序列进行归并
        ModMerge(Array, TempArray, left, right, middle);    // 归并
    }
    else InsertSort(&Array[left], right-left+1);    // 小序列插入排序
}
```

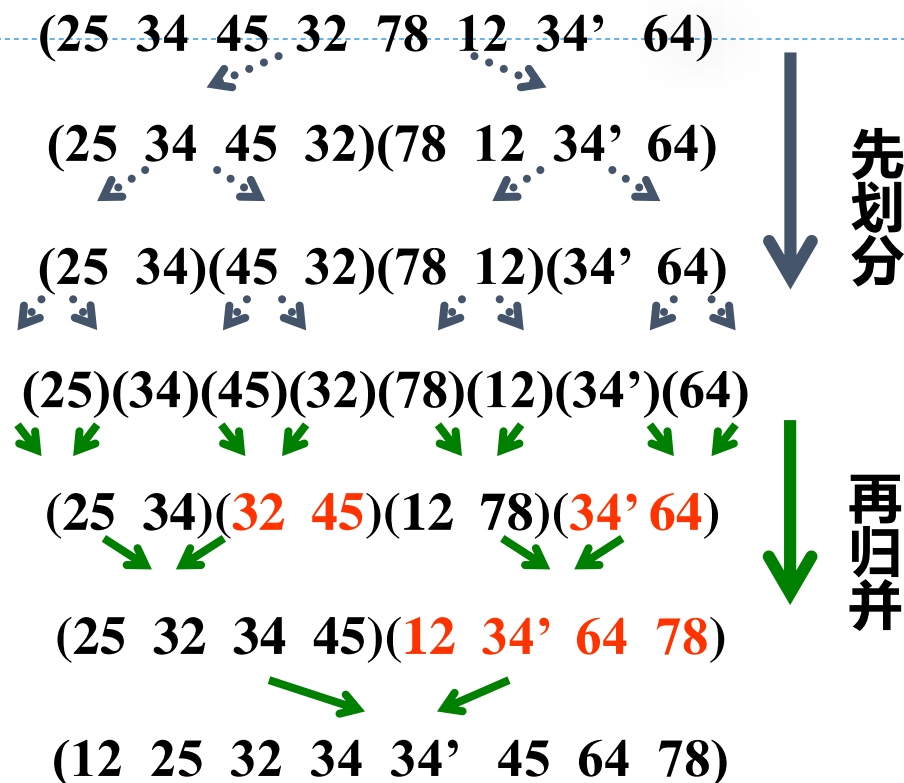
优化的归并函数

(25 32 34 45)(78 64 34' 12)
 ↘ ↙
(12 25 32 34 34' 45 64 78)

```
template <class Record> void ModMerge(Record Array[], Record  
TempArray[], int left, int right, int middle) {  
    int index1, index2;           // 两个子序列的起始位置  
    int i, j, k;  
    for (i = left; i <= middle; i++)  
        TempArray[i] = Array[i];    // 复制左边的子序列  
    for (j = 1; j <= right - middle; j++)    // 颠倒复制右序列  
        TempArray[right - j + 1] = Array[j + middle];  
    for (index1 = left, index2 = right, k = left; k <= right; k++)  
        if (TempArray[index1] <= TempArray[index2])  
            Array[k] = TempArray[index1++];  
        else  
            Array[k] = TempArray[index2--];  
}
```


算法复杂度分析

- **空间**代价: $\Theta(n)$
- 划分时间、排序时间、归并时间
$$T(n) = 2T(n/2) + cn$$
$$T(1) = 1$$
- 归并排序总时间代价也为 $\Theta(n \log n)$
- 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 $\Theta(n \log n)$

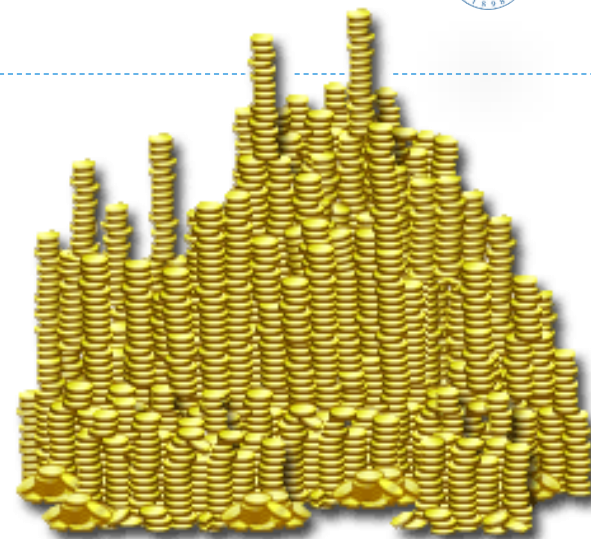


思考

- 普通归并和 Sedgewick 算法都是**稳定**的吗？
- 两个归并算法**哪个更优**？
 - 二者的**比较**次数和**赋值**次数
 - 归并时子序列下标是否需要**边界判断**

假币问题

- 9枚硬币，其中一枚是假的且重量较轻。请用一个没有砝码的天平，以最少的比较次数来寻找假币
- a,b,c,d,e,f,g,h 八枚硬币，用一个没有砝码的天平，以最少的比较次数找出假币
-





讨论：统计逆序对

- 给定 n 个数 $a_0, a_1 \dots a_{n-1}$
 - 如果存在存在 $a_i > a_j$ 且 $i < j$, 则称这样的元素对 $\langle a_i, a_j \rangle$ 为一个逆序对
 - 统计这 n 个数中逆序对的总数
- 比如说, $n=5$, a_0 到 a_4 分别为5,3,1,4,3
 - 则逆序对有 $\langle 5,3 \rangle, \langle 5,1 \rangle, \langle 5,4 \rangle, \langle 5,3 \rangle, \langle 3,1 \rangle, \langle 4,3 \rangle$ 共6对
- “逆序对(逆置)”的分析, 有助于插入排序等算法的效率分析

大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- **8.6 分配排序和索引排序**
 - **8.6.1 桶式排序**
 - 8.6.2 基数排序
 - 8.6.3 索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结

8.6 分配排序和基数排序

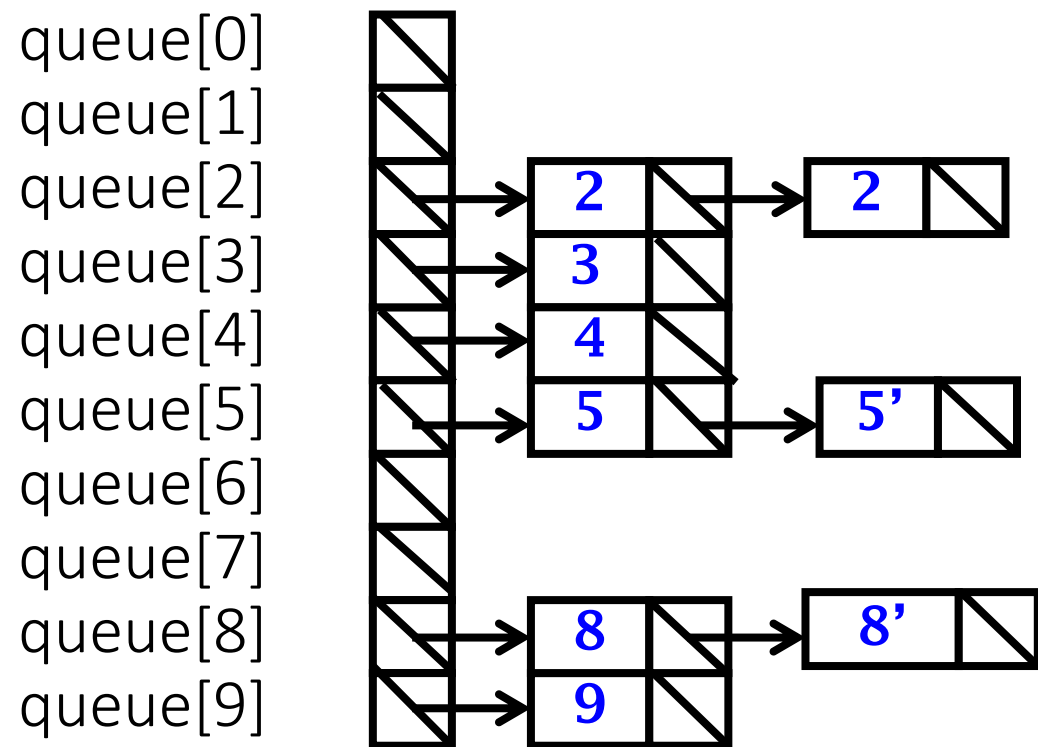
- 纪录之间**不需要**两两比较
- **需要**事先知道记录序列的**分布特征**

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

♠3 ♥J ♣8 ♥9 ♠9 ♦3 ♣1 ♦7

8.6.1 桶式排序

- 记录都位于某小区间段 $[0, m)$ 内, 分配到各桶, 再收集



4	3	2	5	2'	9	8	8'	5'
---	---	---	---	----	---	---	----	----

2	2'	3	4	5	5'	8	8'	9
---	----	---	---	---	----	---	----	---

8.6.1 桶式排序

待排数组: 7 3 8 9 6 1 8' 1' 2

每个桶
count分配:

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

+ + + +

前若干桶的
累计count:

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

8.6.1 桶式排序

桶排序示意

待排数组: 7 3 8 9 6 1 8' 1' 2

每个桶count:

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

前若干桶的
累计count:

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

收集:

0	1	2	3	4	5	6	7	8
1	1'	2	3	6	7	8	8'	9

8.6.1 桶式排序

桶式排序算法

```
template <class Record> void BucketSort(Record Array[], int n, int max) {  
    Record *TempArray = new Record[n]; // 临时数组  
    int *count = new int[max]; // 桶容量计数器  
    int i;  
    for (i = 0; i < n; i++) // 把序列复制到临时数组  
        TempArray[i] = Array[i];  
    for (i = 0; i < max; i++) // 所有计数器初始都为0  
        count[i] = 0;  
    for (i = 0; i < n; i++) // 统计每个取值出现的次数  
        count[Array[i]]++;  
    for (i = 1; i < max; i++) // 统计小于等于i的元素个数  
        count[i] = count[i-1] + count[i]; // c[i]记录i+1的起址  
    for (i = n-1; i >= 0; i--) // 尾部开始, 保证稳定性  
        Array[--count[TempArray[i]]] = TempArray[i];  
}
```

算法分析

- 数组长度为 n , 所有记录区间 $[0, m)$ 上
- **空间**代价:
 - 长度为 n 的临时数组, m 个计数器, $\Theta(n+m)$
- **时间**代价:
 - 统计**计数**: $\Theta(n+m)$, 输出有序序列时循环 n 次
 - **总的**时间代价**为** $\Theta(n+m)$
 - 适用于 m 相对于 n 很小的情况
- 稳定

思考

1. 桶排事先知道序列中的记录都位于某个小区间段 $[0, m)$ 内。**桶m多大**合适？超过这个范围怎么办？
2. 桶排中，**count 数组的作用**是什么？
3. 为什么桶排要**从后往前收集**？



大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- **8.6 分配排序和索引排序**
 - 8.6.1 桶式排序
 - 8.6.2 分配排序和基数排序
 - **顺序基数排序**
 - 链式基数排序
 - 8.6.3 索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结



8.6.2 基数排序

- 假设数组长度为 n , 所有记录区间 $[0, m)$ 上
- 桶式排序只适合 m 很小的情况
- **基数排序**: 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



排序码的多段划分

- 假设长度为 n 的序列

$$R = \{ r_0, r_1, \dots, r_{n-1} \}$$

记录的**排序码 K** 包含 **d 个子排序码**

$$K = (k_{d-1}, k_{d-2}, \dots, k_1, k_0)$$

- R 对排序码有序, 即对于任意

两个记录 R_i, R_j ($0 \leq i < j \leq n-1$), 都满足

$$(k_{i,d-1}, k_{i,d-2}, \dots, k_{i,1}, k_{i,0}) \leq$$

$$(k_{j,d-1}, k_{j,d-2}, \dots, k_{j,1}, k_{j,0})$$



例子

例如：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
 - 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次桶排**
- 4趟分配排序后，整个序列就排好序了

8.6.2 基数排序

黑桃♠(S) > 红心♥(H) > 方片♦(D) > 梅花♣(C)

♠3 ♥J ♣8 ♥9 ♠9 ♦3 ♣1 ♦7

- 高位先排，递归分治

- 先按花色：♣8 ♣1 ♦3 ♦7 ♥J ♥9 ♠3 ♠9

- 再按面值：♣1 ♣8 ♦3 ♦7 ♥9 ♥J ♠3 ♠9

- 低位先排，要求稳定排序！

- 先面值：♣1 ♠3 ♦3 ♦'7 ♣8 ♥9 ♠'9 ♥'J

- 再花色：♣1 ♣'8 ♦3 ♦'7 ♥9 ♥'J ♠3 ♠'9

高位优先法

- MSD, Most Significant Digit first
- 先处理**最高位** k_{d-1} 将序列分到若干桶中
- 然后再对每个桶处理**次高位** k_{d-2} , 分成更小的桶
- **依次重复, 直到对** k_0 排序后, 分成最小的桶, 每个桶内含有相同排序码 $(k_{d-1}, \dots, k_1, k_0)$
- **最后**将所有的桶中的数据**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
- **分、收**; **分、收**; ...; **分、收**的过程
 - 比较简单, 计算机常用



基数排序的实现

- 主要讨论 **LSD**
 - **基于顺序**存储
 - **基于链式**存储
- 原始输入数组 R 的长度为 n , 基数为 r , 排序码个数为 d

基于顺序存储的基数排序

初始数组内容: 97 53 88 59 26 41 88' 31 22

第一趟: count

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

按 count 分配桶:

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

收集:

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

(a) 第一趟处理个位

基于顺序存储的基数排序

第一次收集结果: 41 31 22 53 26 97 88 88' 59

0 1 2 3 4 5 6 7 8 9

第二趟: count

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

0 1 2 3 4 5 6 7 8 9

按 count 分配桶:

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

收集:

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

最终排序结果: 22 26 31 41 53 59 88 88' 97

(b) 第二趟处理十位

基于数组的基数排序

```
template <class Record>
void RadixSort(Record Array[], int n, int d, int r) {
    Record *TempArray = new Record[n];
    int *count = new int[r];      int i, j, k;
    int Radix = 1;                // 模进位, 用于取Array[j]的第i位
    for (i = 0; i <= d-1; i++) {  // 对第 i 个排序码分配
        for (j = 0; j < r; j++)
            count[j] = 0;        // 初始计数器均为0
        for (j = 0; j < n; j++) { // 统计每桶记录数
            k = (Array[j] / Radix) % r; // 取第i位
            count[k]++;              // 相应计数器加1
        }
    }
}
```

8.6.2 基数排序

```
for (j = 1; j < r; j++)           // 给桶划分下标界
    count[j] = count[j-1] + count[j]; // count[i]记录了i+1桶的开始位置
for (j = n-1; j >= 0; j--) {      // 从数组尾部收集
    k = (Array[j] / Radix) % r;    // 取第 i 位
    count[k]--;                   // 桶剩余量计数器减1
    TempArray[count[k]] = Array[j]; // 入桶
}
for (j = 0; j < n; j++)           // 内容复制回 Array 中
    Array[j] = TempArray[j];
Radix *= r;                       // 修改模Radix
}
```




顺序基数排序代价分析

- 待排数组长度 n , 基为 r , 排序码分为 d 段
- 空间代价:
 - 临时数组, n
 - r 个计数器
 - **总空间代价 $\Theta(n+r)$**
- 时间代价
 - 每趟桶式排序: $\Theta(n+r)$
 - d 次桶式排序
 - **总时间代价 $\Theta(d \cdot (n+r))$**



大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- **8.6 分配排序和索引排序**
 - 8.6.1 桶式排序
 - 8.6.2 基数排序
 - 顺序基数排序
 - **链式基数排序**
 - 8.6.3 索引排序
- 8.7 排序算法的时间代价
- 内排序知识点总结



基于静态链的基数排序

- 假设长度为 n 的序列

$$R = \{ r_0, r_1, \dots, r_{n-1} \}$$

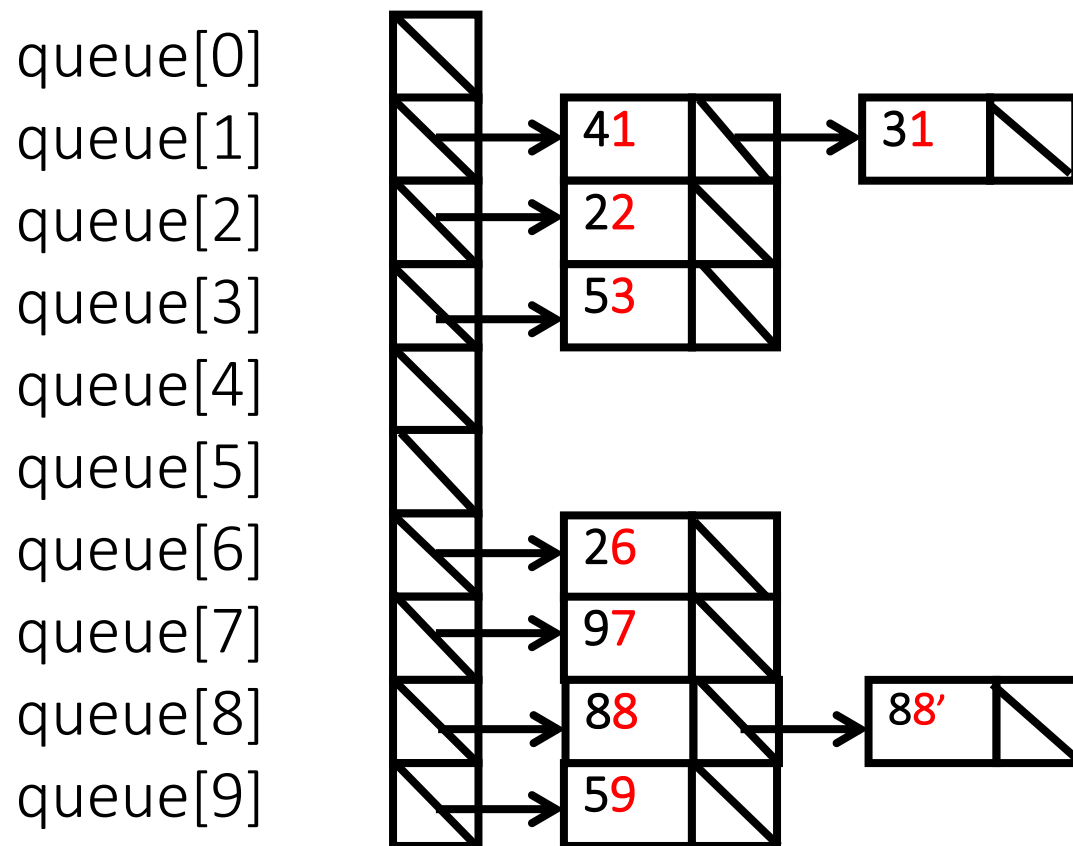
记录的**排序码 K** 包含 **d 个子排序码**

$$K = (k_{d-1}, k_{d-2}, \dots, k_1, k_0)$$

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

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

(a) 初始链表内容

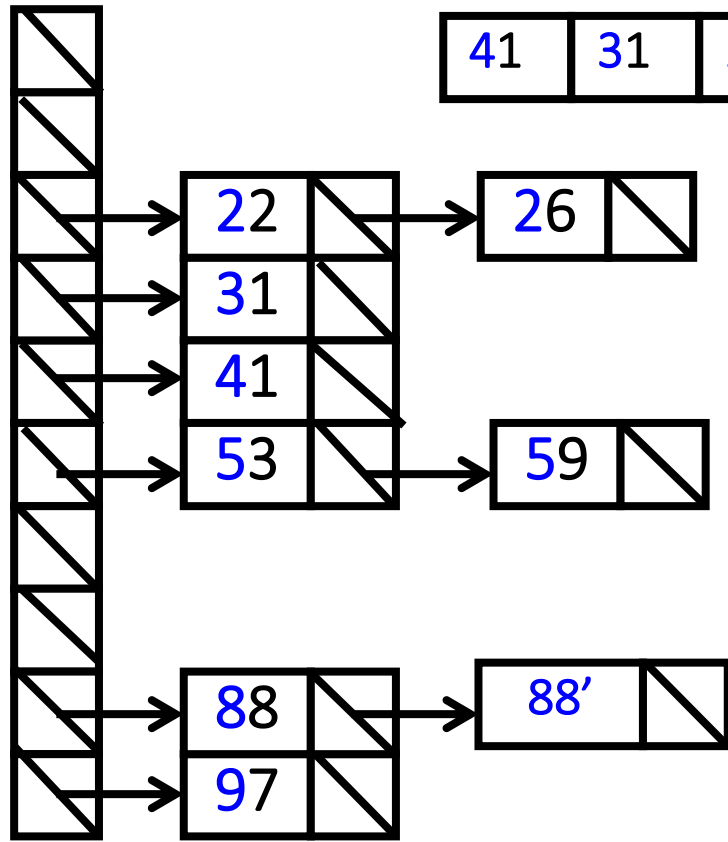


(b) 第一趟分配

(c) 第一趟收集

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

queue[0]
queue[1]
queue[2]
queue[3]
queue[4]
queue[5]
queue[6]
queue[7]
queue[8]
queue[9]



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

(d)第二趟分配

(e) 第二趟收集结果
(最终结果)

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

8.6.2 基数排序

静态队列定义

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

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

index	0	1	2	3	4	5	6	7	8
key		49	38	65	97	76	13	27	52
next	6	8	1	5	0	4	7	2	3



8.6.2 基数排序

基于静态链的基数排序

```
template <class Record>
void RadixSort(Record *Array, int n, int d, int r) {
    int i, first = 0;           // first指向第一个记录
    StaticQueue *queue = new StaticQueue[r];
    for (i = 0; i < n-1; i++)
        Array[i].next = i + 1; // 初始化静态指针域
    Array[n-1].next = -1;       // 链尾next为空
    // 对第i个排序码进行分配和收集，一共d趟
    for (i = 0; i < d; i++) {
        Distribute(Array, first, i, r, queue);
        Collect(Array, first, r, queue);
    }
    delete[] queue;
    AddrSort(Array, n, first); // 整理后，按下标有序
}
```

8.6.2 基数排序

```
template <class Record>
void Distribute(Record *Array, int first, int i, int r, StaticQueue *queue) {
    int j, k, a, curr = first;
    for (j = 0; j < r; j++) queue[j].head = -1;
    while (curr != -1) { // 对整个静态链进行分配
        k = Array[curr].key;
        for (a = 0; a < i; a++) // 取第i位排序码数字k
            k = k / r;
        k = k % r;
        if (queue[k].head == -1) // 把数据分配到第k个桶中
            queue[k].head = curr;
        else Array[queue[k].tail].next = curr;
        queue[k].tail = curr;
        curr = Array[curr].next; // curr移动，继续分配
    }
}
```


8.6.2 基数排序

```
template <class Record>
void Collect(Record *Array, int& first, int r, StaticQueue *queue) {
    int last, k=0;           // 已收集到的最后一个记录
    while (queue[k].head == -1) k++; // 找到第一个非空队
    first = queue[k].head; last = queue[k].tail;
    while (k < r-1) {        // 继续收集下一个非空队列
        k++;
        while (k < r-1 && queue[k].head == -1)
            k++;
        if (queue[k].head != -1) { // 试探下一个队列
            Array[last].next = queue[k].head;
            last = queue[k].tail;   // 最后一个为序列的尾部
        }
    }
    Array[last].next = -1;        // 收集完毕
}
```



链式基数排序算法代价分析

- 待排数组长度 n , 基为 r , 排序码分为 d 段
- 空间代价
 - n 个 next 域空间
 - r 个子序列的头尾指针
 - **总空间代价 $\Theta(n + r)$**
- 时间代价
 - 不需要移动记录本身,
只需要修改记录的 next 指针
 - **总空间代价 $\Theta(d \cdot (n+r))$**

8.6.2 基数排序

基数排序效率

- 因为 $r \ll n$, $\Theta(d \cdot (n+r))$ 可以看作 $\Theta(d \cdot n)$
- d 是比较小的数, 基数排序代价是 $\Theta(n)$ 吗?
- 实际上**还是** $\Theta(n \log n)$
 - **排序码不重复**的情况下, 需要 n 个不同的编码来表示
 - 段数 $d \geq \log_r n$, 即在 $\Omega(\log n)$ 中


大纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- **8.6 分配排序和索引排序**
 - 8.6.1 桶式排序
 - 8.6.2 基数排序
 - **8.6.3 索引排序**
- 8.7 排序算法的时间代价
- 内排序知识点总结

思考：链式基数排序的结果整理？

链式基数排序的结果，放在有头结点的单链表

index	0	1	2	3	4	5	6	7	8
key		49	38	65	97	76	13	27	52
next	6	8	1	5	0	4	7	2	3



整理之后： 13 27 38 49 52 65 76 97

线性时间整理静态链表

```
template <class Record>
void AddrSort(Record *Array, int n, int first) {
    int i, j;
    j = first;                // j待处理数据下标
    Record TempRec;
    for (i = 0; i < n-1; i++) { // 循环，每次处理第 i 个记录
        TempRec = Array[j];    // 暂存第 i 个的记录 Array[j]
        swap(Array[i], Array[j]);
        Array[i].next = j;     // next 链要保留调换轨迹j
        j = TempRec.next;     // j 移动到下一位
        while (j <= i)        // j 比 i 小，则是轨迹，顺链找
            j = Array[j].next;
    }
}
```

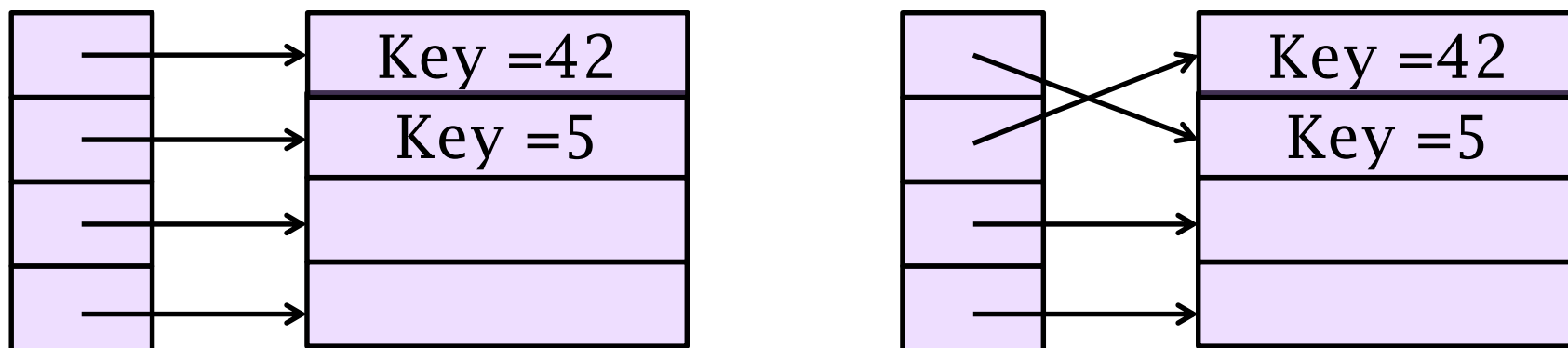
索引数组

Key =42
Key =5

Key =5
Key =42

数据域很大，交换记录的代价比较高

索引数组



交换指针，减少交换记录的次数

8.6.3 索引地址排序

索引结果:

- **谁该来?** 结果下标 `IndexArray[i]` 存放的是 `Array[i]` 中应该摆放的数据位置。
- **整理后 `Array[i]` 对应**原数组中
`Array[IndexArray[i]]`
- 下标 0 1 2 3 4 5 6 7
- 排序码 29 25 34 64 34' 12 32 45
- 索引 5 1 0 6 2 4 7 3

0	1	2	3	4	5	6	7
12	25	29	32	34	34'	45	64

对索引数组的顺链整理

• 下标	0	1	2	3	4	5	6	7
• 排序码	29	25	34	64	34'	12	32	45
• 索引	5	1	0	6	2	4	7	3
• 结果	0 12	1 25	2 29	3 32	4 34	5 34'	6 45	7 64



索引排序的适用性

- 一般的排序方法**都可以**
 - **赋值（或交换）都换成 对 index 索引数组的赋值（或交换）**
- 举例：插入排序

插入排序的索引地址排序版本

```
template<class Record>
void AddrSort(Record Array[], int n) {
    // n为数组长度
    int *IndexArray = new int[n], TempIndex;
    int i,j,k;
    Record TempRec;                // 只需一个临时空间
    for (i=0; i<n; i++)
        IndexArray[i] = i;
    for (i=1; i<n; i++)            // 依次插入第i个记录
        for (j=i; j>0; j--)        // 依次比较, 发现逆置就交换
            if ( Array[IndexArray[j]] < Array[IndexArray[j-1]])
                swap(IndexArray, j, j-1);
            else break;            // 此时i前面记录已排序
}
```

插入排序的索引地址排序版本(续)

// 调整为按下标有序

```
for(i=0;i<n;i++) {  
    j= i;  
    TempRec = Array[i];  
    while (IndexArray[j] != i) {  
        k=IndexArray[j];  
        Array[j]=Array[k];  
        IndexArray[j] = j;  
        j = k;  
    }  
    Array[j] =TempRec;  
    IndexArray[j] = j;  
}
```

下标	0	1	2	3	4	5	6	7
排序码	29	25	34	64	34'	12	32	45
索引	5	1	0	6	2	4	7	3
整理后:	12	25	29	32	34	34'	45	64

第二种索引方法

- **我去哪?** : 结果下标 `IndexArray[i]` 存放的是 `Array[i]` 中数据应该待的位置。
- 排好序的 **`Array[IndexArray[i]]`** 对应原数组中 **`Array[i]`**
- 下标 0 1 2 3 4 5 6 7
- 排序码 29 25 34 64 34' 12 32 45
- 索引 2 1 4 7 5 0 3 6

0	1	2	3	4	5	6	7
12	25	29	32	34	34'	45	64

对第二种索引的顺链整理

• 下标	0	1	2	3	4	5	6	7
• 排序码	29	25	34	64	34'	12	32	45
• 索引	2	1	4	7	5	0	3	6
• 结果	0 12	1 25	2 29	3 32	4 34	5 34'	6 45	7 64

思考

1. 证明**索引排序整理方案**的时间代价为 $\Theta(n)$
2. 修改快速排序，得到第一种索引结果。
3. 对**静态链的基数排序**结果进行简单变换得到**第二种索引**的方法。
4. 还有哪些**排序方法**可以得到第二种索引的结果？

大 纲

- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell 排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- **8.7 排序算法的时间代价**
- 内排序知识点总结

8.7 排序算法的时间代价

- 简单排序算法的时间代价
- 排序算法的**理论和实验**时间
- 排序**问题的下限**

简单排序算法 $\Theta(n^2)$ 的原因

- 一个长度为 n 的序列平均有 $n(n-1)/4$ 个逆置对
- 任何一种基于相邻记录比较的排序算法的平均时间代价都是 $\Theta(n^2)$

8.7.2 排序算法的理论和实验时间

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
冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
直接选择排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	不稳定

8.7.2 排序算法的理论和实验时间

算法	最大时间	平均时间	最小时间	辅助空间	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	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$	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
- Shell 排序选择**增量以3的倍数递减**
 - 需要保证最后一趟增量为1
- **综合性能快速排序最佳**



测试环境

- **硬件**环境
 - CPU: Intel P4 3G
 - 内存: 1G
- **软件**环境
 - Windows XP
 - Visual C++ 6.0



随机生成待排序数组

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


时间测试

```
#include <time.h>
#define CLOCKS_PER_SEC 1000
clock_t tstart = 0; // 开始的时间
// 初始化计时器
void Settime() {
    tstart = clock();
}
// 上次 Settime() 之后经过的时间
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>(&array[i], listsize);  
}  
cout << "Sort with list size " << listsize  
<< ", array size " << ARRAYSIZE << ", and  
threshold " <<  
THRESHOLD << ": " << Gettime() << "  
seconds\n";
```

8.7.2 排序算法的理论和实验时间

数组 规模	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
直接插入排序	0.00000047	0.000020	0.001782	0.1752	17.917	_____	0.00011	0.35094
直接选择排序	0.00000110	0.000041	0.002922	0.2778	36.500	_____	0.27781	0.29109
冒泡排序	0.00000160	0.000156	0.015620	1.5617	207.69	_____	0.00006	2.44840
Shell排序(2)	0.00000156	0.000036	0.000640	0.0109	0.1907	3.0579	0.00156	0.00312
Shell排序(3)	0.00000078	0.000016	0.000281	0.0038	0.0579	0.8204	0.00125	0.00687
堆排序	0.00000204	0.000027	0.000344	0.0042	0.0532	0.6891	0.00406	0.00375
快速排序	0.00000169	0.000021	0.000266	0.0030	0.0375	0.4782	0.00190	0.00199
优化快排/16	0.00000172	0.000020	0.000265	0.0020	0.0235	0.3610	0.00082	0.00088
优化快排/28	0.00000062	0.000011	0.000141	0.0018	0.0235	0.2594	0.00063	0.00063

8.7.2 排序算法的理论和实验时间

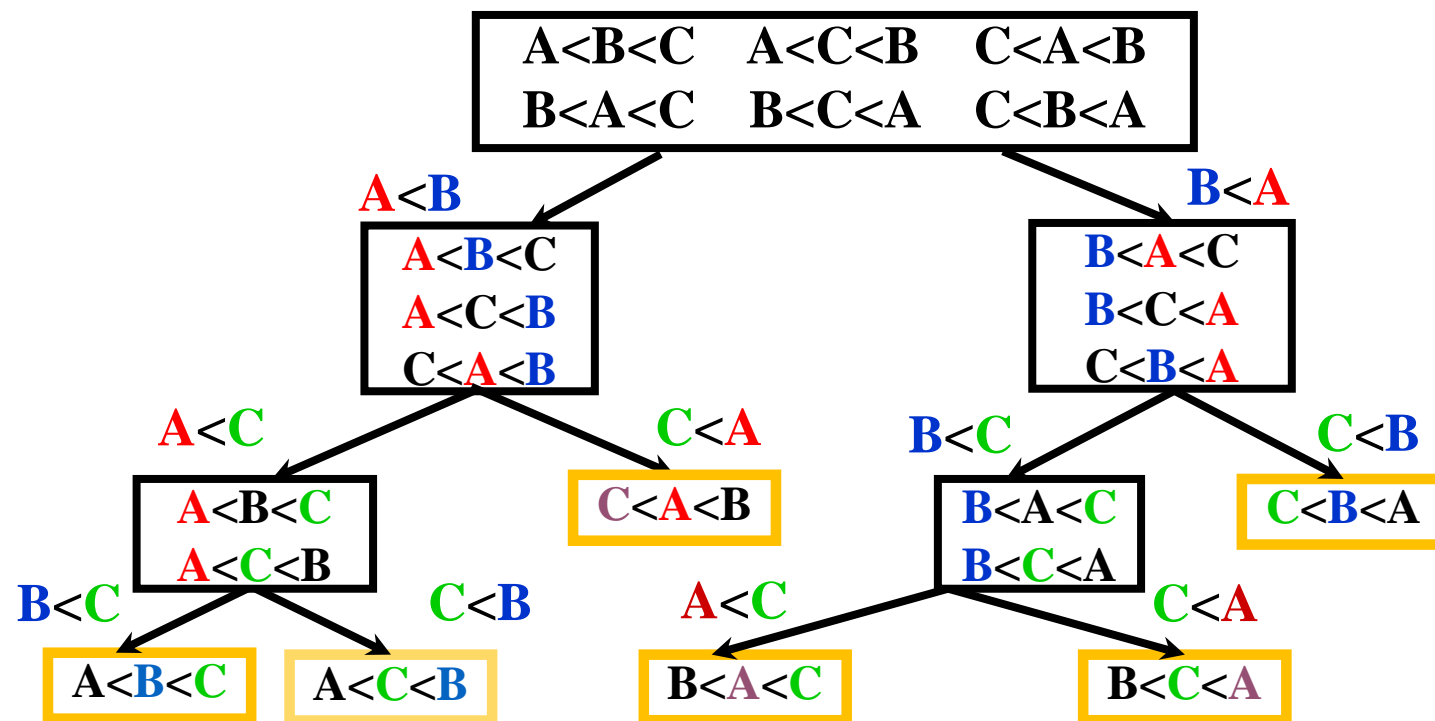
数组规模	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
归并排序	0.00000219	0.000028	0.000375	0.0045	0.0532	0.5969	0.00364	0.00360
优化归并/16	0.00000063	0.000014	0.000188	0.0030	0.0375	0.4157	0.00203	0.00265
优化归并/28	0.00000062	0.000013	0.000204	0.0027	0.0360	0.4156	0.00172	0.00265
顺序基数/8	0.00000610	0.000049	0.000469	0.0048	0.0481	0.4813	0.00484	0.00469
顺序基数/16	0.00000485	0.000034	0.000329	0.0032	0.0324	0.3266	0.00328	0.00313
链式基数/2	0.00002578	0.000233	0.002297	0.0234	0.2409	3.4844	0.02246	0.02281
链式基数/4	0.00000922	0.000075	0.000719	0.0075	0.0773	1.3750	0.00719	0.00719
链式基数/8	0.00000704	0.000048	0.000466	0.0049	0.0502	0.9953	0.00469	0.00469
链式基数/16	0.00000516	0.000030	0.000266	0.0028	0.0295	0.6570	0.00281	0.00281
链式基数/32	0.00000500	0.000027	0.000235	0.0028	0.0297	0.5406	0.00263	0.00266

排序问题的下限

- 排序问题的时间代价在 $\Omega(n)$ (单趟扫描) 到 $O(n \log n)$ (平均, 最差情况) 之间
- 用 **判定树 (Decision Tree)**
基于比较的排序算法的
下限也为 $\Omega(n \cdot \log n)$

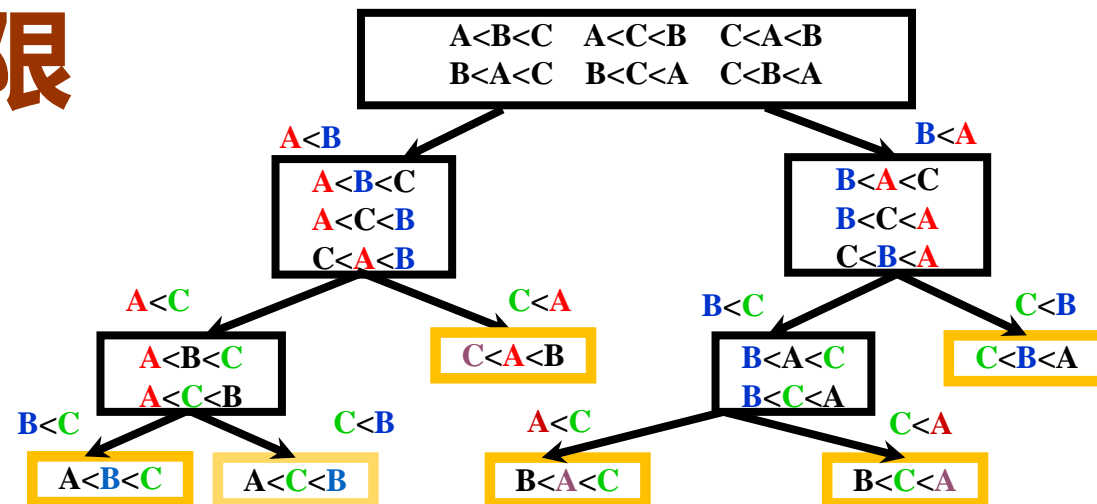
8.7.3 排序问题的下限

用判定树模拟基于比较的排序



基于比较的排序的下限

- 对 n 个记录，共有 $n!$ 个叶结点
 - 树的层数，至少为 $\Omega(\log n!)$
- $\Omega(\log(n!))$ 在 $\Omega(n \cdot \log n)$ 中
- 在最差情况下任何基于比较的排序算法都至少需要 $\Omega(n \log n)$ 次比较
- 而排序问题的上限是 $O(n \log n)$
- 因此可以推导出排序问题的时间代价为 $\Theta(n \cdot \log n)$



讨论:

排序算法时间代价: **比较、赋值**

一次交换是3次赋值, 请写出 `void swap(int &a, int &b)` 的实现

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap(int &a, int &b)
{
    a = a^b;
    b = a^b;
    a = a^b;
}
```

```
void swap(int &a, int &b)
{
    a = a + b;
    b = a - b;
    a = a - b;
}
```

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

讨 论:

1. 本章讨论的排序算法都是基于数组实现的，可否采用**动态链表**？性能上是否有差异？
2. 试总结并证明各种排序算法的**稳定性**，若算法稳定，如何修改可以使之不稳定？若算法不稳定，有没有必要改为稳定的？
3. 试调研**STL中的各种排序函数**是如何组合各种排序算法的。



数据结构与算法

感谢倾听

国家精品课 “数据结构与算法”

<http://jpk.pku.edu.cn/course/sjg/>

<https://www.icourse163.org/course/PKU-1002534001>

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。 “十二五” 国家级规划教材