数据结构与算法
第8章 内排序

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

主要内容

- **排序问题的基本概念**
- **排序方法**
  - 插入排序（Shell排序）
  - 选择排序（堆排序）
  - 交换排序
  - 归并排序
  - 分配排序和索引排序
- **排序算法的时间代价**

排序问题（Sorting）

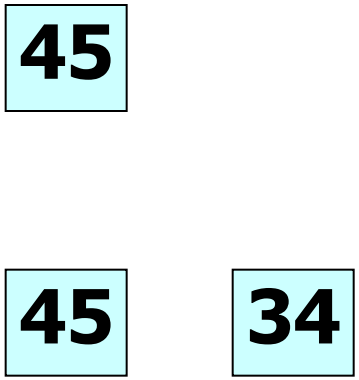
- Google等搜索引擎返回结果排级
- 图书馆员书目编号、上架
- 各种排名
  - 大学排名
  - 考试成绩排名
  - 福布斯 富豪榜
- Windows资源管理器，文件查看
- ……

排序

- 执行最频繁的计算任务之一
- 集前人的研究成果而得到若干经典、巧妙的算法
- 仍有诸多与排序相关的问题尚未解决，适应各种不同要求的新算法也在不断涌现
- 在经典的算法基础之上，可根据实际情况加以一定的改进、增删

小规模的排序问题

- 一个元素
  - 有序
- 两个元素
  - 一次比较
  - 若逆序?
    - 一次交换 = 3次移动（赋值）
- n个元素?



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(Sorting Key)：**

  作为排序依据的一个或多个域 （或关键码或其他域）

- **序列(Sequence)：**

  线性表，由记录组成的集合（排序中常用此词）

排序问题

- 将序列中的记录**按照排序码的顺序**排列起来
- **排序码域的值具有不减（或不增）的顺序**
  - **内排序**（Internal Sorting）：整个排序过程在内存中实施和完成
  - **外排序**（External Sorting）：内存无法容纳所有记录，排序过程中需要访问外存

本章讨论的都是**内排序**的方法，但有些方法（特别是归并排序的思想）也可用于外排序

排序问题再回顾

- 给定一个序列$R = \{r_1, r_2, \ldots, r_n\}$，其排序码分别为$K = \{k_1, k_2, \ldots, k_n\}$
- 将记录按排序码重排（排序的目的）
  - 形成新的有序序列$R' = \{r'_1, r'_2, \ldots, r'_n\}$
  - 相应排序码为$K' = \{k'_1, k'_2, \ldots, k'_n\}$
- 排序码顺序
  - $k'_1 \le k'_2 \le \ldots \le k'_n$，称为不减序
  - $k'_1 \ge k'_2 \ge \ldots \ge k'_n$，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：**

  待排序序列的逆转正好符合排序要求

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

逆序！

正序！

排序的稳定性

- Stable
  - 多个具有相同排序码的记录，在排序后的**相对次序保持不变**

- 示例
  - 34 12 34’ 08 96
  - 08 12 34 34’ 96

排序的稳定性

- Stable
  - 多个具有相同排序码的记录，在排序后的相对次序保持不变

- 示例
  - 34 12 34’ 08 96
  - 08 12 34’ 34 96

排序的稳定性

- 稳定的
  - 形式化证明
- 不稳定，反例说明即可
  - 34 12 34’ 08 96
  - 08 12 34’ 34 96

排序的基本操作

- 比较排序码大小
- 移动/交换元素（记录）

排序算法的衡量标准

- **时间代价**
  - 记录的比较和移动次数
  - 由于排序是频繁使用的一种运算，故其时间开销是算法好坏的最重要标志
- **空间代价**
- **算法本身的繁杂程度**

排序时间的评价

- 一般分**三种情况**来评估：
  - **最差**
  - **最佳**
  - **平均**

  以各自情况下的比较和移动次数来分析算法的时间代价

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?

主要内容

- **排序问题的基本概念**
- **排序方法**
  - 插入排序（Shell排序）
  - 选择排序（堆排序）
  - 交换排序
  - 归并排序
  - 分配排序和索引排序
- **排序算法的时间代价**

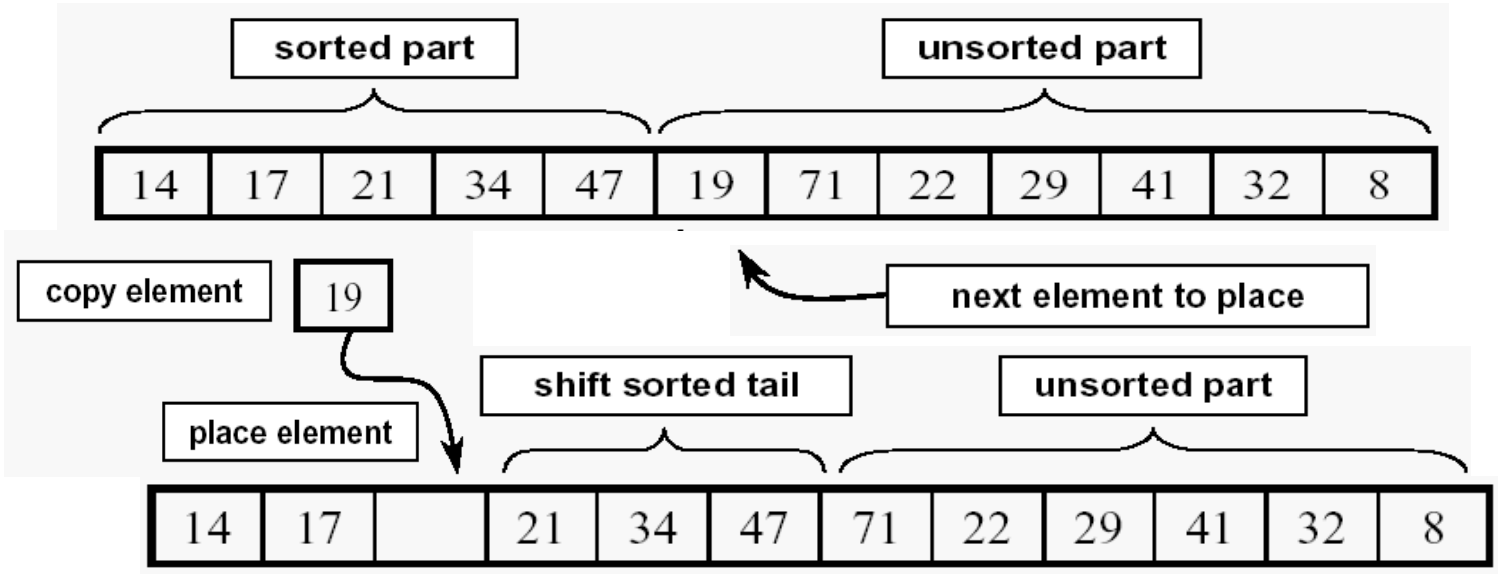
插入排序

- 直接插入排序
- 二分插入排序
- Shell排序

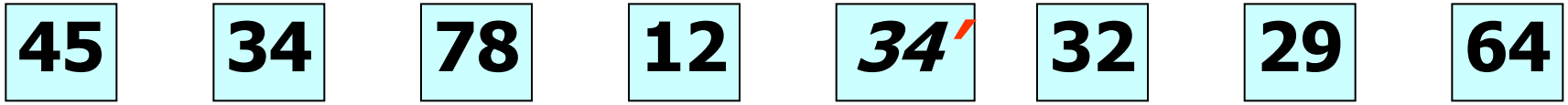

Welcome to BP's Bridge Website

插入排序

- **基本思想**
  - 逐个处理待排序的记录：每步将一个待排序的元素按其排序码大小插入到前面已排序表中的适当位置，直到全部插入完为止



插入排序示例



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];
        int j = i-1;                    // 从 i 开始往前寻找记录 i 的正确位置
        // 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;
    }
}
```

插入排序算法分析

- 稳定性
  - 稳定
- 空间代价：Θ(1)
- 时间代价：
  - 最佳情况：n-1次比较，2(n-1)次移动，Θ(n)
  - 最差情况：Θ(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：Θ(n^2)

插入排序的平均比较次数

- 初始位置为 k 的元素，假定其插入的位置为 j，而 j 的取值只能为在区间 1.. k；找到最终的位置 j 之前需进行 k-j+1 次比较；等概率情况下，放置第 k 个元素的平均比较次数为：

$$\frac{1}{K}\sum_{j=1}^{K}(K-j+1)=\frac{1}{K}\left[K^2-\frac{K(K+1)}{2}+K\right]=\frac{K+1}{2}$$

- 故，对 N 个元素的集合进行插入排序，平均总代价为：

$$\sum_{k=2}^{N}\left(\frac{K+1}{2}\right)=\sum_{k=1}^{N-1}\left(\frac{K+2}{2}\right)=\frac{1}{2}\left[\frac{(N+1)(N+2)}{2}\right]=\frac{1}{4}N^2+\frac{3}{4}N-1$$

插入排序的平均移动次数

- 与比较类似，区别在于：第 k 个元素的最终放置位置 j 在[1..k-1]之间，故移动次数为 k-j+2
  - 特例为元素不移动（仍在第 k个位置）

同比较的处理方式，可得到平均总移动次数为：

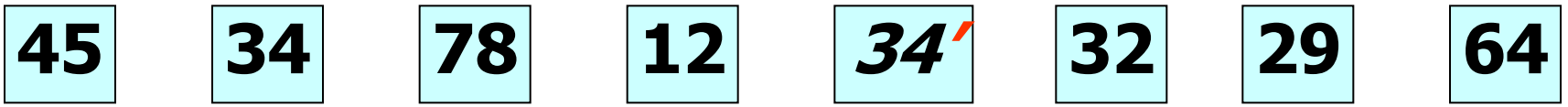
$$\sum_{k=2}^{N}\left(\frac{K+3}{2}-\frac{2}{K}\right)<\sum_{k=1}^{N-1}\left(\frac{K+4}{2}\right)=\frac{1}{4}N^2+\frac{7}{4}N+3$$

优化的插入排序算法

- 能否优化?
- 如何改进?

发现逆序对，交换?

- 依次对第i个记录进行排序
- 对记录Array[i]
  - 插入到有序区间 Array[0..i-1] 的合适位置



二分插入排序

- **算法思想**
  - 在插入第i个记录时，前面的记录已有序
  - 可用二分法查找第i个记录的插入位置
    - 须采用顺序存储方式
    - 比较次数与待排序序列初始状态无关，仅依赖于元素的个数

二分插入排序分析

- 稳定
- 空间代价：$\Theta(1)$
- 时间代价：
  1. 插入每个记录需要 $\Theta(\log i)$ 次比较
  2. 最多移动 $i+1$ 次， 最少 2 次（移动临时记录）
  3. ∴
     - 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 $\Theta(n^2)$

Shell排序

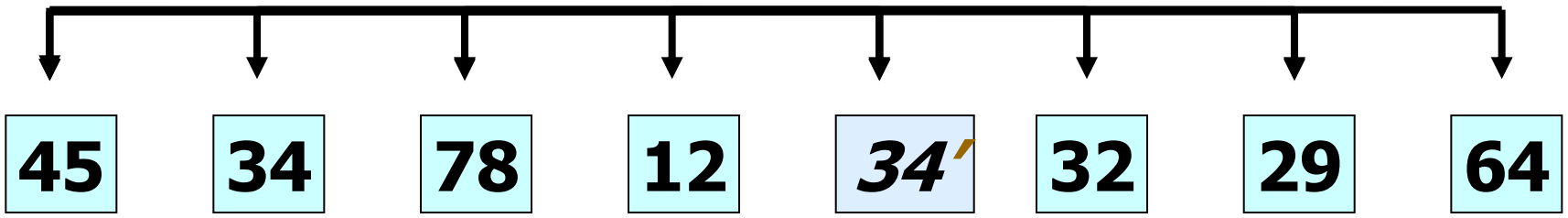
- **直接插入排序的两个特点：**
  1. **最佳**情况（序列本身已基本有序）下时间代价为Θ(n)
  2. 适用于 短序列

Shell排序**有效利用了直接插入排序**的这**两个特点**

Shell排序

- 由Donald Shell于1959年提出而得名，又称**缩小增量排序**（Diminishing Increment Sorting）
- **基本思想：**
  - 先将整个待排记录序列**分割**成若干个小的**子序列**，在这些小序列内分别进行**直接插入排序**；
  - 逐渐**扩大小序列的规模**，**减少**小序列**个数**，使得待排序序列逐渐处于**更为有序**的状态；
  - 待整个序列中的记录“基本有序”时，再对全体记录进行一次扫尾**直接插入排序**，从而完成整个的排序
- **特点：子序列不是简单的“逐段分割”，而是由相隔某个增量的记录构成**

Shell排序示例



增量除以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

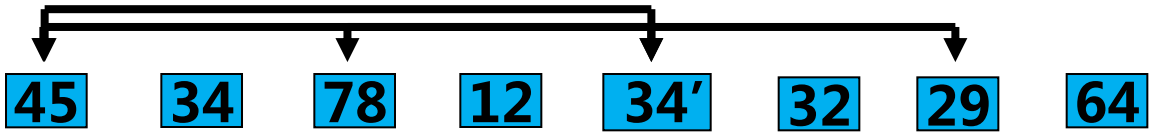
```
template <class Record>                          // 参数delta表示当前的增量
void ModInsSort(Record Array[], int n, int delta) {
   int i, j;
   for (i = delta; i < n; i += delta)            // 第i个记录找插入位置
          //  j 以 delta 为步长向前寻找逆置对进行调整
          for (j = i; j >= delta; j -= delta)  {
             if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

Shell排序算法分析

- **稳定性**
  - **不稳定**
- 空间代价
  - $\Theta(1)$
- 时间代价
  - $\Theta(n^2)$， 增量每次除以2递减
  - 可接近$\Theta(n)$，若选择适当的增量序列

Shell 排序选择增量序列

- **增量每次除以2递减** 时，效率仍为 $\Theta(n^2)$
- **原因**：选取的**增量** **相互间并不互质**
  - 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
  - 前轮循环中这些子序列可能都已相互比较过，导致处理效率不高



Shell 排序选择增量序列

- Hibbard增量序列
  - $\{2^k-1，2^{k-1}-1，…，7，3，1\}$
- 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进一步降低时间代价

Shell 排序最佳情况

- 呈 2^p3^q 形式的一系列整数：
  - 1, 2, 3, 4, 6, 8, 9, 12, ……
- $\Theta(n \log n)$

思考

- 插入排序的变种
  - 发现逆序对直接交换
- Shell 排序中增量作用是什么?
- Shell 排序的每一轮子序列排序， 可以用其他方法吗?

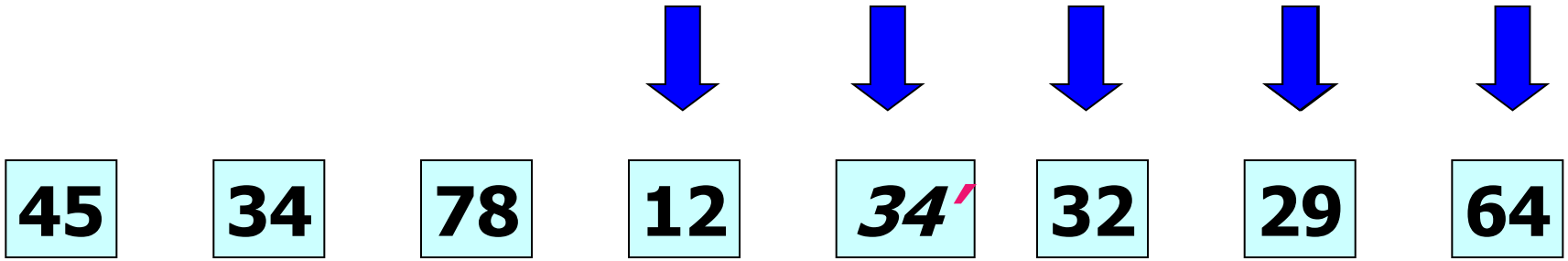
选择排序

- 算法思想
  - 找出尚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： 为每个位置选择合适的记录
- 方法
  - 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
    - 树形选择排序
    - 堆排序：基于最大值堆来实现，效率更高
  - 外排序
    - 置换选择排序
    - 赢者树、败方树

直接选择排序

- **步骤**
  - 首先在所有元素中选出排序码最小的元素，与第一个元素交换；
  - 然后在其余的元素中再选出排序码最小的元素与第二个元素交换；
  - 以此类推，直到所有元素排好序
- **主要操作**
  - **比较**

直接选择排序示例



直接选择排序

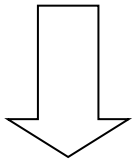
```
template <class Record>
void SelectSort(Record Array[], int n) {
// 依次选出第i小的记录，即剩余记录中最小的那个
    for (int i=0; i<n-1; i++) {
        // 首先假设记录i就是最小的
        int Smallest = i;
        // 开始向后扫描所有剩余记录
        for (int j=i+1;j<n; j++)
            // 如果发现更小的记录，记录它的位置
        if (Array[j] < Array[Smallest])
        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

- 稳定性
  - 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：$\Theta(n^2)$

选择排序的关键

- 主要操作是**比较**
  - 提高其速度必须减少比较的次数
  - 可否利用以前的比较结果来提高速度?



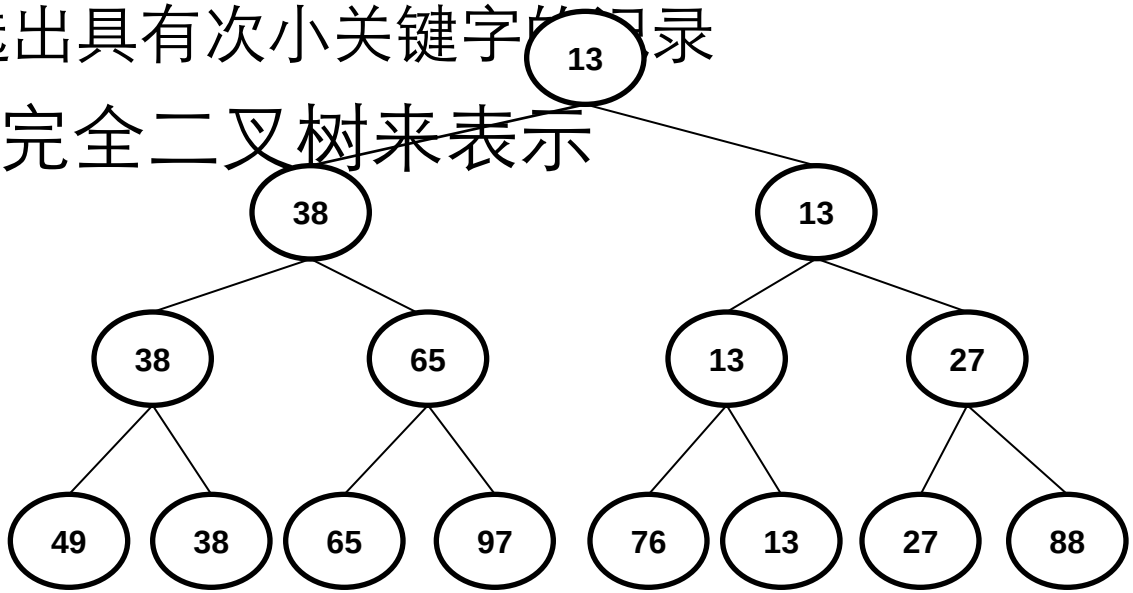
树形选择排序 (Tree Selection Sort)

又称 锦标赛排序(Tournament Sort)

树形选择排序

- 基本思想
  - 首先对n个记录的关键码进行两两比较
  - 然后在⌈n/2⌉个较小者之间再进行两两比较
  - 如此重复直到选出最小关键字的记录为止
  - 重复上述比较，选出具有次小关键字的记录

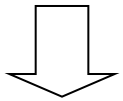
该过程可以用一个完全二叉树来表示



树形选择排序算法分析

- 时间代价
  - O(n *log* n)
  - 缺点：冗余比较（和“最大值”比较多次）
- 空间代价
  - 使用了较多的辅助空间，增加*n-1*个结点保存比较结果

优化可能？



YES！

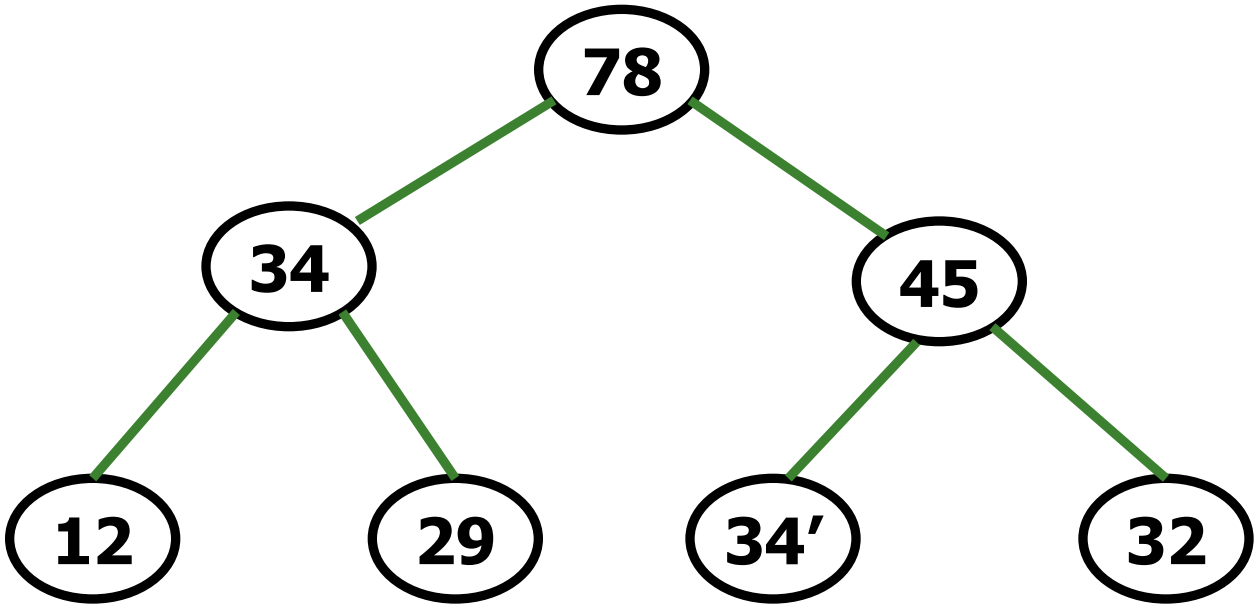
堆排序

- 1964年由Williams提出
  - Algorithm 232 (HEAPSORT). Communications of ACM, 7:347-348, 1964
- 引入“**堆**”作为中间数据结构，一种新的算法设计技巧
  - **利用数据结构来管理算法执行过程中的信息**
  - **步骤**
    - 先将一组待排序数据建成**堆**
    - 不断删除堆顶（并进行维护以保持堆的特性）
    - 直到堆成空为止

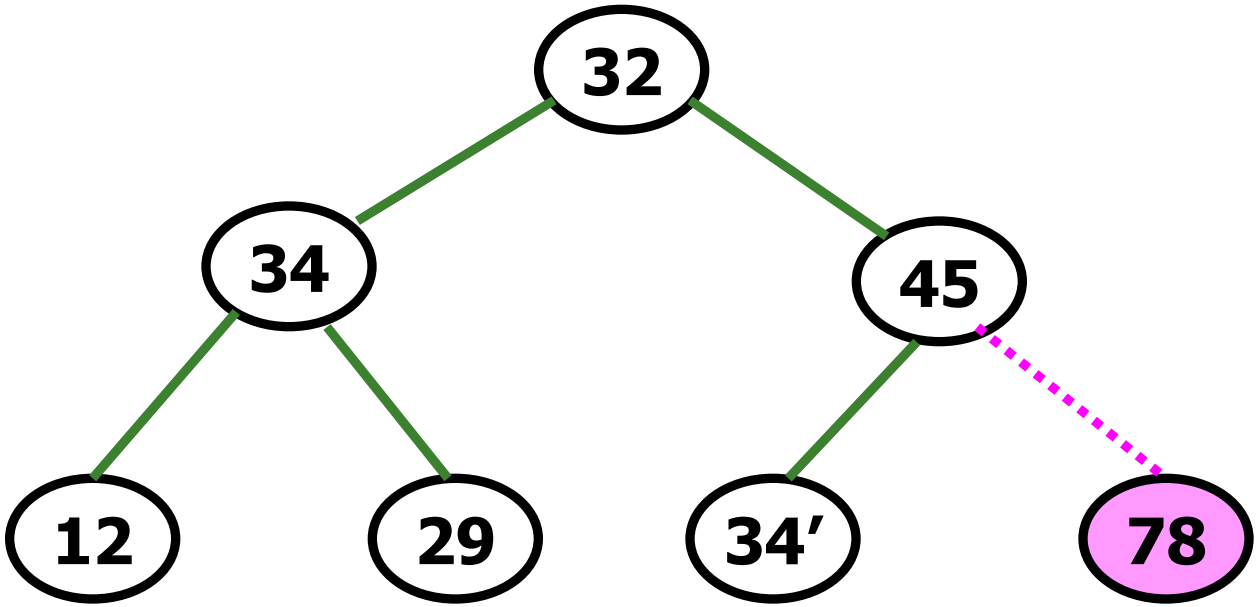
堆排序

- **需解决的两个问题：**
  1. 如何将一个无序序列建成堆
  2. 如何在输出堆顶元素后，将剩余元素调整为一个新的堆（满足堆的性质）
- **解决方案的关键**
  1. 筛选(siftdown/siftup)操作
  2. 从一个无序序列建堆的过程就是一个反复筛选的过程

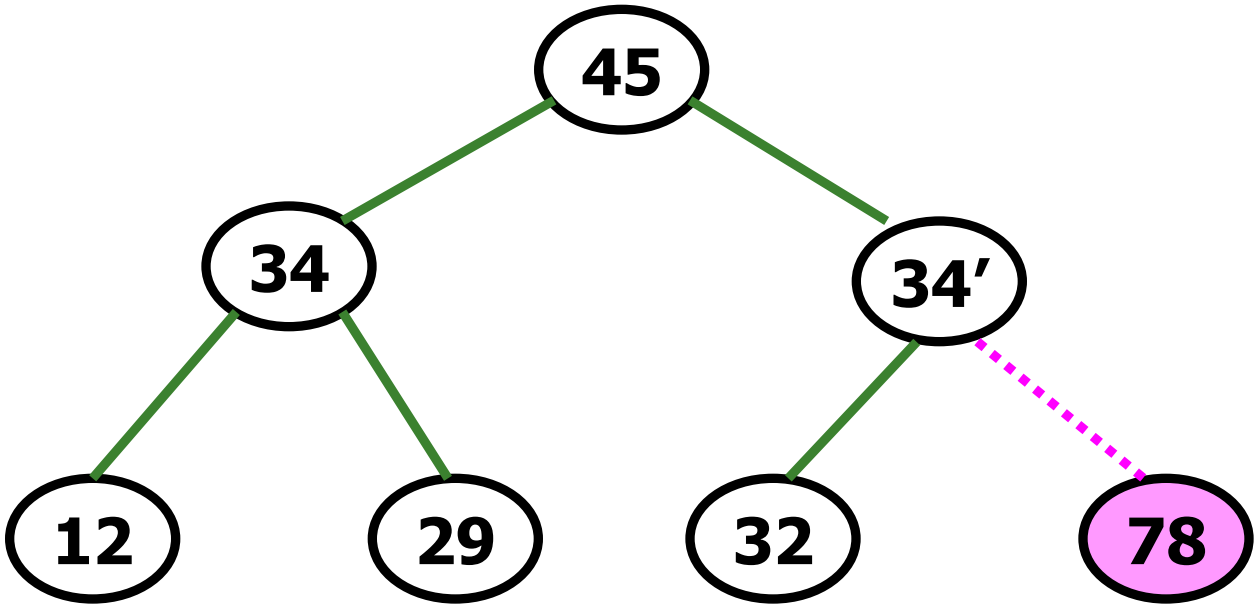
最大值堆排序过程示例



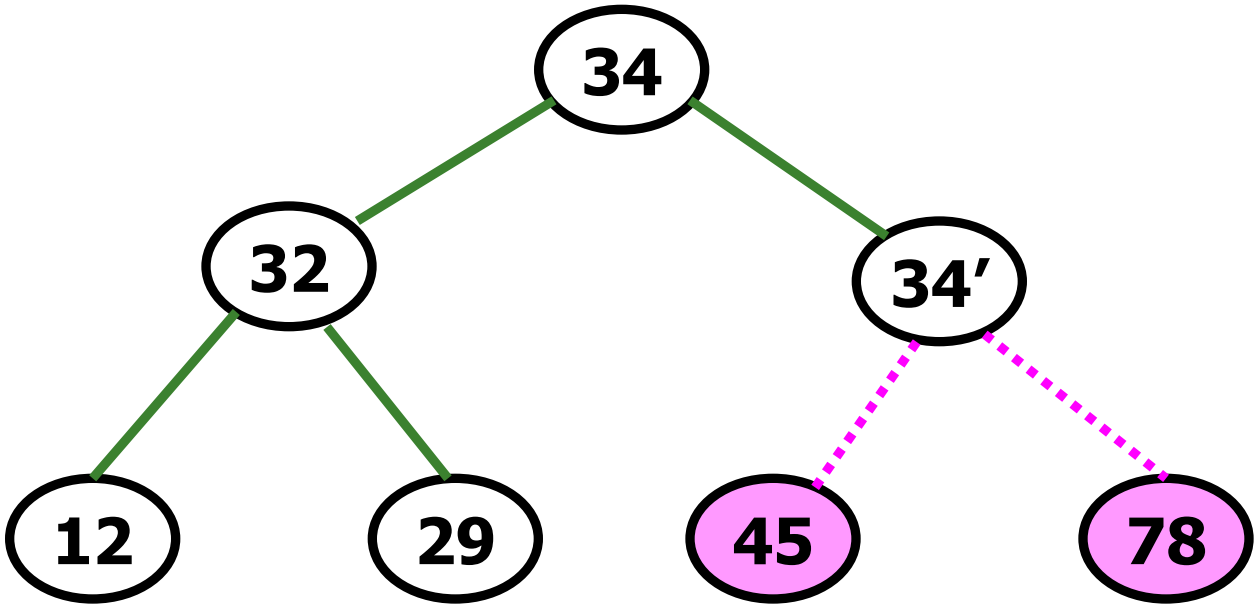
最大值堆排序过程示例



最大值堆排序过程示例



最大值堆排序过程示例



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();
}
```

堆排序分析

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

思考

- 直接选择排序为什么不稳定？如何修改可使其稳定?
- 请改写堆排序算法，发现逆序对直接交换

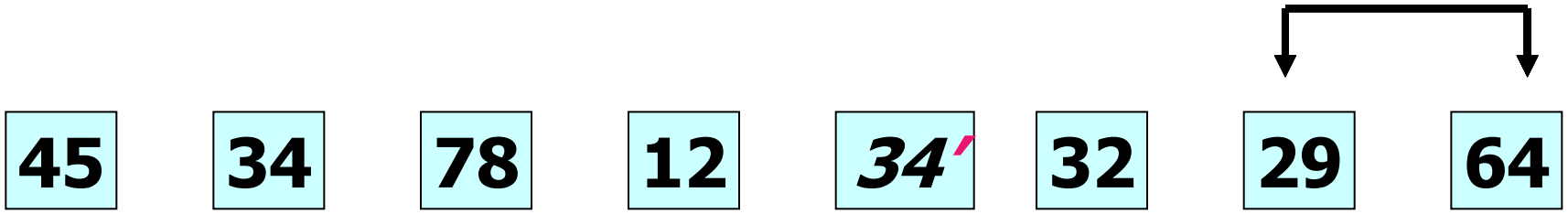
交换排序

- **基本思想**
  - 两两比较待排序记录的排序码，交换不满足排序要求的偶对
- **常用方法**
  - 冒泡排序（bubble sort）
  - 快速排序（quick sort）

冒泡排序

- **算法思想**
  - 比较相邻记录，若不满足排序要求，就**交换**相邻记录，直到所有的记录都已经排好序
- **优化**：检查每次冒泡过程中**是否发生过交换**，若**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
- 也称**起泡排序**

冒泡排序示例



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;
        }
    }
}
```

冒泡排序分析

- 稳定性
  - **稳定**
- 空间代价
  - $\Theta(1)$
- 时间代价
  - 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 最多为$\Theta(n^2)$，最少为0，平均为$\Theta(n^2)$
    - 最大、平均时间代价均 $\Theta(n^2)$
    - 最小时间代价 $\Theta(n)$

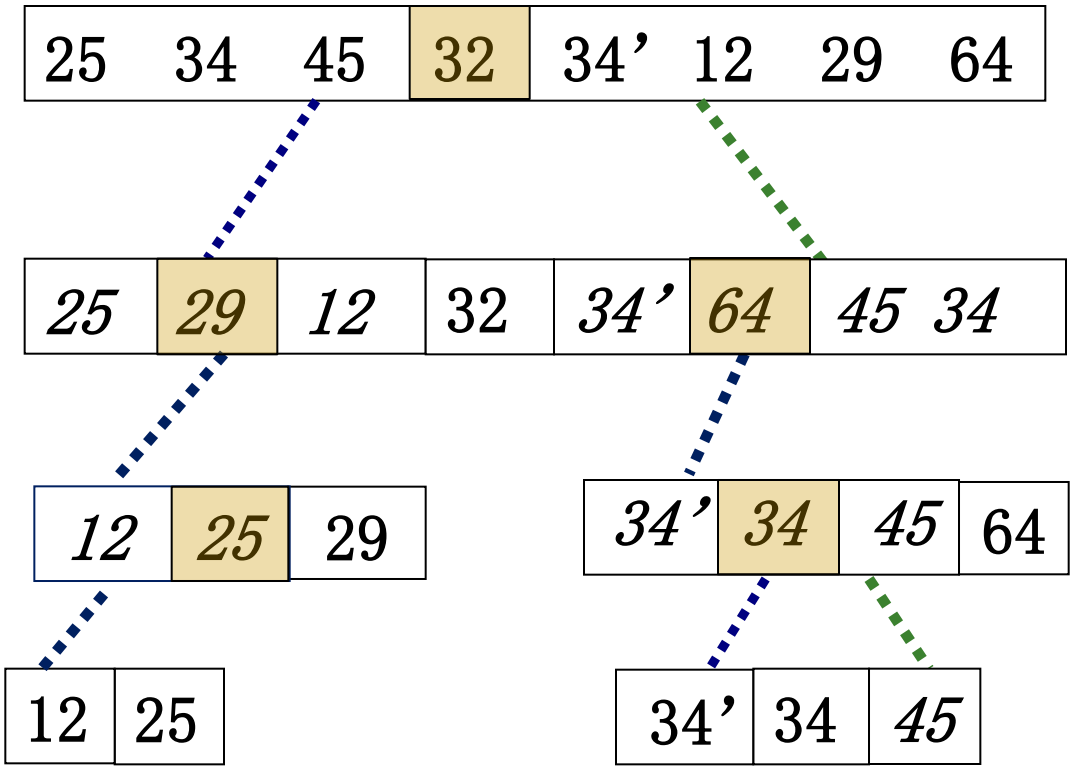
快速排序

- Top 10 Algorithms of the 20th Century
  - 1962 by Tony Hoare（Elliot Brothers Ltd in London)
- **算法思想**
  - 采用**分治** (Divide-and-conquer) 方法，将待排元素分割成**独立的两部分**，其中一部分元素的排序码均比另一部分元素的排序码小，则可分别对这两部分元素继续进行排序，以达到整个序列有序，即
    - 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

分治策略的基本思想

- **分治策略的实例**
  - BST查找、插入、删除算法
  - 快速排序、归并排序
  - 二分检索
- **基本步骤**
  - **分——划分子问题**
  - **治——求解子问题(子问题不重叠)**
  - **合——综合解**

快速排序的分治思想



最终排序结果：12 25 29 32 34’34 45 64

轴值选择

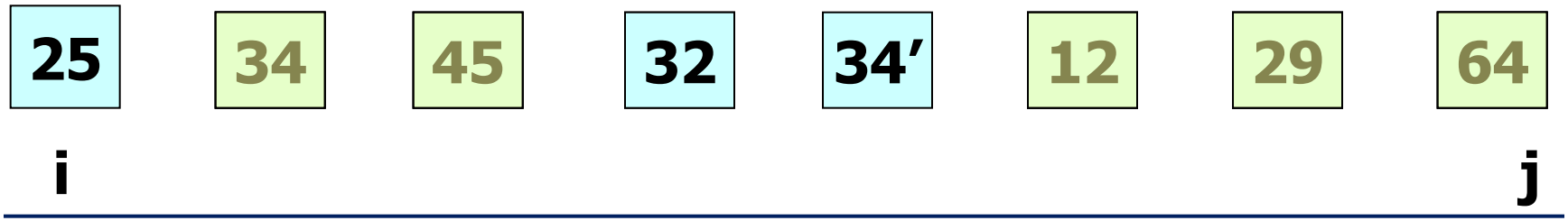
- **尽可能使L，R长度相等**
- **选择策略**
  - 选择最左边记录
  - 随机选择
  - 选择平均值
  - ……

分割过程（Partition）

- 整个快速排序的**关键**
- **轴值**位于**最终该待的位置**，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

一次分割过程

1. 选择并存储**轴值**
2. 最后一个元素放到轴值原本的位置
3. 初始化下标i、j，分别指向头、尾
4. 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
5. 重复上一步直到i==j，将轴值放到 i 的位置，完毕



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，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
            while (Array[l] <= TempRecord && r > l) // l 指针右移直到一个大于轴值的记录
                  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
            }
            // 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
}
```

快速排序时间代价

- 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 O(1)
- 分割时间为 cn
  - 分割后长度分别为 i 和 n-i-1
  - 子序列快速排序所需时间分别为 T(i) 和 T(n-1-i)
- 求解**递推方程**

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

快速排序最差情况

$$T(n) = T(n-1) + cn$$
$$T(n-1) = T(n-2) + c(n-1)$$
$$T(n-2) = T(n-3) + c(n-2)$$
$$\cdots$$
$$T(2) = T(1) + c(2)$$

- 总的时间代价为:

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

快速排序最佳情况

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

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

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

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

$$\cdots$$

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

log n 次分割

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

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

快速排序平均情况：等概率分割

- *T(i)* 和 *T(n-1-i)* 的平均值均为

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

- 代入方程

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

得出

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

快速排序平均情况：等概率分割

- 上式乘以n

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

  - 带入n-1

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

- 上述二式相减

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

- 上式两边同时除以 n(n+1)，并忽略常数系数

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

快速排序平均情况：等概率分割

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

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

$$\cdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

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

快速排序分析

- 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： $\Theta(\log n)$

快速排序分析

- 稳定性
  - 不稳定
- 可能优化：
  - 轴值选择
    - RQS
  - 小子串不递归
  - 消除递归

优化的快速排序

```
#define THRESHOLD 28
template <class Record>
void ModQuickSort(Record Array[], int left, int right) {
    if (right-left+1 > THRESHOLD)  {                    // 长子串处理
        int pivot = SelectPivot(left, right);           // 选择轴值
        swap(Array, pivot, right);                      // 将轴值放在数组末端
        pivot = Partition(Array, left, right);          // 分割
        ModQuickSort(Array, left, pivot-1);             // 处理左
        ModQuickSort(Array, pivot+1, right);            // 处理右
    }
}
```

优化的快速排序

```
template <class Record>
void QuickSort(Record *Array, int n)  {
    // 调用优化的递归快排，不处理小子串
    ModQuickSort(Array, 0, n-1);
    // 最后这个序列进行扫尾插入排序
    InsertSort(Array, n);
}
```

思考

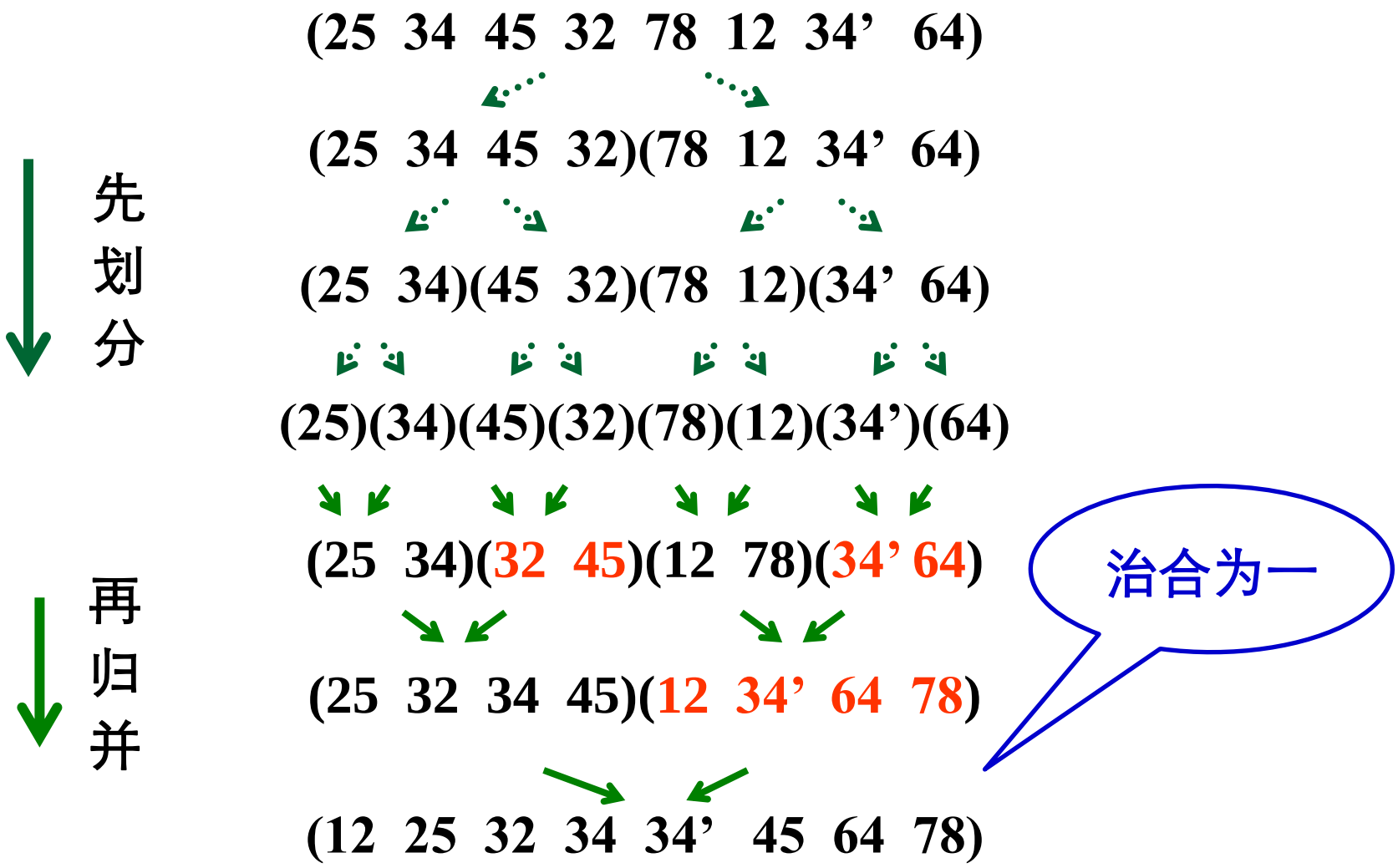
- 冒泡排序和直接选择排序哪个更优
- 快速排序为什么不稳定
- 快速排序可能的优化
  - 轴值选择 RQS
  - 小子串不递归 （阈值 28？ ）
  - 消除递归（用栈，队列？）

归并排序 （MergeSort）

- 将两个或两个以上的**有序表**（集合）合并成一个有序表

- 算法思想
  - **划分**：将原始待排序序列简单划分为两个或多个子序列
  - **分治**：分别对每个子序列递归排序
  - **综合**：将排好序的子序列合并为一个有序序列，即归并过程

归并思想


(25 34 45 32 78 12 34’ 64)
(25 34 45 32)(78 12 34’ 64)
(25 34)(45 32)(78 12)(34’ 64)
(25)(34)(45)(32)(78)(12)(34’)(64)
(25 34)(32 45)(12 78)(34’ 64)
(25 32 34 45)(12 34’ 64 78)
(12 25 32 34 34’ 45 64 78)
先划分
再归并
治合为一

归并排序

- 要求待排序集合已部分排序
- 合并时只要比较各子序列首元素的排序码，最小者即为排序后序列的首元素的排序码，取出该元素，继续比较各子序列的第1个元素，便可找出排序后序列的第2个元素，如此继续，只要经过一遍扫描，即可得到排序结果

利用归并的思想可实现排序

- 两路
- 多路

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
    }
}
```

归并函数

```
// 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;
      for (j = left; j <= right; j++)                    // 将数组暂存入临时数组

            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
    while (index1 <= middle && index2 <= right) {
            if (TempArray[index1] <= TempArray[index2])
                  Array[i++] = TempArray[index1++];      // 取较小者插入合并数组中
            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++];
}
```

归并排序时间代价

- 由三部分组成
  - 划分时间
  - 两个子序列的排序时间
  - 归并时间，n

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

  - T(1) = 1
  - 归并排序总时间代价为 Θ(nlog n)

归并排序分析

- 归并排序的**比较次数**很接近**理论上的最优值**
  - N个元素的序列，归并排序的比较次数为

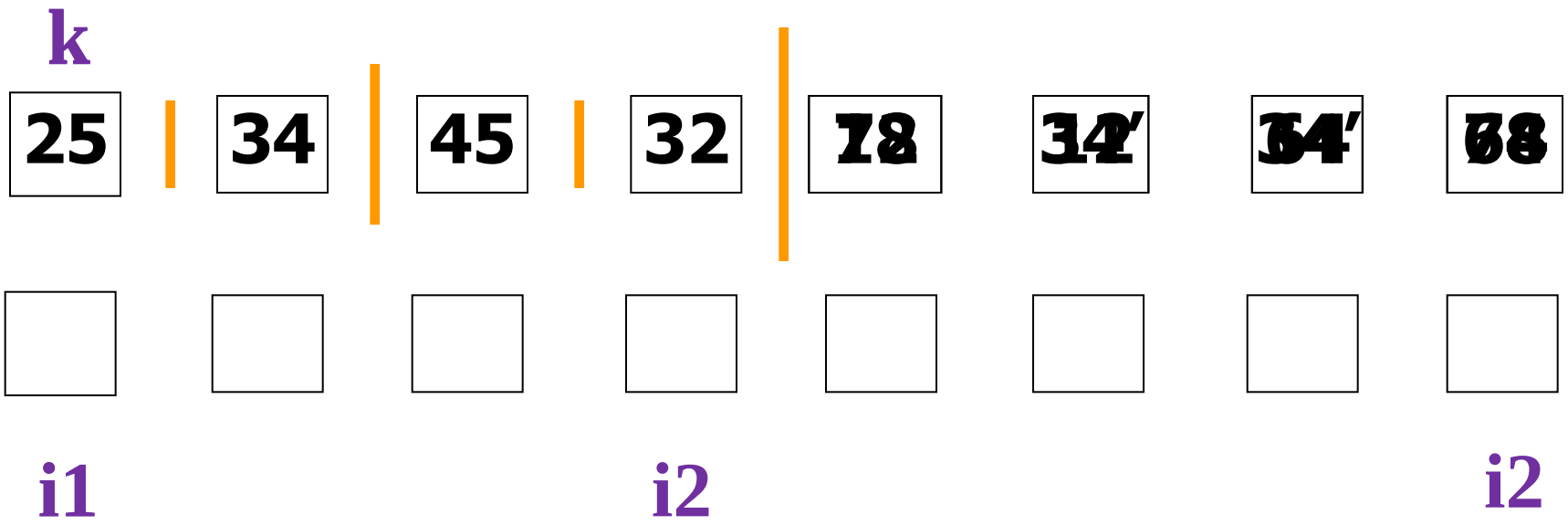
$$O(N\log N)-1.1583N+1$$

归并排序分析

- 空间代价：Θ(n)
- 时间代价：Θ(nlog 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Θ(nlog n)
- 稳定性
  - **稳定**
- 优化的可能
  - 同快速排序的优化，对基本有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，简化边界判断

R. Sedgewick优化归并

R. Sedgewick优化归并



优化的归并排序

```
#define THRESHOLD 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

```
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--];
}
```

思考

- 普通归并和 Sedgewick 算法均稳定吗?
- 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

排序算法小结

- 特点?
  - 复杂度?
  - 核心操作?

- 有无不通过比较就可实现排序的方法？即，可否设计某些算法，使其不用对待排序元素进行一一比较?
- Such algorithms require special assumptions regarding the type and/or range of the key values to be sorted

其他排序方法

- 日常实例
  - 图书馆中书目管理
    - 作者姓名分类，首字母，从左到右
  - 邮局分发邮件
    - ZIP Code 具有同样的长度
  - 整数
    - [23, 123, 234, 567, 3] => [123, 23, 234, 3, 567]
    - [023,123, 234, 567, 003] => [003, 023,123, 234, 567]

分配排序和基数排序

- 不必记录间的两两比较
- 需事先知道记录序列的一些具体情况

- 基本思想
  - 将排序码**分解成**若干部分，通过对各个部分排序码的分别排序，最终达到对整个排序码的排序
- 主要介绍两类
  - 桶式排序
  - 基数排序

桶式排序

- 事先知道序列中的记录都处于某个小区间段[0，m)内
- 将具有相同值的记录分配到同一个桶中，然后按照编号依次从桶中取出记录，组成一个有序序列

桶式排序示例

count[i] = count[i-1] + count [i];

待排数组: 7 3 8 9 6 1 8’ 1’ 2

	0	1	2	3	4	5	6	7	8	9
每个桶count：	0	2	1	1	0	0	1	1	2	1

＋＋＋＋

	0	1	2	3	4	5	6	7	8	9
前若干桶的累计count_->后继起始下标：	0	2	3	4	4	4	5	6	8	9

桶排序示例

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

后继起始下标：

0	0	2	4	4	4	5	6	7	9

收集：

0	1	2	3	4	5	6	7	8
1	1’	2	3	6	7	8	8’	9

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];
}
```

桶式排序分析

- 数组长度为n，所有记录区间[0,m)上

- 时间代价
  - 统计计数时间：$\Theta(n+m)$，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
- 空间代价
  - m个计数器，长度为n的临时数组，$\Theta(m+n)$

- 稳定