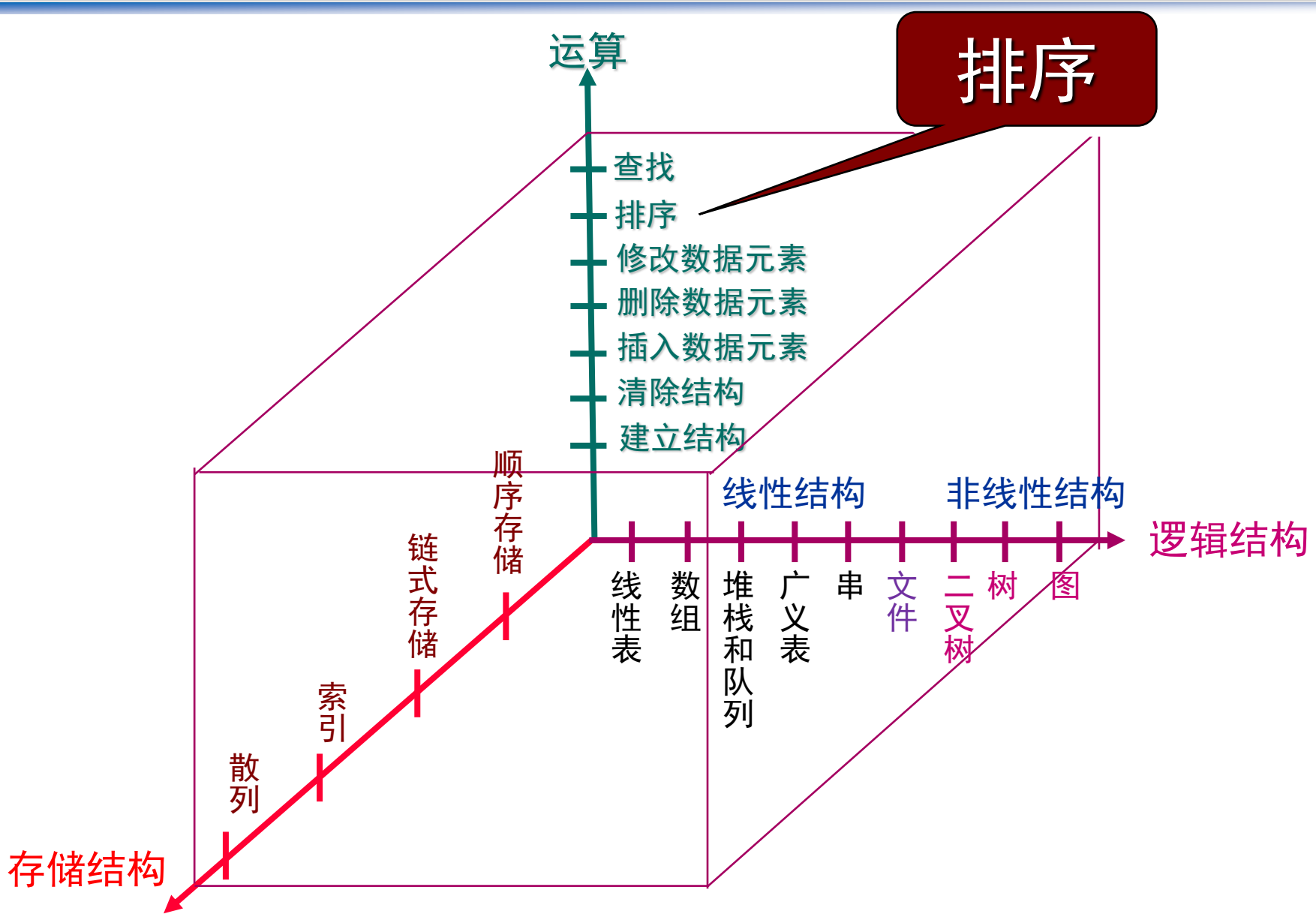
数据结构与程序设计
(Data Structure and Programming)

排序
(Sort)

北航计算机学院 林学练

数据结构的基本问题空间



本章内容

8.1 排序的基本概念

8.2 插入排序法

8.3 选择排序法

8.4 泡排序法

8.5 堆排序法

8.6 快速排序法

8.7 归并排序法

8.8 谢尔排序法

8.9 非比较排序法*

Sort Benchmark Home Page

New: We are happy to announce the 2013 winners listed below. The new, 2013 records are listed in green. This was a competitive year. For each new winning entry there was at least one entry from a competing group. Thank you to all the 2013 entrants!

Background

Until 2007, the sort benchmarks were primary defined, sponsored and administered by Jim Gray. Following Jim's disappearance at sea in January 2007, the sort benchmarks have been continued by a committee of past colleagues and sort benchmark winners. The Sort Benchmark committee members include:

- Chris Nyberg of Ordinal Technology
- Mehul Shah of Amiato
- Naga Govindaraju of Microsoft

Top Results

	Daytona	Indy
Gray	2013, 1.42 TB/min **Hadoop** 102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) Thomas Graves Yahoo! Inc.	2013, 1.42 TB/min **Hadoop** 102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) Thomas Graves Yahoo! Inc.
Penny	2011, 286 GB **psort** 2.7 Ghz AMD Sempron, 4 GB RAM, 5x320 GB 7200 RPM Samsung SpinPoint F4 HD332GJ, Linux Paolo Bertasi, Federica Bogo, Marco Bressan and Enoch Peserico Univ. Padova, Italy	2011, 334 GB **psort** 2.7 Ghz AMD Sempron, 4 GB RAM, 5x320 GB 7200 RPM Samsung SpinPoint F4 HD332GJ, Linux Paolo Bertasi, Federica Bogo, Marco Bressan and Enoch Peserico Univ. Padova, Italy
Minute	2012, 1,401 GB **Flat Datacenter Storage** 256 heterogeneous nodes, 1033 disks Johnson Apacible, Rich Draves, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, Ed Nightingale, Reuben Olinksy, Yutaka Suzue Microsoft Research	2012, 1,470 GB **Flat Datacenter Storage** 256 heterogeneous nodes, 1033 disks Johnson Apacible, Rich Draves, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, Ed Nightingale, Reuben Olinksy, Yutaka Suzue Microsoft Research
Joule 10^8 recs	2013, 889 Joules **NTOSort** 112,545 records sorted / joule Lenovo X220, 2.8 Ghz Intel i5-2640M, 16GB RAM, Nsort, Windows 8, 1 OCZ 120GB mSATA Nocti SSD, 2 Samsung 840 Pro 256GB SSDs Andreas Ebert Microsoft	2013, 889 Joules **NTOSort** 112,545 records sorted / joule Lenovo X220, 2.8 Ghz Intel i5-2640M, 16GB RAM, Nsort, Windows 8, 1 OCZ 120GB mSATA Nocti SSD, 2 Samsung 840 Pro 256GB SSDs Andreas Ebert Microsoft
Joule 10^9 recs	2013, 12,092 Joules **NTOSort** 82,697 records sorted / joule Lenovo X220, 2.8 Ghz Intel i5-2640M, 16GB RAM, Nsort, Windows 8, 1 OCZ 120GB mSATA Nocti SSD, 2 Samsung 840 Pro 256GB SSDs	2013, 12,092 Joules **NTOSort** 82,697 records sorted / joule Lenovo X220, 2.8 Ghz Intel i5-2640M, 16GB RAM, Nsort, Windows 8, 1 OCZ 120GB mSATA Nocti SSD, 2 Samsung 840 Pro 256GB SSDs

8.1 排序的基本概念

一、排序的定义

将一个按值任意的数据元素序列转换为一个按值有序的数据元素序列。

对于含有n个记录的序列$\{ R_1, R_2, \ldots, R_n \}$，对应的关键字序列为$\{k_1, k_2, \ldots, k_n \}$，确定一种置换关系

$$\sigma(1), \sigma(2), \ldots, \sigma(n)$$

使得相应文件成为按关键字值有序的文件

$$\{ R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)} \}$$

其中关键字序列满足：

$$k_{\sigma(1)} \le k_{\sigma(2)} \le \ldots \le k_{\sigma(n)} \text{ 或 } k_{\sigma(1)} \ge k_{\sigma(2)} \ge \ldots \ge k_{\sigma(n)}$$

这一过程称为 **排序 (Sort)** 。

基于文件提出的概念

二、排序的功能

1. 能够将记录按关键字值任意排列的数据文件转换为一个记录按关键字值有序排列的数据文件。

或者

能够将一个按值任意排列的数据元素序列转换为一 个按值有序排列的数据元素序列。

2. 能够提高查找的时间效率。

三、排序的分类

按存储类型：

内排序 —— 参加排序的数据量不大，以致于能够一次将参加排序的数据全部装入内存实现排序。

外排序 —— 当参加排序的数据量很大，以致于无法一次将参加排序的数据全部装入内存，排序过程中需要不断地在内存与外存之间交换数据。

按是否进行关键字比较：

基于比较的排序 ——包括插入排序法、选择排序法、泡排序法、谢尔排序法、堆排序法、归并排序法、快速排序法

非比较的排序 ——“分配式排序”(distribution sort)，包括计数排序、桶排序、基数排序

四、排序算法的性能

排序操作的性能评价主要包括：

① 时间性能

② 空间性能

③ 稳定性

1.时间性能 —— 排序过程中元素之间的**比较次数**与元素的**移动次数**。

注：讨论各种排序方法的**时间复杂度**时一般按照最差情况下所需要的比较次数来进行。

2.空间性能 —— 除了存放参加排序的元素之外，排序过程中所需要的其他**辅助**空间。

3.排序稳定性 ——对于值相同的两个元素，排序前后的先后次序不变，则称该方法为**稳定性排序方法**；否则，称为**非稳定性排序方法**。

注：在所有可能的输入实例中，只要有一个实例使得该排序方法不满足稳定性要求，该排序方法就是非稳定的！

趟 ——将具有n个数据元素(关键字)的序列转换为一个按照值的大小从小到大排列的序列通常要经过若干 **趟** (Pass)。

约定

1. 只针对一个数据元素(关键字)序列讨论排序方法。

2. 假设序列中具有n个数据元素(关键字)。

$$(k_1, k_2, k_3, \cdots, k_{n-1}, k_n)$$

存放于数组元素 K[1],K[2], …, K[n]中

3. 排序结果按照数据元素(关键字)值的大小从小到大排列。

本章内容

8.1 排序的基本概念

简单排序
- 8.2 插入排序法 （已在线性表中讲解）
- 8.3 选择排序法
- 8.4 泡排序法

改进算法
- 8.5 堆排序法 （已在二叉树中讲解）
- **8.6 快速排序法**
- **8.7 归并排序法**
- **8.8 谢尔排序法**

8.9 非比较排序法*

8.2 插入(insert)排序法 （已在线性表中讲解）

第 i 趟排序将无序序列的第 i+1 个元素插入到一个长度为i、且已经按值有序的子序列($k_{i-1,1}$，$k_{i-1,2}$，…，$k_{i-1,i}$)的合适位置，得到一个长度为i+1、且仍然按值有序的子序列($k_{i,1}$，$k_{i,2}$，…，$k_{i,i+1}$)。

（1，4，8，12，6，11，7，…）

（1，4，6，8，12，11，7，…）

算法关键步骤示例

49 38 97 76 65 13 27 50

… (若干趟后)

（有序子序列） （无序子序列）

38 49 76 97 65 13 27 50

先查找后插入 或 边比较边移动/交换

38 49 65 76 97 13 27 50

一趟结束

从第2个元素开始

i：1→n-1

j：i-1→0

初 始:	49	38	97	76	65	13	27	50
第1趟:	38	49	97	76	65	13	27	50
第2趟:	38	49	97	76	65	13	27	50
第3趟:	38	49	76	97	65	13	27	50
第4趟:	38	49	65	76	97	13	27	50
第5趟:	13	38	49	65	76	97	27	50
第6趟:	13	27	38	49	65	76	97	50
第7趟:	13	27	38	49	50	65	76	97

一个完整的过程


```
void insertSort(keytype k[ ],int n){
    int i, j;
    keytype  temp;
    for(i=1;i<n; i++){
        temp=k[i];
        for(j=i-1; j>=0 && temp<k[j]; j--)
            k[j+1]=k[j];
        k[j+1]=temp;
    }
}
```

n−1趟排序

思考

1. 排序的时间效率与什么直接有关?

答案

主要与排序过程中元素之间的比较次数直接有关。

2. 若原始序列为一个**按值递增**的序列，则排序过程中一共要经过多少次元素之间的比较?

答案

由于每一趟排序只需要经过一次元素之间的比较就可以找到被插入元素的合适位置，因此，整个n-1趟排序一共要经过n-1次元素之间的比较。

3. 若原始序列为一个按值递减的序列，则排序过程中一共要经过多少次元素之间的比较?

答案

由于第i趟排序需要经过i次元素之间的比较才能找到被插入元素的合适位置,因此,整个n-1趟排序一共要经过

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

次元素之间的比较。

若以最坏的情况考虑，则插入排序算法的时间复杂度为 $O(n^2)$。

插入排序法是一种稳定排序方法。

插入排序算法分析

稳 定 性： 稳定

时间复杂度：

- 最佳情况：n-1次比较，0交换，O(n)
- 最差情况：比较和交换次数为$O(n^2)$
- 平均情况：$O(n^2)$

空间复杂度： O(1)

注：此处仅考虑辅助空间
实际上全量数据必须加载在内存

插入排序算法优化

用折半查找法
找到插入位置

13 38 49 65 76 97 27

依次右移一个位置

temp

27

13 27 38 49 65 76 97

可减少比较次数，
但不能减少移动次数

折半插入排序法

采用折半查找法确定插入位置

```
void  insertBSort(keytype k[ ], int n){
      int i, j, low, high, mid;
      keytype temp;
      for(i=1;i<n;i++ ){
          temp=k[i];
          low=0;
          high=i–1;
          while(low<=high){
                mid=(low+high)/2;
                if(temp<k[mid])
                      high=mid–1;
                else
                      low=mid+1;
          }
          for(j=i–1; j>=low; j––)
                k[j+1]=k[j];
          k[low]=temp;
      }
}
```

8.3 选择(select)排序法 (已在线性表中讲解)

核心思想

第 i 趟排序从未排序子序列(原始序列的后n−i+1个元素)中 **选择** 一个值最小的元素，将其置于子序列的最前面。

选择	35	97	38	27	65	13	80	75
交换	13	97	38	27	65	35	80	75
选择	13	97	38	27	65	35	80	75
交换	13	27	38	97	65	35	80	75

n−1趟

i：0→n-2

j：i→n-1

初 始:	49	97	38	76	65	13	27	50
第1趟:	13	97	38	76	65	49	27	50
第2趟:	13	27	38	76	65	49	97	50
第3趟:	13	27	38	76	65	49	97	50
第4趟:	13	27	38	49	65	76	97	50
第5趟:	13	27	38	49	50	76	97	65
第6趟:	13	27	38	49	50	65	97	76
第7趟:	13	27	38	49	50	65	76	97

一个完整的过程

```
void  selectSort(keytype k[ ],int n) {
      int i, j, d;
      keytype  temp;
      for(i=0;i<n−1;i++){
         /*寻找值最小的元素,并记录其位置*/
          d=i;
          for(j=i+1;j<n;j++)
               if(k[j]<k[d])
                    d=j;
          /* 交换最小值元素位置 */
          if(d!=i){
                temp=k[d] ;
                k[d]=k[i];
                k[i]=temp;
          }
      }
}
```

n-1趟排序

思考

若原始序列为一个按值**递增**的序列，则排序过程中一共要经过多少次元素之间的比较？

若原始序列为一个按值**递减**的序列，则排序过程中一共要经过多少次元素之间的比较？

答案： 无论原始序列为什么状态，第i趟排序都需要经过n−i次元素之间的比较，因此，整个排序过程中元素之间的比较次数为$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2$。

结论

选择排序法的元素之间的比较次数与原始序列中元素的分布状态无关。

时间复杂度为$O(n^2)$。

不稳定排序方法

选择排序算法分析

稳 定 性： 不稳定

时间代价：

- 比较次数：$O(n^2)$
- 交换次数：$n-1$
- 总时间代价：$O(n^2)$

空间代价： $O(1)$

注：此处仅考虑辅助空间

选择最小（最大）元素的效率比较低

8.4 冒泡(bubble)排序法 (已在线性表中讲解)

值大的元素往后“沉”
值小的元素向前“浮”

核心思想

第i趟排序对序列的前n−i+1个元素从第一个元素开始依次作如下操作：相邻的两个元素比较大小，若前者大于后者，则两个元素交换位置，否则不交换位置。

效果

该n−i+1个元素中最大值元素移到该n−i+1个元素的最后。

i：n-1→1
j：0→i-1

或 i：0→n-2
j：n-1→i+1

初始	49	97	38	13	27	50	76	65
第1趟	49	38	13	27	50	76	65	97
第2趟	38	13	27	49	50	65	76	97
第3趟	13	27	38	49	50	65	76	97
第4趟	13	27	38	49	50	65	76	97

排序总趟数可以小于n-1

设置一标志

$$flag = \begin{cases} 0 & \text{某趟排序过程中无元素交换位置的动作} \\ 1 & \text{某趟排序过程中有元素交换位置的动作} \end{cases}$$

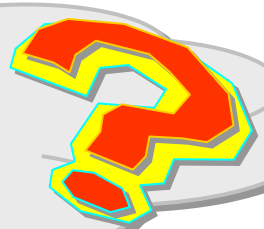
每一趟趟排序前置flag为0；若排序过程中出现元素交换动作，则置flag为1。

```
void  bubbleSort(keytype k[ ],int n){
      int i, j, flag=1;
      keytype temp;
      for(i=n-1; i>0 && flag==1; i--){
            flag=0;                          /* 每趟排序前标志flag置0 */
            for(j=0;j<i;j++)
                  if(k[j]>k[j+1]){
                        temp=k[j];
                        k[j]=k[j+1];
                        k[j+1]=temp;  /* 交换两个元素的位置 */
                        flag=1;              /* 标志flag置1 */
                  }
      }
}
```

泡排序法的排序趟数与原始序列中数据元素的排列有关，因此，排序的趟数为一个范围，即[1..n−1]。

什么情况下只要排序一趟

什么情况下要排序n−1趟



结论

泡排序方法比较适合于参加排序的序列的原始状态基本有序的情况

$O(n^2)$

泡排序法是稳定性排序方法。

（改进）冒泡排序算法分析

稳 定 性： 稳定

时间代价：

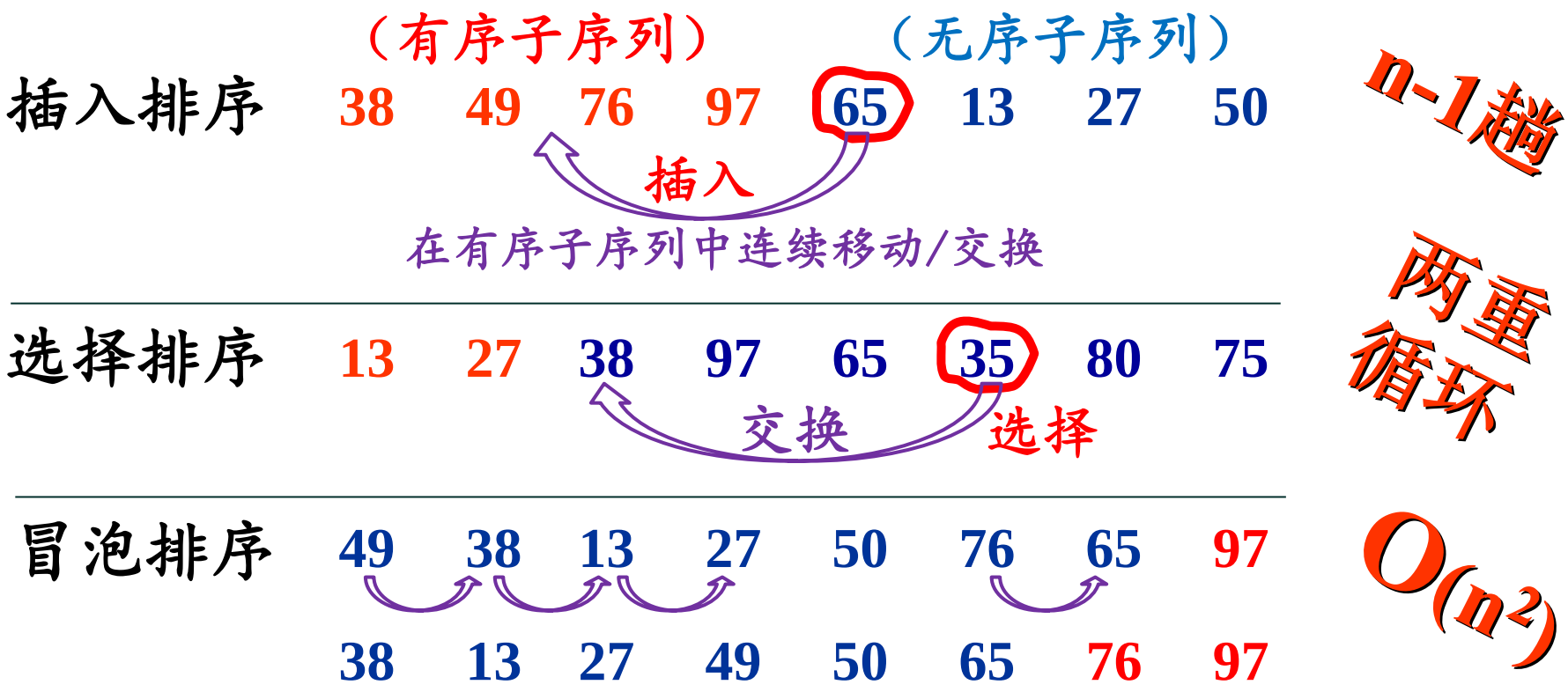
- 最小时间代价为O(n)，最佳情况下只运行第一轮循环
- 一般情况下为$O(n^2)$

空间代价： O(1)

注：此处仅考虑辅助空间

三种简单排序方法的直观比较

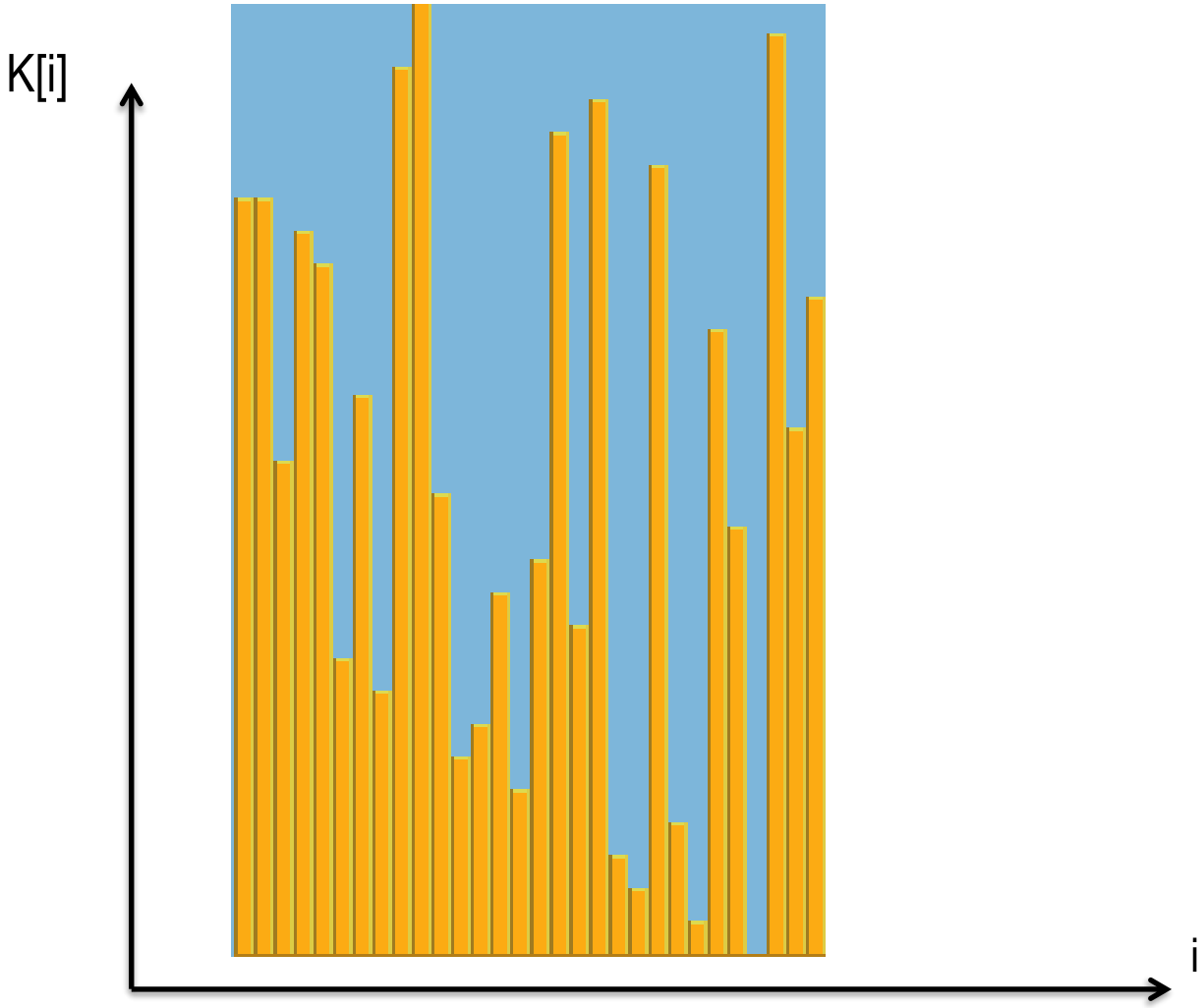
共性：每趟都从无序子序列中找一个元素，将其加入有序子序列



冒泡排序部分融合了“选择”和“插入”排序法：

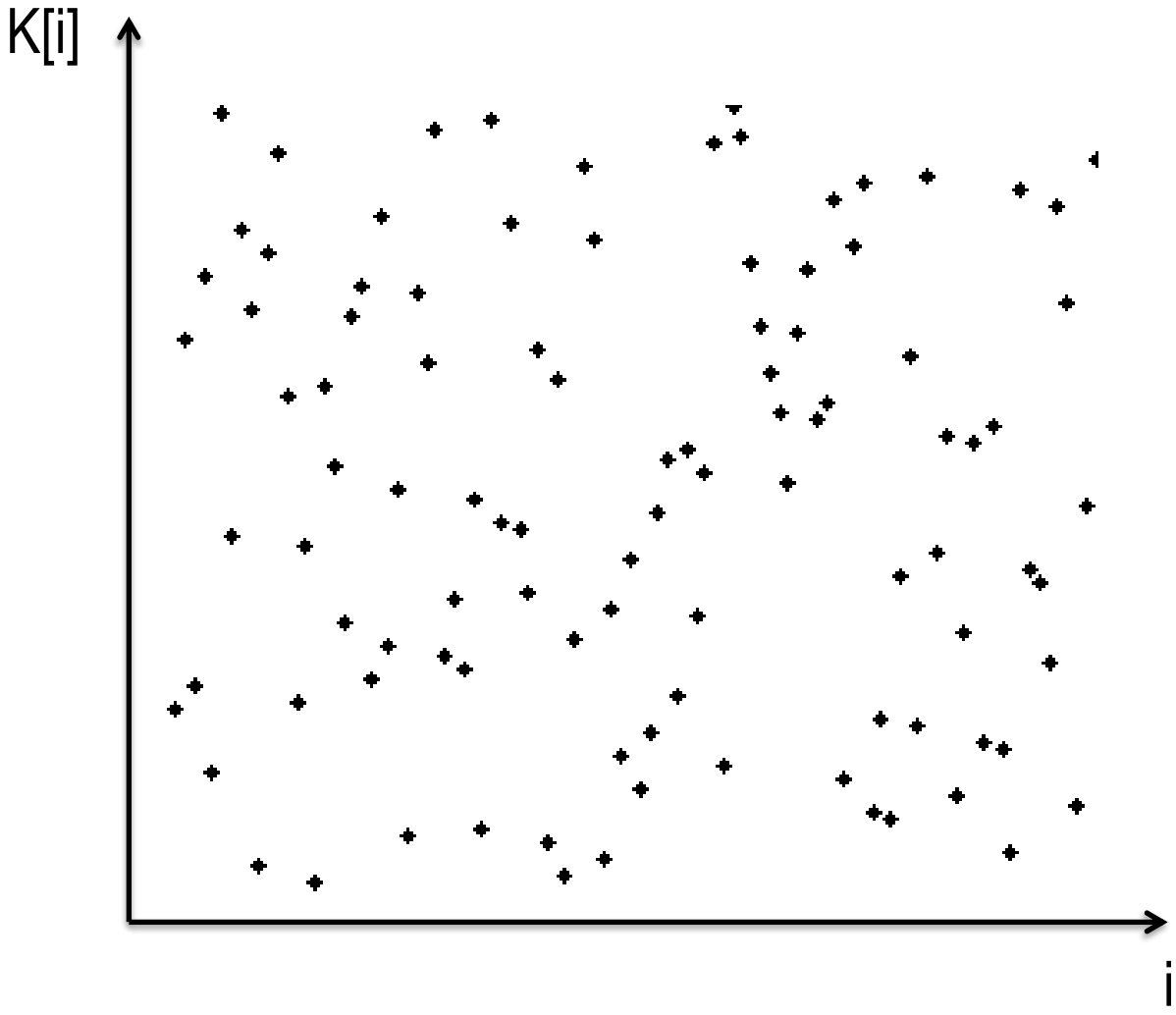
- 在无序序列中做了“选择”
- 在无序序列中做了“移动”

什么算法？



思考

什么算法？



8.5 堆(Heap)排序法 (已在二叉树——堆中讲解)

堆是一棵完全二叉树，二叉树中任何一个分支结点的值都大于或者等于它的孩子结点的值，并且每一棵子树也满足堆的特性。

堆排序的核心思想

是选择排序的堆优化！

第i趟排序将未排序元素(序列的前n−i+1个元素组成的子序列) **转换** 为一个**堆积**，然后将堆的第一个元素与堆的最后那个元素交换位置。

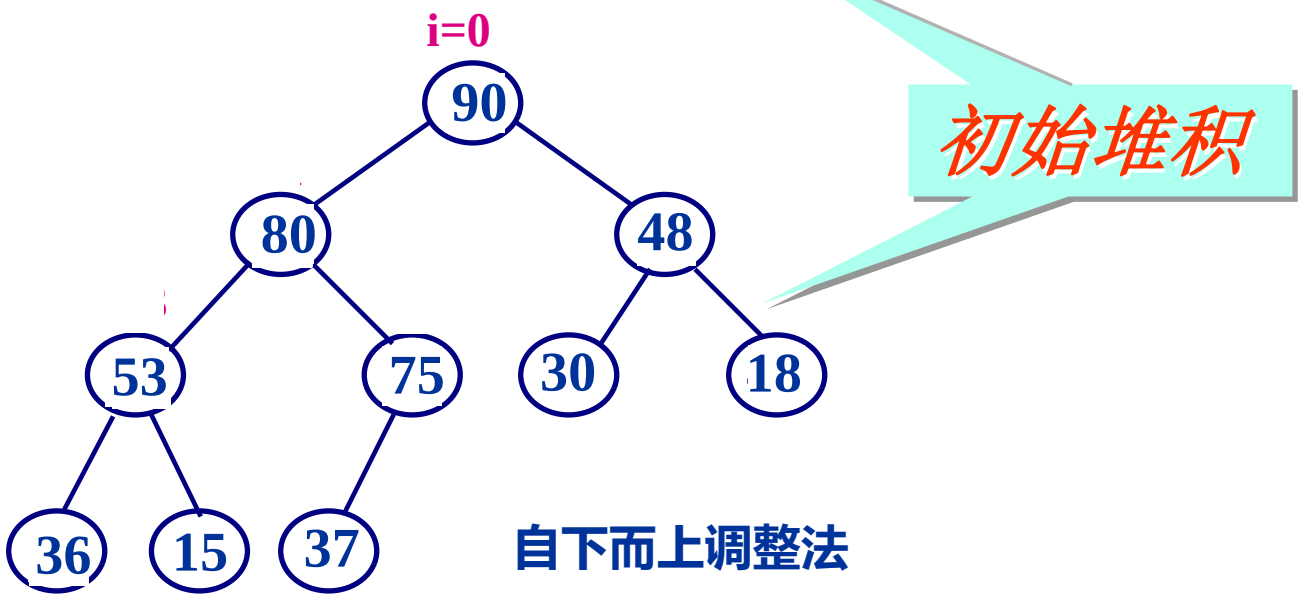
由John Williams提出

建初始堆（自上而下逐点插入法或自下而上调整法）

从二叉树的最后那个分支结点(编号为 $i=\lfloor n/2-1 \rfloor$)开始,依次将编号为i的结点为根的二叉树转换为一个堆，每转换一棵子树，做一次i减1，重复上述过程，直到将i=0的结点为根的二叉树转换为堆。

例

75	36	18	53	80	30	48	90	15	37
90	80	48	53	75	30	18	36	15	37



堆排序步骤

建初始堆积

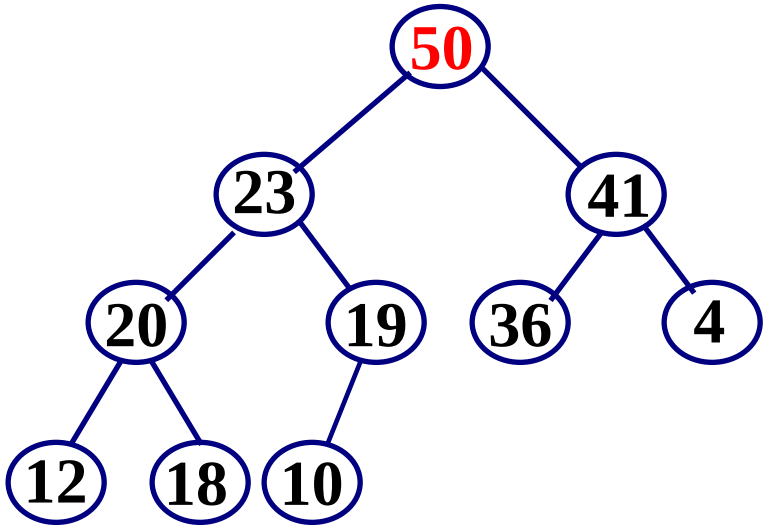
注意：堆可以存储在顺序表中

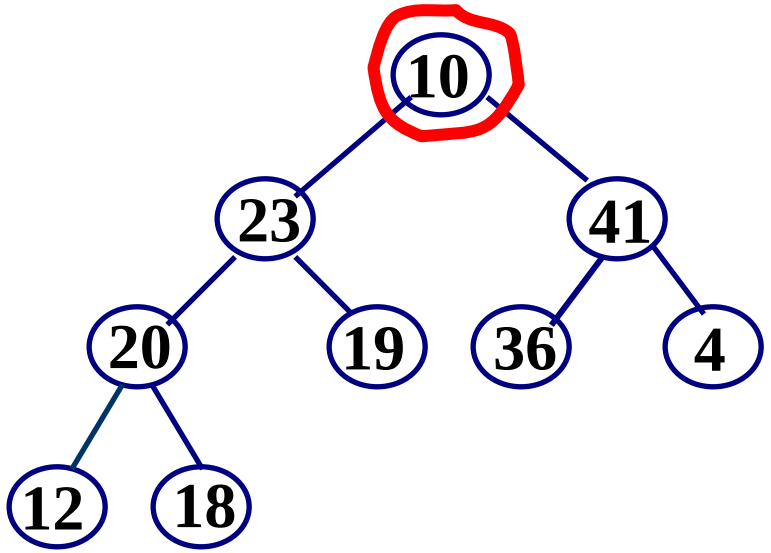
1. 将原始序列**转换**为第一个堆。

2. 将堆的第一个元素与堆积的最后那个元素交换位置。（即“选出”最大值元素并交换位置）

3. 将（“去掉”最大值元素后）剩余元素组成的子序列重新**调整**为堆。

4. 重复上述过程的第2至第3步n–1次。





下滤：将根结点往叶结点方向移

… …

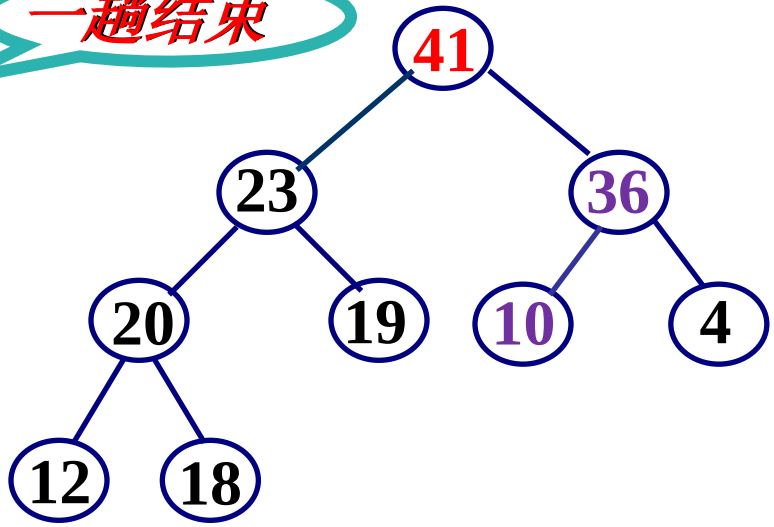
50 23 41 20 19 36 4 12 18 10
10 23 41 20 19 36 4 12 18 50

41 23 36 20 19 10 4 12 18 50
18 23 36 20 19 10 4 12 41 50

36 23 18 20 19 10 4 12 41 50
12 23 18 20 19 10 4 36 41 50

… …

一趟结束



堆调整子算法

```
/*调整一棵子树：i为根，n为子树最后一个结点*/
void  adjust(keytype k[ ], int i, int n){
      int j;
      keytype  temp;
      temp=k[i];
      j=2*i+1;
      while(j<n){
          if(j+1<n && k[j]<k[j+1])  //选出大的孩子
              j++;
          if(temp<k[j]) {  //下滤
              k[(j-1)/2]=k[j];
              j=2*j+1;
          }
          else break;
       }
       k[(j-1)/2]=temp;
}
```

功能

向下调整结点i的位置，使得其祖先结点值 都比其大。如果一棵树仅根结点i不满足堆条件，通过该函数可将其调整为一个堆。

K: 序列
i : 被调整的二叉树的根的序号
n: 被调整的二叉树的结点数目

堆排序完整过程示例

N-1趟，每趟代价为logn

原始	75	36	18	53	80	30	48	90	15	37
第1趟	37	80	48	53	75	30	18	36	15	90
第2趟	15	75	48	53	37	30	18	36	80	90
第3趟	15	53	48	36	37	30	18	75	80	90
第4趟	18	37	48	36	15	30	53	75	80	90
第5趟	18	37	30	36	15	48	53	75	80	90
第6趟	15	36	30	18	37	48	53	75	80	90
第7趟	15	18	30	36	37	48	53	75	80	90
第8趟	15	18	30	36	37	48	53	75	80	90
第9趟	15	18	30	36	37	48	53	75	80	90

堆排序算法

```
void heapSort(keytype k[ ],int n){
    int i,
    keytype  temp;
    for(i=n/2-1;i>=0;i--)
        adjust(k,i,n);
    for(i=n-1;i>=1;i--){
        temp=k[i];
        k[i]=k[0];
        k[0]=temp;
        adjust(k,0,i);
    }
}
```

建初始堆积

具体排序

n−1趟排序

$O(n\log_2 n)$

heap排序是 不稳定的 。

```
for(i=n/2-1;i>=0;i– –)
    adjust(K,i,n);
```

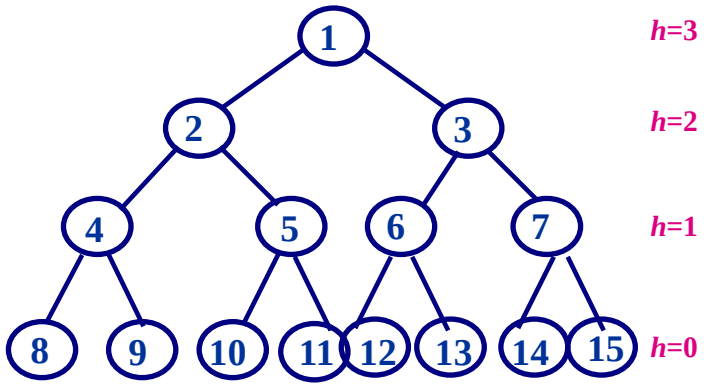
建初始堆积复杂度

第h层节点数 $\leq \left\lceil 2^{(\lg n-h)-1} \right\rceil = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

时间复杂度= $\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$

$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$

$= O(n)$



堆排序算法分析

稳定性： 不稳定

时间代价： $O(n\log_2 n)$

空间代价： O(1)

注：此处仅考虑辅助空间

注：可看作是选择排序的优化（提高选择的效率）

本章内容

8.1 排序的基本概念

8.2 插入排序法

8.3 选择排序法

8.4 泡排序法

8.5 堆排序法

8.6 快速排序法

8.7 归并排序法

8.8 谢尔排序法

8.9 非比较排序法*

8.6 快速排序法(Quick Sort)

核心思想

也称为划分元素、基准元素、支点等。可选第1个或者最后1个、或位置居中元素作为分界元素。

从当前参加排序的元素中任选一个元素(称为**分界元素** pivot)，将当前参加排序的元素分成前后两部分（凡是小于分界元素的元素都移到分界元素的前面，凡是大于分界元的元素都移到分界元素的后面）。然后，分别对这两部分数据重复上述过程，直到每一部分的长度等于1。

本质上它是一种分治算法
(divide and conquer algorithm)

递归过程

快速排序算法最早由图灵奖获得者Tony Hoare设计出来，被列为20世纪十大算法之一。也是目前实践中已知最快的排序算法。

如何实现高效的划分?

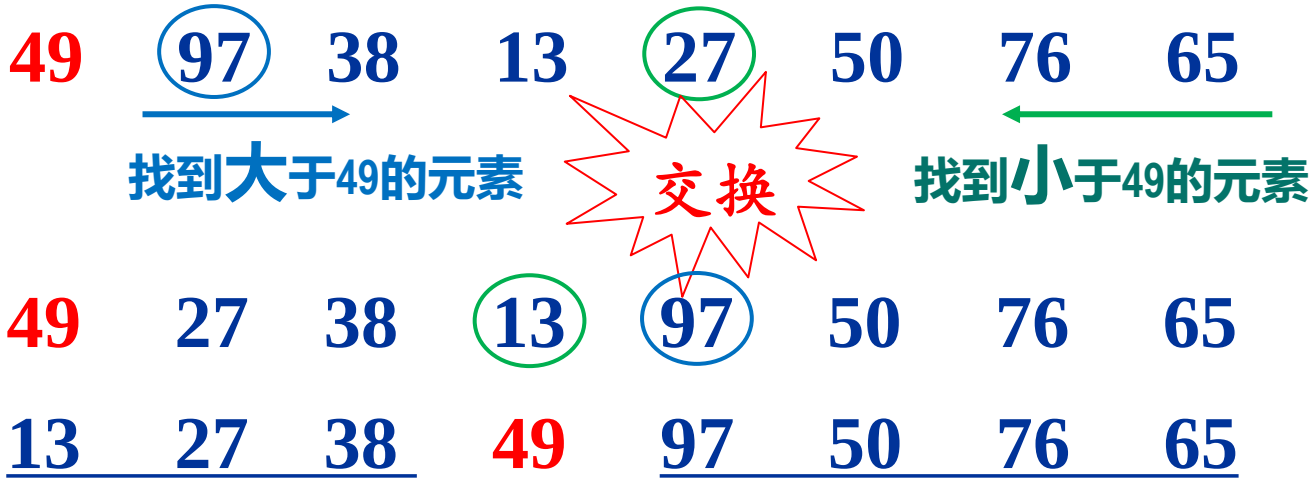
分界元素

49 97 38 13 27 50 76 65

朴素的想法……

38 13 27 49 97 50 76 65

高效的做法……(双向扫描)



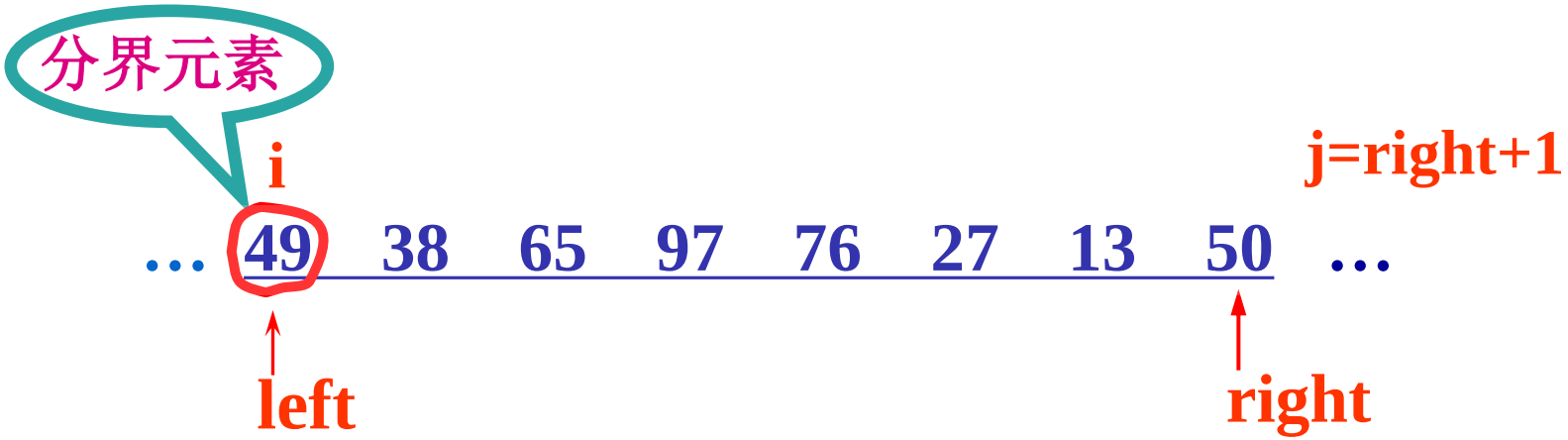
算法步骤

算法中用到的变量

left：当前参加排序的序列的第一个元素在序列中的位置，初始值为0。

right：当前参加排序的那些元素的最后那个元素在序列中的位置，初始值为n-1 。

i，j：两个位置变量，初始值分别为left 与right+1。



步骤

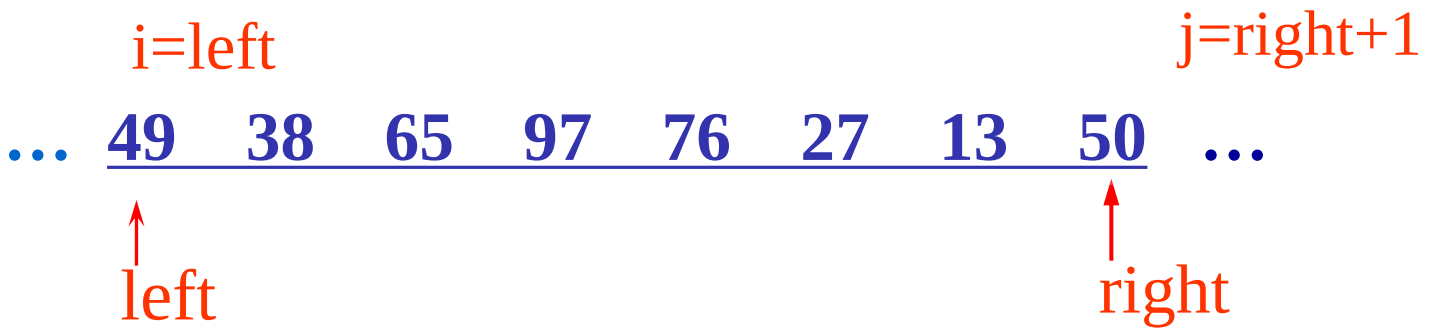
分界元素K[Left]

1. 反复执行动作++i，直到K[left]≤K[i]或者i=right。
   反复执行动作--j，直到K[left]≥K[j]或者j=left。

2. 若i<j，则交换K[i]与K[j]，转到第1步。

3. 若i≥j，则交换K[left]与K[j],到此,分界元素K[left]的最终位置已经确定，本趟排序结束。

一趟排序后，序列被分界元素分成两部分，对长度大于1的部分重复上述过程，直到排序结束。



left right

49	38	65	97	76	27	13	50
i							j

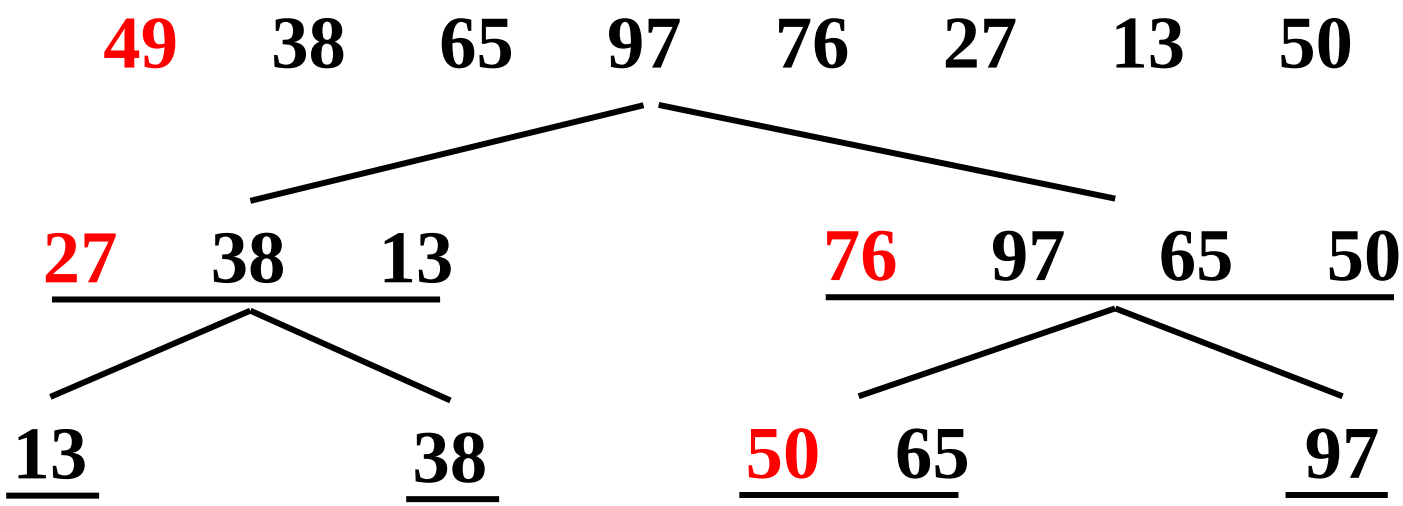
49	38	65	97	76	27	13	50
		i				j	

49	38	13	97	76	27	65	50
			i		j		

49	38	13	27	76	97	65	50
			j	i			

i>j, 此时 i 停留在第一个比分界数（49）大的位置
j 停留在最后一个比分界数（49）小的位置

27	38	13	49	76	97	65	50
			j	i			



每次划分：逻辑上以分界元素为根划分左右子树

递归的划分过程逻辑上构造了一棵二叉查找树

平均情况树的深度为logN
最差情况(已经有序)退化成深度为N的二叉树

快速排序算法(双向扫描版)

```
void quick(keytype k[ ], int left, int right){
    int i, j;
    keytype pivot;
    if(left<right){
        i=left; j=right+1;
        pivot = k[left];
        while(1){
            while(k[++i]<pivot && i!=right) {; }
            while(k[--j]>pivot && j!=left) {; }
            if(i<j)
                swap(&k[i], &k[j]);  /*交换K[i]与K[j]的内容*/
            else
                break;
        }
        swap(&k[left],&k[j]);  /*交换K[Left]与K[j]的内容*/
        quick(k,left,j−1);          /*  对前一部分排序  */
        quick(k,j+1,right);         /*  对后一部分排序  */
    }
}
```

主算法

```
void quickSort(keytype k[],int n){
    quick(K,0,n-1);
}
```

快速排序算法(单向扫描版)


```
void qsort(keytype v[ ],int left, int right){
    int i, last;

    if(left >= right)
        return;
    swap(v, left, (left+right)/2); //move partition elem to v[0]
    last = left;
    for(i=left+1; i<=right; i++)   //partition
        if(v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); //restore partition elem
    qsort(v, left, last);
    qsort(v, last+1, right);
}
```

主算法

```
void quickSort(keytype k[],int n){
    qsort(k,0,n-1);
}
```

from K & R

快速排序算法分析

稳定性： 不稳定

最差情况：
- 时间代价：$O(n^2)$
- 空间代价：$O(n)$

类似于二叉树退化成线性表

最佳情况：
- 时间代价：$O(n\log_2 n)$
- 空间代价：$O(\log_2 n)$

平均情况：
- 时间代价：$O(n\log_2 n)$
- 空间代价：$O(\log_2 n)$

8.7 归并(Merge)排序法

一、什么是二路归并?

由John von Neumann提出

将两个位置相邻、并且各自按值有序的子序列合并为一个按值有序的子序列的过程称为**二路归并**。

$(K_s, K_{s+1}, K_{s+2}, \ldots\ldots, K_u)$ $(K_{u+1}, K_{u+2}, K_{u+3}, \ldots\ldots, K_v)$

$(X_s, X_{s+1}, X_{s+2}, X_{s+3}, \ldots\ldots, X_v)$

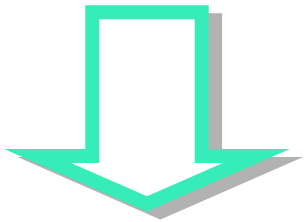
其中 $K_s \le K_{s+1} \le K_{s+2} \le \ldots\ldots \le K_u$

$K_{u+1} \le K_{u+2} \le K_{u+3} \le \ldots\ldots \le K_v$

$X_s \le X_{s+1} \le X_{s+2} \le X_{s+3} \le \ldots\ldots \le X_v$

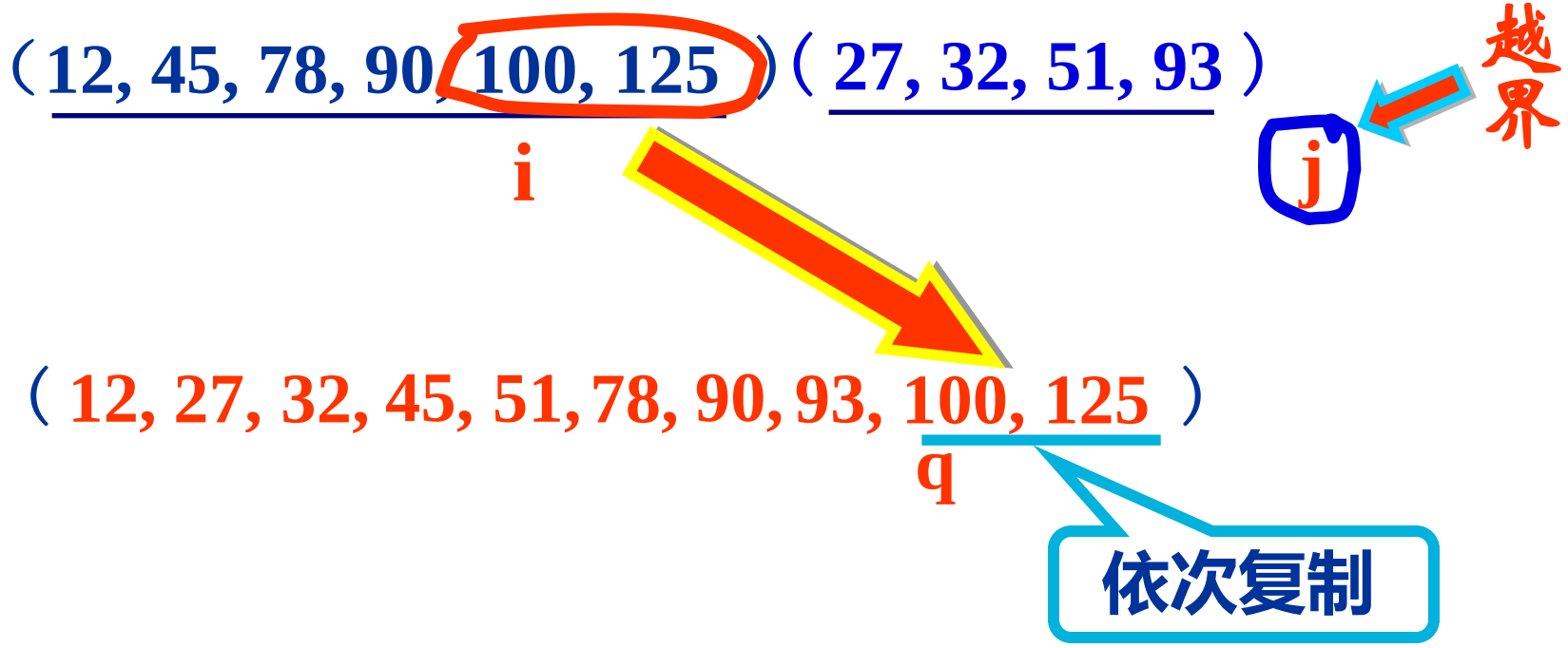
… K_s, K_{s+1}, K_{s+2}, K_{s+3}, K_{s+4}, K_{s+5} K_{s+6}, K_{s+7}, K_{s+8}, K_{s+9} …

（12, 45, 78, 90, 100, 125) (27, 32, 51, 93)



(12, 27, 32, 45, 51, 78, 90, 93, 100, 125)

二路归并


（12, 45, 78, 90, 100, 125）（27, 32, 51, 93）
i
j
越界
（12, 27, 32, 45, 51, 78, 90, 93, 100, 125）
q
依次复制

```
void merge(keytype x[ ], keytype tmp[ ],
                    int left, int leftend, int rightend){
    int i=left, j=leftend+1, q=left;
    while(i<=leftend && j<=rightend)
        if(x[i]<=x[j])
            tmp[q++]=x[i++];
        else
            tmp[q++]=x[j++];
    while(i<=leftend)
        tmp[q++]=x[i++];
    while(j<=rightend)
        tmp[q++]=x[j++];
    for(i=left; i<=rightend; i++)
        x[i]=tmp[i];
}
```

功能 将两个位置相邻且按值有序子序列合并为一个按值有序的序列

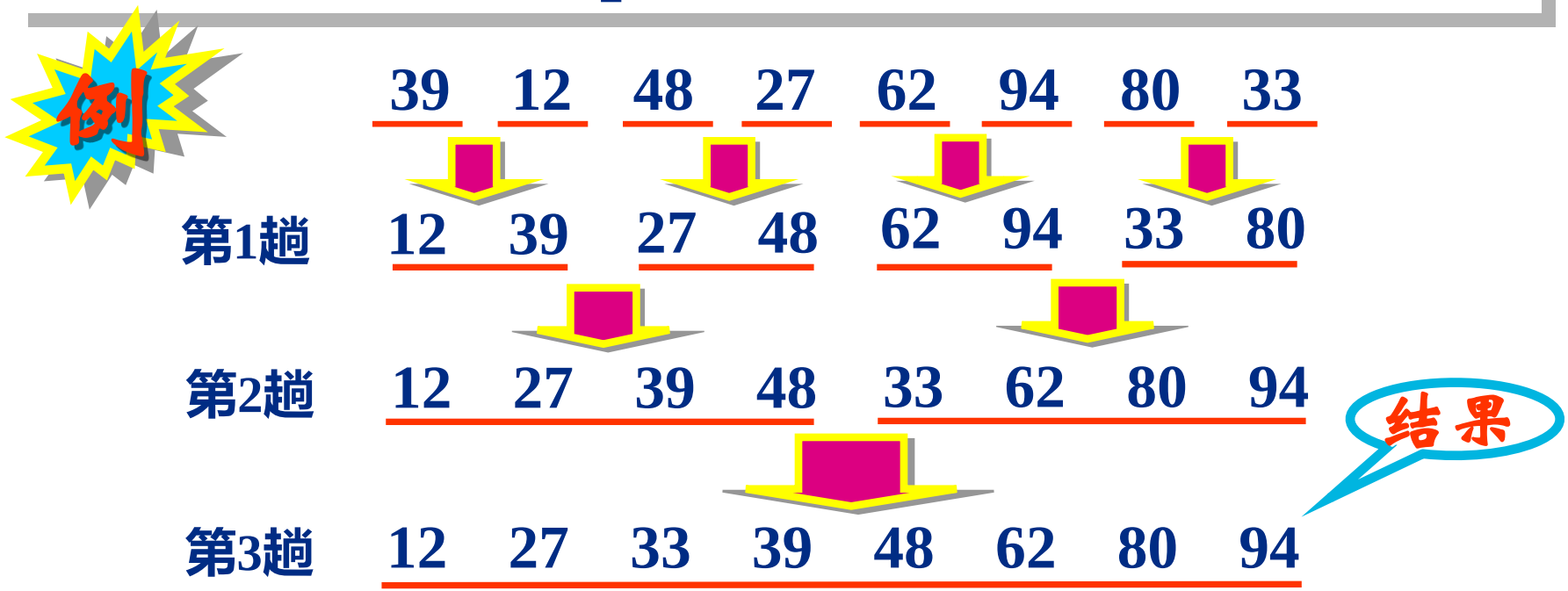
复制第一个子序列的剩余部分

复制第二个子序列的剩余部分

将合并后内容复制回原数组

二、归并排序完整过程

第i趟排序将序列的$\lfloor \frac{n}{2^{i-1}} \rfloor$个长度为$2^{i-1}$的按值有序的子序列依次两两合并为$\lfloor \frac{n}{2^{i}} \rfloor$个长度为$2^{i}$的按值有序的子序列。



该排序可以实现成递归或非递归算法。

排序过程逻辑上对应了一棵平衡二叉树。

三、归并排序算法

此处为递归算法实现

```
void mSort(keytype k[], keytype tmp[], int left, int right){
    int center;
    if(left < right){
        center = (left+right)/2;            //数据划分为左右两片
        mSort(k, tmp, left, center);        //左半片数据排序
        mSort(k, tmp, center+1, right);     //右半片数据排序
        merge(k, tmp, left, center, right); //合并左右两部分数据
    }
}
```

时间复杂度 $O(n\log_2 n)$

空间复杂度 $O(n)$

```
void mergeSort(keytype k[ ],int n){//主程序
    keytype *tmp;
    tmp = (keytype *)malloc(sizeof(keytype) * n);
    if(tmp != NULL) {
        mSort(k, tmp, 0, n-1);
        free(tmp);
    }else
        printf("No space for tmp array!!!\n");
}
```

归并排序算法分析

稳定性： 稳定

时间代价： $O(n\log_2 n)$

最好、最坏及平均时间代价均为$O(n\log_2 n)$

空间代价： $O(n)$

Sort-Merge： 数据划分成多个子序列，子序列内采用QuickSort等算法排序，最后依次Merge子序列，直到完成排序

无需将数据整体加载至内存，可流式处理，因而适合超大规模数据的排序

8.8 谢尔(Shell)排序法

由Donald Shell提出

缩小增量排序法

核心思想

首先确定一个元素的间隔数**gap**。将参加排序的元素按照**gap**分隔成若干个子序列(即分别把那些位置相隔为**gap**的元素看作一个子序列)，然后对各个子序列采用 ***某一种排序方法*** 进行排序；

此后减小**gap**值，重复上述过程，直到**gap<1**。

一种减小gap的方法:

$gap_1 = \lfloor n/2 \rfloor$

$gap_i = \lfloor gap_{i-1}/2 \rfloor \quad i = 2, 3, \ldots$

初 始	49	97	38	50	76	65	13	27	25
第1趟	49	97	38	50	76	65	13	27	25
gap=4	25	65	13	27	49	97	38	50	76
第2趟	25	65	13	27	49	97	38	50	76
gap=2	13	27	25	50	38	65	49	97	76
第3趟 gap=1	13	25	27	38	49	50	65	76	97

注意： 第i趟排序时，各子序列内已相对有序！此时子序列内应采用哪种排序？用选择排序？

原始算法中，子序列内使用插入排序，因此希尔排序被认为是对插入排序的改进。实际上可结合使用泡排序等。

```
void  shellSort(int v[ ], int n) { //from K & R
    int gap, i, j, temp;
    for( gap = n/2; gap >0; gap /= 2)
        for(i=gap; i<n; i++)
            for(j=i-gap; j>=0 && v[j]>v[j+gap]; j -= gap){
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

from K & R

```
void  shellSort(keytype k[ ],int n){
    int  i, j, gap=n;
    keytype  temp;
    while(gap>1){
        gap=gap/2;
        //子系列使用插入排序
        for (i = gap; i < n; i ++)  {
            temp = k[i]
            for (j = i; j >= gap && k[j - gap] > temp; j -= gap)
                k[j] = k[j - gap]
            k[j] = temp
        }
    }
}
```

```
void  shellSort(keytype k[ ],int n){
      int  i, j, flag, gap=n;
      keytype  temp;
      while(gap>1){
            gap=gap/2;
            do{ //子序列用泡排序
                flag=0;  /* 每趟排序前, 标志flag置0 */
                for(i=0;i<n–gap;i++){
                    j=i+gap;
                    if(k[i]>k[j]){
                        temp=k[i];     k[i]=k[j];     k[j]=temp;
                        flag=1;
                    }
                }
             }while(flag!=0);
       }
}
```

经过$\lfloor \log_2 n \rfloor$趟排序

$O(n\log_2 n)$与$O(n^2)$之间

通常$<O(n^{3/2})$

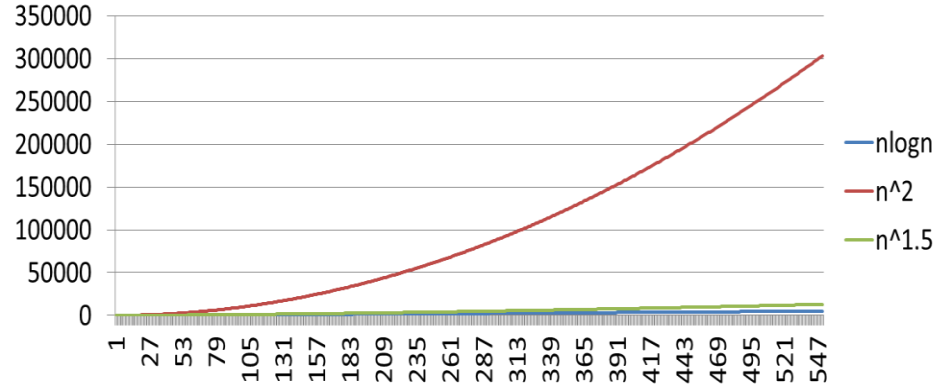
shell排序是不稳定的 。

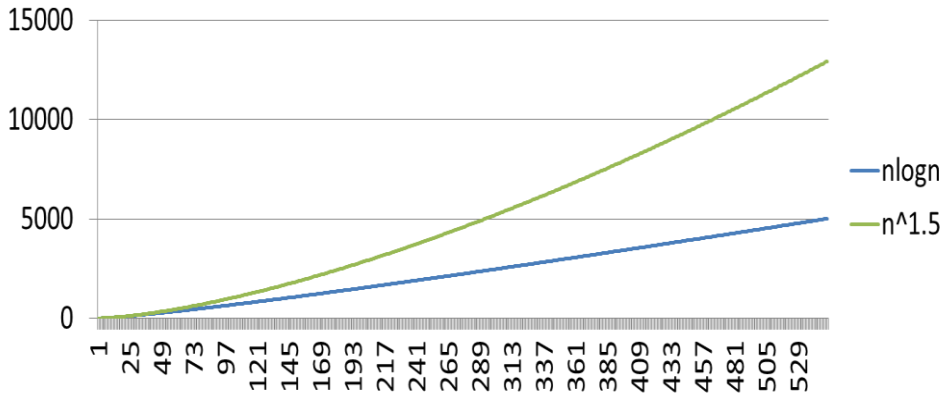
Shell排序算法分析

稳定性： 不稳定

时间代价： $O(n\log_2 n)$与$O(n^2)$之间
（通常<$O(n^{3/2})$）

空间代价： O(1)





排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n)\sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)\sim O(n)$	不稳定

从算法性质来看：

- 简单算法：冒泡、选择、插入
- 改进算法：谢尔、堆、归并、快速

属于“基于关键字比较”的排序

从时间复杂度来看：

- 平均情况：后3种改进算法 > 谢尔（远）> 简单算法
- 最好情况：冒泡和插入排序要更好一些
- 最坏情况：堆和归并排序要好于快速排序及简单排序

从空间复杂度来看：

- 归并排序有额外空间要求，快速排序也有空间要求，堆排序则基本没有。

从算法稳定性来看：

- 除了简单排序，归并排序不仅速度快，而且还稳定

工程实践中可能综合/混合以上算法，如Sort-Merge

思考题

a1，a2，a3，a4，a5，a6，a7，a8，a9

(插入、选择、冒泡、堆排、快排、归并、谢尔)

- **排序的稳定性**
  - 稳　定：插入、冒泡、归并
  - 不稳定：选择、堆排、快排、谢尔
- **排序在一趟结束后可选出一个元素放在其最终位置上**
  - 可以确定　：选择、冒泡、堆排、快排、
  - 不可以确定：插入、归并、希尔

【填空5：插入排序】插入排序法的时间花费主要取决于元素间的比较次数，若具有n个元素的序列初始时已经是一个递增序列，则排序过程中一共要进行_____次比较。

【题10：排序】若要进行从小到大排序，数据元素序列 11, 12, 13, 7, 8, 9, 23 , 4, 5是采用下列排序方法之一得到的第二趟排序后的结果，则该排序算法只能是（）

A: 冒泡排序 B: 插入排序 C: 选择排序 D: 二路归并排序

【选择14：快排】给出一组关键字序列{12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18}，当用快速排序（选第一个记录为基准点进行划分，采用教材P336-337描述的算法）从小到大进行排序第一趟结束时的序列为【 】

A. 6，2，8，10，4，12，28，30，16，20，18
B. 6，4，8，10，2，12，28，30，16，20，18
C. 4，2，6，10，8，12，28，30，20，16，18
D. 4，2，8，10，6，12，16，20，28，30，18

【快排】若利用**快速排序**算法进行从小到大排序，下列选项中，不可能是经过两次选择分界元素并确定其最终位置后的排序结果的是 【】：

A. 2,3,5,4,6,7,9　　B. 2,7,5,6,4,3,9
C. 3,2,5,4,7,6,9　　D. 4,2,3,5,7,6,9

8.9 非关键字比较排序*

“分配式排序”(distribution sort)，包括

- 计数排序
- 桶排序
- 基数排序

计数排序(counting sort)*

假设$a_1,a_2...a_n$由小于M的正整数组成，计数排序使用一个大小为M的数组C(初始化为0)，当处理a_i时，使$C[a_i]$增1。最后遍历数组C输出排序后的表。

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

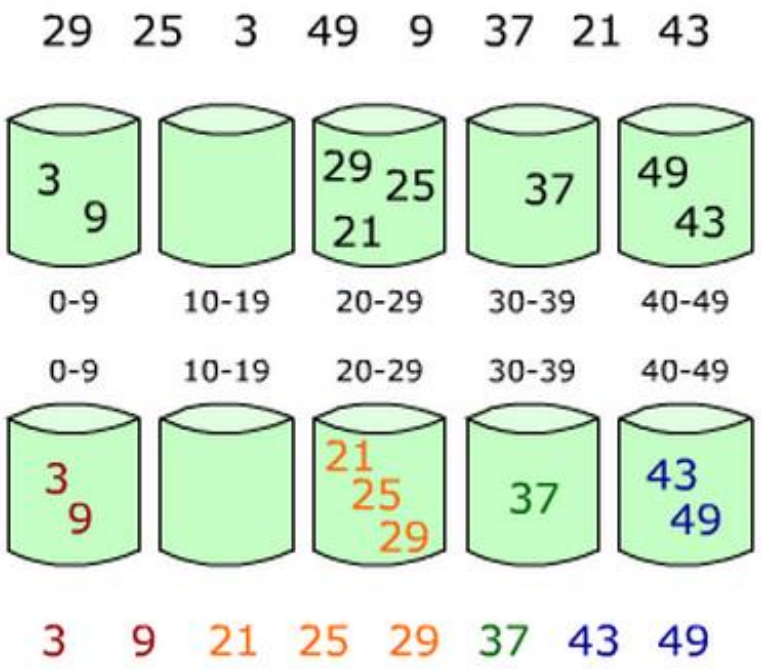
算法基本思想由E.J.Issac R.C.Singleton提出，是最简单、最快的排序，其时间复杂度为O(M+N)。

限制：数值范围和数组C的大小

桶(Bucket)排序法*

其他策略？

采用分治策略(divide-and-conquer)，首先根据数值范围建立多个桶(buckets)，将数据数据元素分配到相应的桶里，然后每个bucket各自排序，或用不同的排序算法，或者递归的使用bucket sort算法。



Range Partition

限制：数值范围和桶的数量/大小

基数排序(Radix Sort)*

基数排序的发明可以追溯到1887年赫尔曼·何乐礼在打孔卡片制表机(Tabulation Machine)上的贡献，其原理：

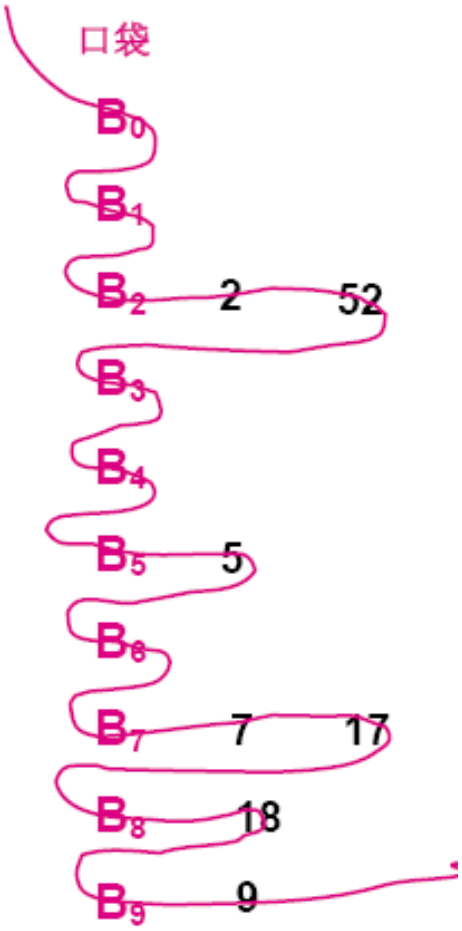
将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

基数排序的方式可以采用LSD（Least significant digital）或MSD（Most significant digital）

- LSD由键值的最右边开始的排序方式
- MSD由键值的最左边开始

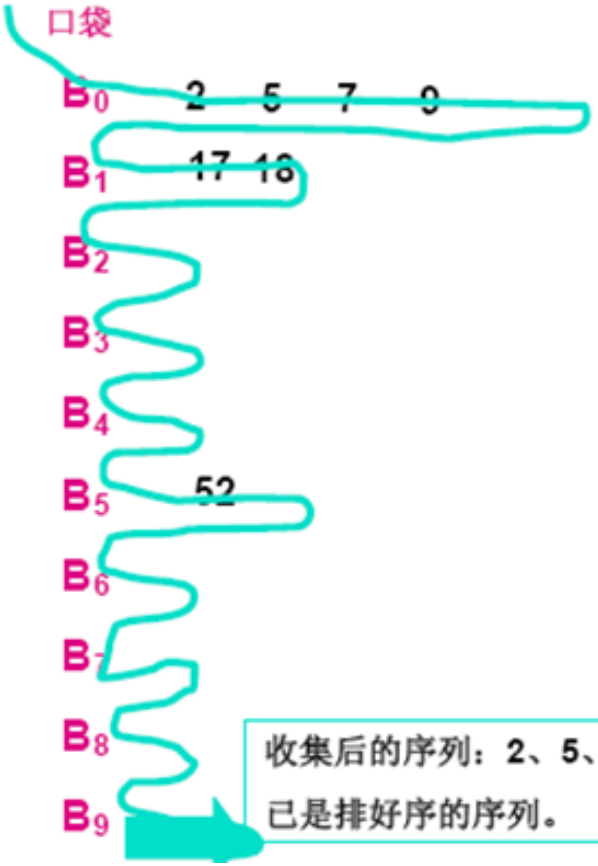
基数排序示例

B = 5、2、9、7、18、17、52



按最后一位有序

B = 2、52、5、7、17、18、9（第一次收集的结果）



按倒数第二位有序

本章结束！