第九章 排序

- 问题定义
- 基本排序方法
- 归并排序、快速排序、堆排序、希尔排序
- 基数排序
- 内部排序方法的比较
- 本章小结

9.1问题定义：
排序(sorting)

- 排序的功能是将一个数据元素（或记录）的任意序列重新排列成一个按关键字有序的序列。
  - 升序：关键字从小到大
  - 降序：关键字从大到小

排序的稳定性

- 若序列中关键字值相等的节点经过某种排序方法进行排序之后，仍能保持它们在排序前的相对顺序，则称这种排序方法是稳定的；否则，称这种排序方法是不稳定的。
- 稳定性例

  待排序列： **34** 12 34′ 08 96

  稳定： 08 12 **34** 34′ 96

  不稳定： 08 12 34′ 34 96

排序的分类——内排、外排

- 根据内存使用情况：
  - 内部排序：数据存储调整均在内存中进行
  - 外部排序：大部分节点在外存中，借助内存进行调整

排序的分类——“比较-交换”、“分配”

- 根据排序实现手段：
  - 基于“比较-交换”的排序——通过对关键字的比较，交换关键字在序列中的位置
    - 插入排序、冒泡排序、选择排序、快速排序、归并排序、希尔排序、堆排序
  - 基于“分配”的排序
    - 桶排序、基数排序

排序的分类——基本、高级排序

- 根据实现的难易程度：
  - 基本排序：插入、冒泡、选择排序
  - 高级排序：快速排序、归并排序、堆排序

9.2.1插入排序(insertion sort)

9.2.1插入排序(insertion sort)

- 基本思想:将一个记录插入到已排好顺序的序列中，形成一个新的、记录数增1的有序序列。
- 假设$a_0,a_1,\ldots,a_{i-1}$已排序($a_0 \le a_1 \le \ldots \le a_{i-1}$)，对于$t=a_i$，从大到小进行比较，比t大的节点右移一位，直到发现某个值不大于t。最后令$a_{j+1}=t$。

插入排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	D	T	A	S	T	R	U	C	T	U	R	E
A	D	T	A	S	T	R	U	C	T	U	R	E
A	A	D	T	S	T	R	U	C	T	U	R	E
A	A	D	S	T	T	R	U	C	T	U	R	E
A	A	D	S	T	T	R	U	C	T	U	R	E
A	A	D	R	S	T	T	U	C	T	U	R	E
A	A	D	R	S	T	T	U	C	T	U	R	E
A	A	C	D	R	S	T	T	U	T	U	R	E
A	A	C	D	R	S	T	T	T	U	U	R	E
A	A	C	D	R	S	T	T	T	U	U	R	E
A	A	C	D	R	R	S	T	T	T	U	U	E
A	A	C	D	E	R	R	S	T	T	T	U	U

插入排序动画

45	34	78	12	*34'*	32	29	64

程序9-1 插入排序方法

```
template <class Item>
void InsertionSort(Item a[], int l, int r)
{
        int i,j; Item t;
        for (i=l+1; i<=r; ++i)// 从左边界开始，依次获取每个记录
         { // 将获取到的记录插入到前面已排好序的序列的合适位
          //置。方法是：从当前记录开始逐个比较前面的记录若当
        //前记录小，则把前面的记录向后移一个位置
                for (j=i-1,t=a[i]; j>=0&&t<a[j]; j--)
                        a[j+1] = a[j];
                // 将最初获取的记录复制到相应位置
                a[j+1] = t;
        }
}
```

插入排序性能分析

- 时间代价：
  - 最佳情况（排序）：n-1次比较，0次移动，O(n)
  - 最差情况（逆序）：O(n^2)
    - 比较次数为: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
    - 移动次数为: $\sum_{i=1}^{n-1}(i+1) = \frac{(n+2)(n-1)}{2}$
  - 平均情况：O(n^2)

- 空间代价：O(1)
- 稳定
- 适用节点个数少的场合

9.2.2冒泡排序(bubble sort)

- 基本思想:依次比较相邻的两个元素的顺序，如果顺序不对，则将两者交换，重复这样的操作直到整个序列被排好序。
- 实际是通过比较与交换使得待排序列一个最值元素“上浮”到序列一端，然后缩小排序范围

冒泡排序步骤

- 假设待排序的序列为$a_0,a_1,a_2,\ldots,a_{n-1}$
- 起始时排序范围是从a_0至a_{n-1}，
- 自右向左对相邻两结点进行比较，让较大向右移，让较小向左移。当比较完当前排序范围后，键值最小的元素被移动到序列的a_0位置，故a_0无需再参加下一次比较
- 下一次比较的范围为a_1至a_{n-1}。

冒泡排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	D	A	T	C	S	T	R	U	E	T	U	R
A	A	D	C	T	E	S	T	R	U	R	T	U
A	A	C	D	E	T	R	S	T	R	U	T	U
A	A	C	D	E	R	T	R	S	T	T	U	U
A	A	C	D	E	R	R	T	S	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U

冒泡排序动画

45	34	78	12	*34′*	32	29	64

程序9-2 冒泡排序方法

```
template <class Item>
void BubbleSort(Item a[], int l, int r)
{
// 1. 比较相邻的元素。如果第一个比第二个大，就交换它们。
// 2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最
//后一对这时，最后的元素应该会是最大的数。
// 3. 针对所有的元素重复以上的步骤，除了最后一个。
// 4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何
//一对数字需要比较。
        for (int i=l; i<r; ++i)                  // 进行r-l趟过程
              for (int j=i; j<r-1; ++j)     // 从左至右比较相邻记录
                    if (a[j]>a[j+1])           // 若a[j]>a[j+1]
                          swap(a[j],a[j+1]);   // 交换a[j]和a[j+1]
}
```

冒泡排序性能分析

- 时间代价：
  - 最佳情况（排序）：$\frac{n(n-1)}{2}$ 次比较，0次移动，**O(n²)**(可通过改进达到O(n))
  - 最差情况（逆序）：**O(n²)**
    - 比较与移动次数均为: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - 平均情况：**O(n²)**
  - 移动与比较次数都很多，速度很慢
- 空间代价：**O(1)**
- 稳定

9.2.3选择排序(selection sort)

- 基本思想:首先选出键值最小的项，将它与第一个项交换位置；然后选出键值次小的项，将其与第二个项交换位置；…；直到整个序列完成排序。

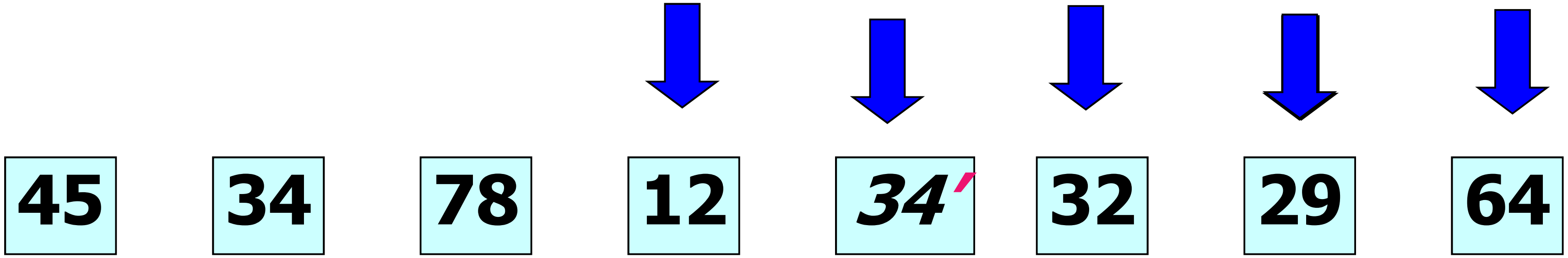
9.2.3选择排序步骤

- 假设待排序的序列为$a_0, a_1, a_2, \ldots, a_{n-1}$，
- 依次对$i=0,1,\ldots,n-2$分别执行如下的选择步骤：
  - 在$a_i, a_{i+1}, \ldots, a_{n-1}$中选择一个键值最小的项$a_k$
  - 然后将a_k与a_i交换。
- 与冒泡排序区别在于选择待排序列最小项只进行一次交换：

选择排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	D	T	A	S	T	R	U	C	T	U	R	E
A	A	T	D	S	T	R	U	C	T	U	R	E
A	A	C	D	S	T	R	U	T	T	U	R	E
A	A	C	D	S	T	R	U	T	T	U	R	E
A	A	C	D	E	T	R	U	T	T	U	R	S
A	A	C	D	E	R	T	U	T	T	U	R	S
A	A	C	D	E	R	R	U	T	T	U	T	S
A	A	C	D	E	R	R	S	T	T	U	T	U
A	A	C	D	E	R	R	S	T	T	U	T	U
A	A	C	D	E	R	R	S	T	T	U	T	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U

选择排序动画



程序9-3 选择排序方法

```
template <class Item>
void SelectionSort(Item a[], int l, int r)
{
        int i,j,min;
        for (i=l; i<r; ++i){
         //依次从剩余的未排序序列中选取一个最小的记录
         for (min = i, j=i+1; j<=r; ++j)
              if (a[j]<a[min])
                   min = j;
         //将当前的最小记录放入已排序好的队列的末尾
         swap(a[i],a[min]);
        }
}
```

选择排序性能分析

- 时间代价：
  - 最佳情况（排序）：$\frac{n(n-1)}{2}$ 次比较，0次移动，**O(n²)**
  - 最差情况：**O(n²)**
    - 比较次数均为：$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
    - 移动次数为：**(n-1)**
  - 平均情况：**O(n²)**
- 空间代价：**O(1)**
- 不稳定
- 运行时间与记录顺序关系很小
- 交换次数很少

9.3归并排序(merging sort)

- 基于分治法
- 基本思想:将两个或两个以上的有序序列合并为一个新的有序序列。
  - **2**路归并：在两个已排序的序列中分别选出最小的项，比较选出较小的放入新序列中，循环直到两序列所有项均在新序列中。
  - 多路归并：选择合并的序列多于**2**
- 归并的次序
  - 自顶向下
  - 自底向上

自顶向下归并排序过程

(25 34 45 32 78 12 34’ 64)

(25 34 45 32)(78 12 34’ 64)

先划分

(25 34)(45 32)(78 12)(34’ 64)

(25)(34)(45)(32)(78)(12)(34’)(64)

(25 34)(32 45)(12 78)(34’ 64)

再归并

(25 32 34 45)(12 34’ 64 78)

(12 25 32 34 34’ 45 64 78)

自顶向下归并排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	**D**											
		A	**T**									
A	**A**	**D**	**T**									
				S	**T**							
				R	**S**	**T**						
A	**A**	**D**	**R**	**S**	**T**	**T**						
							C	**U**				
							C	**T**	**U**			
										R	**U**	
										E	**R**	**U**
							C	**E**	**R**	**T**	**U**	**U**
A	**A**	**C**	**D**	**E**	**R**	**R**	**S**	**T**	**T**	**T**	**U**	**U**

程序9-5 归并排序方法

```
template <class Item>
void merge(Item a[ ], int l, int m, int r)
{ //2路归并过程
        int i, j;
        //重新形成一个序列: 序列的左半部分升序, 右半部分降序
        static Item aux[N];
        for (i=m+1;i>l;i--)   aux[i-1]=a[i-1];
        for (j=m;j<r;j++)     aux[r+m-j]=a[j+1];
        // 将aux数组中的两个序列归并到a数组
        for (int k=l; k<=r; k++)
                if (aux[j]<aux[i])
                        a[k]=aux[j--];
                else
                        a[k]=aux[i++];
}
```

程序9-5 归并排序方法

```
template <class Item>
void MergeSort(Item a[], int l, int r)
{
        // 递归退出条件: 如果区间内已只有一个记录, 直接返回
        // 同时, 也是一个参数合理与否的判断条件
        if (r<=l) return;
        // 将待排序序列按中心位置划分为两个待排序序列
        int m = (r+l)/2;
        // 对序列的前半段进行递归的归并排序
        MergeSort(a,l,m);
        // 对序列的后半段进行递归的归并排序
        MergeSort(a,m+1,r);
        // 对两个已排好序的序列进行归并
        merge(a,l,m,r);
}
```

自底向上归并排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	**D**											
		A	**T**									
				S	**T**							
						R	**U**					
								C	**T**			
										R	**U**	
A	**A**	**D**	**T**									
				R	**S**	**T**	**U**					
								C	**R**	**T**	**U**	
A	**A**	**D**	**R**	**S**	**T**	**T**	**U**					
								C	**E**	**R**	**T**	**U**
A	**A**	**C**	**D**	**E**	**R**	**R**	**S**	**T**	**T**	**T**	**U**	**U**

程序9-6 自底向上归并排序方法

```
inline int min(int A, int B)
{
        return (A<=B)?A:B;
}

template <class Item>
void MergeSortBU(Item a[], int l, int r)
{
        //归并: 首先按步长为1进行归并, 然后按步长×2进行归并,
        //    ..., 直到步长大于当前序列
        for (int m=1; m<=r-l; m=m+m)
         //内层循环, 对当次归并: 从左至右, 按步长依次两两归并
                for (int i=l; i<=r-m; i+= m+m)
                        merge(a, i, i+m-1, min(i+m+m-1,r));
                        //右边界不要超出序列范围
}
```

归并排序性能分析

- 时间代价 $O(n \log_2 n)$
  - 共[$\log_2(n-1)+1$]次归并
  - 每次参与归并的节点不多于n
  - 共需$n\log_2 n$次比较
- 空间代价：**O(n)**
- 稳定
- 运行时间与记录顺序关系很小

9.4快速排序(quick sort)

- 平均性能最好的排序算法
- 基本思想（分治、二分法）：
  1. 选择控制值。
  2. 放置控制值，使得控制值左面所有项小于控制值，右面所有项大于控制值
  3. 对控制值左、右两面分别进行以上操作

快速排序重要操作——partition

- 划分操作主要作用是确立控制值的位置，还有缩小待排序列元素的交换范围，使其只能在控制值左（右）序列范围内移动

一次划分的具体过程示例

一次划分的具体过程为：

1. low指向待划分区域首元素，high指向待划分区域尾元素；

如： 将序列 49、38、65、97、76、13、27、49‘ 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

	49	38	65	97	76	13	27	49'

low high

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素;
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;

如： 将序列 **49、38、65、97、76、13、27、49‘** 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49		38	65	97	76	13	27	49'

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为：

1. low指向待划分区域首元素，high指向待划分区域尾元素;
2. t=R[low] (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;
3. high从后往前移动直到R[high].key<t.key;

如： 将序列 49、38、65、97、76、13、27、49‘ 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49		38	65	97	76	13	27	49'

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为：

1. low指向待划分区域首元素，high指向待划分区域尾元素;
2. t=R[low] (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;
3. high从后往前移动直到R[high].key<t.key;
4. R[low]=R[high], low++;

如： 将序列 49、38、65、97、76、13、27、49‘ 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49	27	38	65	97	76	13		49'

界点 low high

一次划分的具体过程示例

一次划分的具体过程为:

1. low指向待划分区域首元素，high指向待划分区域尾元素;
2. t=R[low] (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;
3. high从后往前移动直到R[high].key<t.key;
4. R[low]=R[high], low++;
5. low从前往后移动直到R[low].key>=t.key;

如： 将序列 49、38、65、97、76、13、27、49‘ 一次划分的过程为:

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49	27	38	65	97	76	13		49'

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素;
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存 到**t**中, 最后再放入最终位置) ;
3. **high**从后往前移动直到**R[high].key<t.key;**
4. **R[low]=R[high], low++;**
5. **low**从前往后移动直到**R[low].key>=t.key;**
6. **R[high]=R[low], high--;**

如： 将序列 **49**、**38**、**65**、**97**、**76**、**13**、**27**、**49'** 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49	27	38		97	76	13	65	49'

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为:

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素;
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;
3. **high**从后往前移动直到**R[high].key<t.key;**
4. **R[low]=R[high], low++;**
5. **low**从前往后移动直到**R[low].key>=t.key;**
6. **R[high]=R[low], high--;**
7. **goto 3;**

如： 将序列 **49、38、65、97、76、13、27、49‘** 一次划分的过程为:

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49	27	38		97	76	13	65	49'

界点　　　　low　　　　high

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素;
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存 到**t**中, 最后再放入最终位置) ;
3. **high**从后往前移动直到**R[high].key<t.key;**
4. **R[low]=R[high], low++;**
5. **low**从前往后移动直到**R[low].key>=t.key;**
6. **R[high]=R[low], high--;**
7. **goto 3;**

如： 将序列 **49**、**38**、**65**、**97**、**76**、**13**、**27**、**49'** 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

49	27	38	13	97	76		65	49'

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素;
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存 到**t**中, 最后再放入最终位置) ;
3. **high**从后往前移动直到**R[high].key<t.key;**
4. **R[low]=R[high], low++;**
5. **low**从前往后移动直到**R[low].key>=t.key;**
6. **R[high]=R[low], high--;**
7. **goto 3;**

如： 将序列 **49**、**38**、**65**、**97**、**76**、**13**、**27**、**49'** 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	**38**	**65**	**97**	**76**	**13**	**27**	**49'**

49	**27**	**38**	**13**		**76**	**97**	**65**	**49'**

界点

low

high

一次划分的具体过程示例

一次划分的具体过程为：

1. low指向待划分区域首元素，high指向待划分区域尾元素;
2. t=t (为了减少数据的移动, 将作为标准的元素暂存 到t中, 最后再放入最终位置) ;
3. high从后往前移动直到R[high].key<t.key;
4. R[low]=R[high], low++;
5. low从前往后移动直到R[low].key>=t.key;
6. R[high]=R[low], high--;
7. goto 3;
8. 直到low==high时， R[low]=t(即将作为标准的元素放到其最终位置) 。

如： 将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

	27	38	13	49	76	97	65	49'

low high

概括地说，一次划分就是从表的两端交替地向中间进行扫描，将小的放到左边，大的放到右边，作为标准的元素放到中间。

程序9-7 快速排序方法

```
template <class Item>
int partition(Item a[], int l, int r)        {
        int i=l-1, j=r;
//选取控制值, 这里选择当前序列的最后一个记录作为控制值
        Item t=a[r];
        for (;;){
//从序列的最左向右扫描数据，直到找到一个比控制值大的项a_i
                while (a[++i]<t);
//从序列的最右向左扫描，直到找到一个比控制值小的项a_j
                while (t<a[--j]) if (j==l) break;
//此时, a_j<t<a_i, 交换a_i和a_j, 继续上述的扫描和交换过程，直到i>=j
                if (i>=j)   break;
                swap(a[i],a[j]);
        }
//这时，i-1以左小于等于控制值t，i位置以右大于等于t
//交换a_i和a_r，并退出函数
        swap(a[i],a[r]);
        return i;
}
```

快速排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
D	A	C	A	E	T	R	U	T	T	U	R	S
A	A	C	D									
		C	D									
					R	R	S	T	T	U	T	U
					R	R						
								T	T	T	U	U
								T	T	T		
A	A	C	D	E	R	R	S	T	T	T	U	U

```
template <class Item>
void QuickSort(Item a[], int l, int r)
{
        int i;
        if (r<=l)return;                    //边界判定, 递归退出条件
        //选取控制值, 并将该值放置在队列的合适的位置上, 该位
        //                            置存放在i中.
        // 同时将小于等于该值的记录放在该位置的左边; 大于等
        //                  于的放在该位置右边
        i = partition(a,l,r);
        //递归对i左边的序列做快速排序
        QuickSort(a,l,i-1);
        //递归对i右边的序列做快速排序
        QuickSort(a,i+1,r);
}
```

9.4.2快速排序性能分析

- 时间代价：
  - 最差情况（划分基准每次均为最值）：　$O(n^2)$
  - 平均情况：　$O(n\log n)$
    - 比较次数为:2*n*ln *n*
- 空间代价(递归压栈)：$O(\log n)$
- 不稳定

9.4.3快速排序的一些改进策略

- 程序递归空间不足:
  - 使用栈
- 避免最坏情况
  - 划分基准的选择
- 加快短序列排序
  - 使用插入排序

9.4.3改进策略——栈的使用

- 程序递归需要系统栈空间:O(log n)（最坏情况O(n)），若排序序列过长，系统会因缺少空间。
- 改进:
  - 用栈将递归改非递归，栈存储待排序列范围
  - 优先把长序列压栈，则栈的深度最多为$\log_2 n$

程序**9-8** 自底向上的快速排序方法

```
inline void push_interval(stack<int> &s, int a, int b)
{       s.push(b);      s.push(a);      }

template <class Item>
void QuickSortBU(Item a[], int l, int r)
{
        stack<int> s;   int i;
        //将初始区间(l,r)压栈
        push_interval(s,l,r);
        //如果栈不空, 循环; 栈空: 说明初始区间(l,r)已经排好序, 并
                //被弹出栈外.
        while(!s.empty()){
         //获取当前栈顶存放的区间
         l=s.top();     s.pop();
         r=s.top();     s.pop();
```

```
    //如果当前区间左右边界重合, 则结束当前循环(并开始下次
          //循环)
     if (r<=l) continue;
     //调用程序9-7中的partition函数
     i=partition(a,l,r);
     //比较两个序列的长度, 将长的区间范围先压入栈中后, 再
          //将短序列的区间压进栈
     //   以保证栈的深度比较小
     if (i-l > r-i){
          push_interval(s,l,i-1);  push_interval(s,i+1,r);
     }
     else{
          push_interval(s,i+1,r); push_interval(s,l,i-1);
     }
    }
}
```

9.4.3改进策略——划分基准的选择

- 当划分基准不能将序列一分为二，则快速排序会退还成**O(n²)**。
- 改进：
  - 随机选择降低最坏发生的概率（最坏情况还是会发生）
  - 从序列中选择三项，然后选择三项中的中间项

9.4.3改进策略——短子序列

- 对于短序列，快速排序效率不如一些简单的排序方法。而插入排序在这种情况表现更好。
- 短子序列长度M：研究表明，M在5~25时，效率比M=1提高10%。经验表明，M=8~10效率会更高

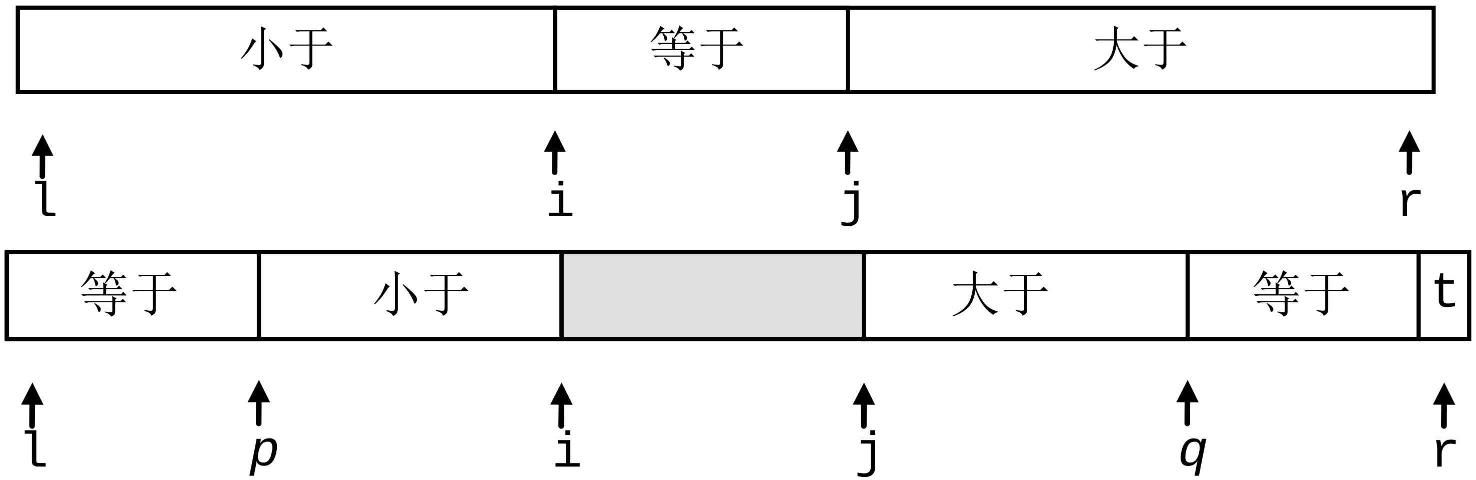
9.4.3重复值

- 当存在大量重复项时，排序算法会退化。
- 改进——三路划分（小于、等于、大于）：

① 扫描左子序列时，将与控制值相等的项放置在该子序列的最左端

② 扫描右子序列时，将与控制值相等的项放置在该子序列的最右端

③ 当i和j相遇后，循环将等于控制值的项交换到序列中央



程序9-9 快速排序的三路划分实现

```
template <class Item>
int operator==(const Item &A, const Item &B)
{
        return !less(A,B)&&!less(B,A);
}
template <class Item>
void QuickSort(Item a[], int l, int r)
{
        int k; Item v=a[r];                    //控制值选取
        if (r<=l)return;                 //递归退出条件
        int i=l-1, j=r, p=l-1, q=r;
        for(;;){
//从序列的最左向右扫描数据，直到找到一个比控制值大的项a_i
                while (a[++i]<v);
//从序列的最右向左扫描，直到找到一个比控制值小的项a_j
                while (v<a[--j]) if (j==l) break;
```

```
//此时, a_j<t<a_i, 交换a_i和a_j, 继续上述的扫描和交换过程，直到i>=j
                if (i>=j) break;
                swap(a[i],a[j]);
                //将与控制值相等的记录交换到左侧队列的最左端,
                        //同时调整p
                if (a[i]==v) {p++; swap(a[p],a[i]);}
                //将与控制值相等的记录交换到右侧队列的最右端,
                        //同时调整q
                if (v==a[j]) {q--; swap(a[q],a[j]);}
        }
        swap(a[i],a[r]);j=i-1;    i=i+1;
        //重复值交换: 将与控制值相同的记录交换到序列正中
        for (k=l; k<=p; k++,j--) swap(a[k],a[j]);
        for (k=r-1; k>=q; k--,i++)        swap(a[k],a[i]);
        // 三路划分完成, 递归对左子序列和右子序列做快速排序
        QuickSort(a,l,j);        // 左子序列三路划分快速排序
        QuickSort(a,i,r);        // 右子序列三路划分快速排序
}
```

9.5堆排序(heap sort)

- 基本思想:

9.6希尔排序(shell sort)

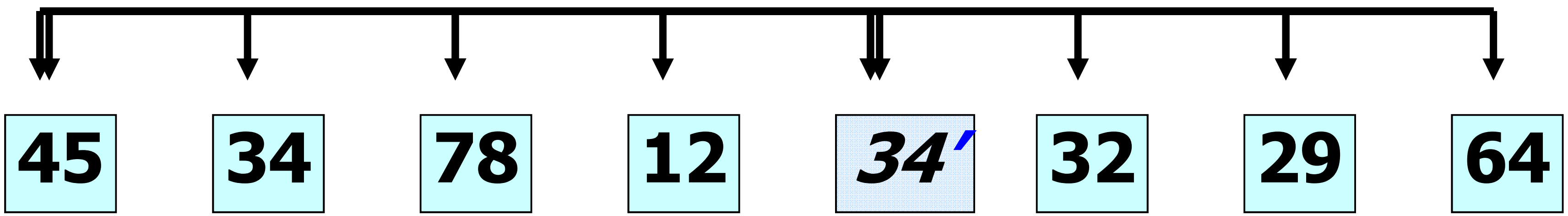
- 插入排序的两个性质:
  - 在最好情况下（序列本身是有序），时间复杂度为**O(n)**
  - 对于短序列，插入排序很有效
- 希尔排序有效地利用了插入排序的这两个性质

希尔排序的基本思想

- 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

希尔排序过程

- 先给定一组严格递减的正整数增量$d_0,d_1,\ldots,d_{t-1}$，且取$d_{t-1}=1$。
- 对于$i=0,1,\ldots,t-1$，进行下面各遍的处理：
  - 将序列分成d_i组，每组中结点的下标相差d_i
  - 对每组节点使用插入排序



希尔排序过程

46 26 22 68 48 42 36 84 66

(a) $d_0=4$

46 26 22 68 48 42 36 84 66

(b) $d_1=3$

36 26 22 46 48 42 68 84 66

(c) $d_0=1$

22 26 36 42 46 48 66 68 84

(d) 排序完成

程序9-12 希尔排序方法

```
const int d[]={4,3,1};
const int t=3;
template <class Item>
void ShellSort(Item a[], int l, int r)
{
        int h,m,i,j;    Item v;
        for (m=0;m<t;m++) //总计包括3种步长, 循环3次
        //依次对每个步长所确定的子序列进行插入排序
                for (h=d[m],i=l+h;i<=r;i++,a[j]=v){
                        for (j=i,v=a[i];j>=l+h && v<a[j-h];j-=h)
                                a[j]=a[j-h];
                        a[j]=v;
                }
}
```

希尔排序性能分析

- 时间代价：
  - 依赖于增量序列，没有确切结论

步长序列	最坏情况下复杂度
$n / 2^i$	(n^2)
$2^k - 1$	$(n^{3/2})$
$2^i 3^i$	$\mathbf{(n\log {}^2 n)}$

- 空间代价：**O(1)**
- 不稳定

9.7基数排序(radix sort)

- 基于分配的排序，与基于比较-移动的排序区别在于:
  - 不需要进行记录之间的两两比较
  - 需要知道记录的一些具体情况
- 常见基于分配的排序:
  - 桶排序
  - 基数排序

基数排序——元组的比较

- 两个d元组$(x_0,x_1,\ldots,x_{d-1})$和$(y_0,y_1,\ldots,y_{d-1})$，当且仅当$x_0=y_0, x_1=y_1, \ldots, x_i=y_i (0\le i<d-1)$，
  - 且$x_{i+1}<y_{i+1}$，$(x_0,x_1,\ldots,x_{d-1})<(y_0,y_1,\ldots,y_{d-1})$
  - 且$x_{i+1} >y_{i+1}$，$(x_0,x_1,\ldots,x_{d-1})>(y_0,y_1,\ldots, y_{d-1})$
- 也就是说从高位（最左位）起第一个不相等的位决定两个元组的大小

基数排序的定义

- 节点序列$v_0,v_1,\ldots,v_{n-1}$，排序若满足：
  - 结点v_i的关键字是由d元组$k_i=(k_i^0,k_i^1,\ldots,k_i^{d-1})$组成，$k_i^0$、$k_i^{d-1}$分别为关键字的最高位、最低位
  - 通过比较元组本身而不是节点的值
  - 排序后使得任意v_i和$v_j(i<j)$都有$(k_i^0,k_i^1,\ldots,k_i^{d-1})\leq(k_j^0,k_j^1,\ldots,k_j^{d-1})$

则这样的排序为基数排序

基数排序的方法

- 按照元组元素从高位还是低位开始排序分两种：
  - 最高位优先（**most significant digit first**，缩写**MSD**）
  - 最低位优先（**Least significant digit first**，缩写**LSD**）

MSD基数排序的方法

- 先对高位k^0进行排序，将若干个序列，每个序列的k^0相同
- 然后对每个序列再按次高位k^1进行排序，分成更小的序列
- 依次重复，直到对k^{d-1}排序后，分成最小的序列，每个序列内含有相同的排序码(k^0，k^1，…，k^{d-1})
- 最后将所有的序列依次连接在一起，成为一个有序序列
- 这是一个分、分、…、分、收的过程

LSD基数排序的方法

- 从最低位k^{d-1}开始排序
- 对于排好的整个序列再用次低位k^{d-2}排序；
- 依次重复，直至对最高位k^{0}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

基数排序的实现

- 基于顺序存储结构（数组）
  - 速度快
  - 通用
- 基于链式存储结构：
  - 避免空间浪费

基数排序的基于数组实现

- 选择256（一字节）作为基数
- 利用count数组记录每个桶中元素个数：
- 待排元素个数少于M时使用插入排序

LSD基数排序的基于数组实现示例

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：

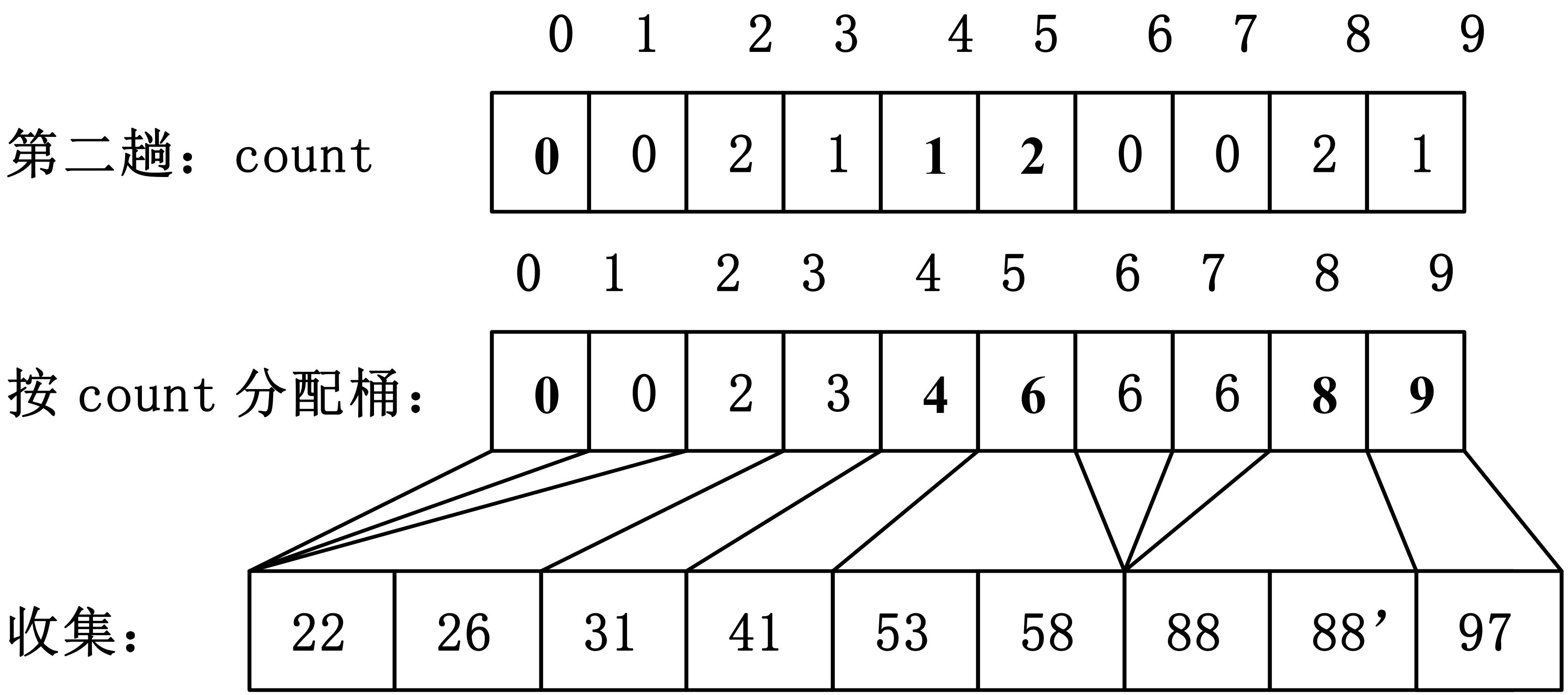
0	1	2	3	4	5	6	7	8	9
0	2	3	4	**4**	**4**	5	6	**8**	**9**

收集：

41	31	22	53	26	97	88	88’	59

(a) 第一趟分配个位

LSD基数排序的基于数组实现示例



最终排序结果：22 26 31 41 53 59 88 88’ 97

(b) 第二趟分配十位

程序9-13 MSD基数排序

```
#define bin(A) l+count[A]
const int radix=256;    // 基数
const int M=3;
const int maxN=1000;
const int bytes=4;
template <typename Item>
inline int getDigit(Item A, int B)
 {    A =A>>((bytes-B-1)*8);     return A%radix; }
inline int getDigit(Item &A, int B) {     return A+B; }
template <typename Item>
void RadixSort_MSD(Item a[], int l, int r, int d)
{
            int i,j,count[radix+1];
            static Item aux[maxN];
            if (d>bytes)      return;
```

```
    if (r-l+1<=M) {InsertionSort(a,l,r); return;}
    // 统计各桶元素个数
    for (j=0;j<radix;j++)     count[j]=0;
    for (i=l;i<=r;i++)              count[getDigit(a[i],d)+1]++;
    // 安排各桶元素的放置位置
    for (j=1;j<radix;j++)     count[j]+=count[j-1];
     // 将各桶元素按桶的顺序放置在辅助数组aux中
    for (i=l;i<=r;i++)              aux[count[getDigit(a[i],d)]++]=a[i];
     // 将aux数组写回原数组
    for (i=l;i<=r;i++)              a[i]=aux[i-l];
    // 递归调用MSD基数排序
    RadixSort_MSD(a,l,bin(0)-1,d+1);
    for (j=0;j<radix-1;j++)
            RadixSort_MSD(a,bin(j),bin(j+1)-1,d+1);
}
```

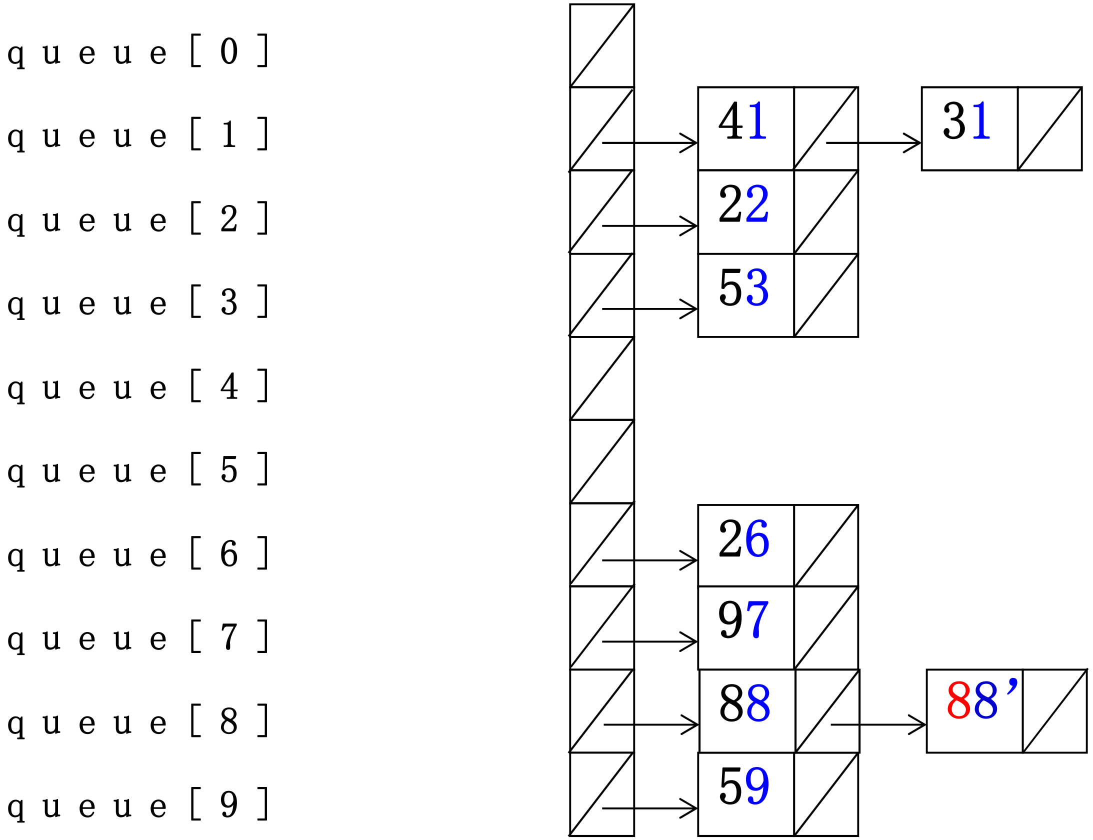
程序**9-14 LSD**基数排序

```
template <typename Item>
void RadixSort_LSD(Item a[], int l, int r){
        static Item aux[maxN];
        int i,j,d,count[radix+1];
         if (r-l+1<=M) {InsertionSort(a,l,r); return;}
        for (d=bytes-1;d>=0;d--){
                // 统计每个桶中元素个数
                for (j=0;j<radix;j++)         count[j]=0;
                for (i=l;i<=r;i++)
                        count[getDigit(a[i],d)+1]++;
                // 安排元素的安放位置
                for (j=1;j<radix;j++)
                        count[j]+=count[j-1];
                 // 将所有元素按所在的桶分配到辅助数组aux中
                for (i=l;i<=r;i++)
                        aux[count[getDigit(a[i],d)]++]=a[i];
                // 将辅助数组aux中的元素收集到原数组中
                for (i=l;i<=r;i++)
                        a[i]=aux[i-l];
        }
}
```

基数排序的基于链式存储结构实现的示例

97	53	88	59	26	41	88’	31	22

（a）初 始 链 表 内 容

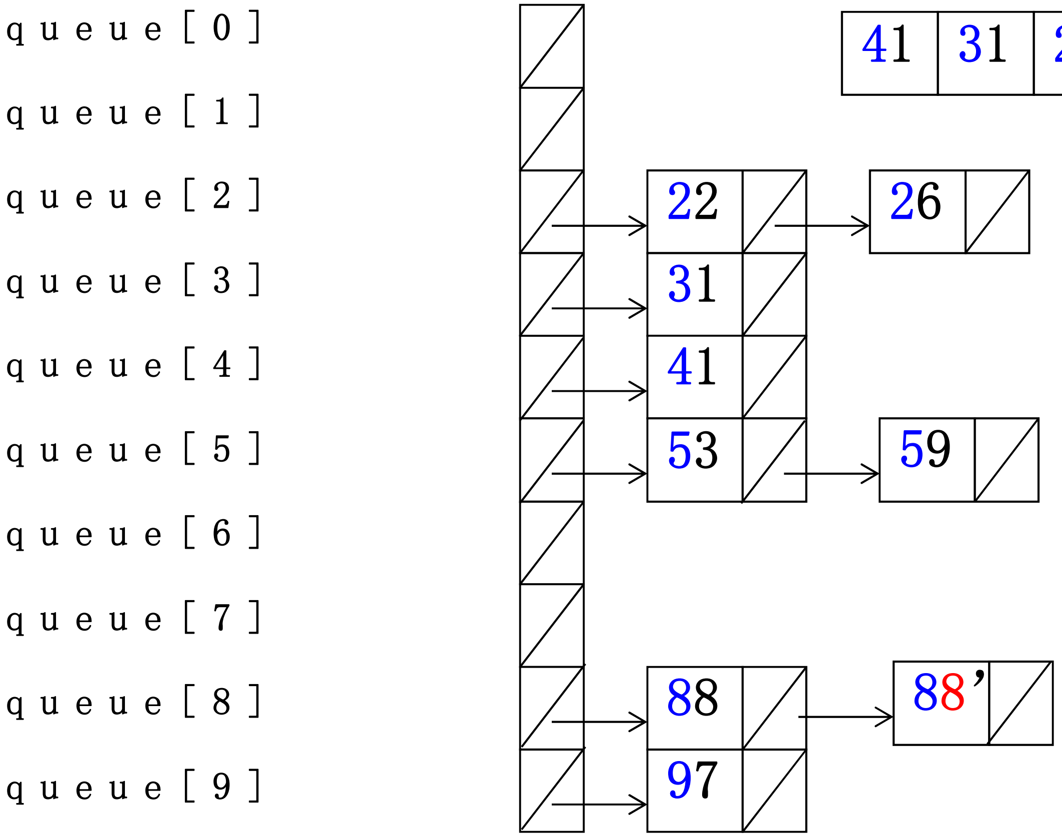


(b)第一趟分配

(c) 第一趟收集

41	31	22	53	26	97	88	88’	59

41	31	22	53	26	97	88	88’	59



（d）第 二 趟 分 配

(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

基数排序性能分析

- 时间代价：
  - 基数排序算法对数据进行d趟扫描，每趟需时间O(n+radix)。因此总的计算时间为O(d(n+radix))。当n较大或d较小时，这种方法较为节省时间
- 空间代价：**O(n)**
- 稳定
- 对整数和字符串，基数排序的效率优于快速排序

9.8内部排序方法的比较

- 基于“比较-分配”的排序时间复杂度理论下界$\Omega(n \cdot \log n)$
  - 比较树证明
- 简单排序：插入排序、冒泡排序、选择排序：
  - 不需要额外辅助空间
  - 平均时间复杂度均为$O(n^2)$
  - 在元素个数较少时，插入排序性能好，比大多排序都快

9.8内部排序方法的比较

- 高级排序：快速排序、堆排序、归并排序、希尔排序：
  - 快速排序：平均性能最好、最坏情况时间复杂度$O(n^2)$
  - 堆排序：平均性能仅次于快速排序、不需额外的存储空间
  - 归并排序：稳定的排序方法，没有最坏情况。
  - 希尔排序：算法时间复杂度有待研究

9.8内部排序方法的比较

- 基于分配的排序：桶排序、基数排序
  - 桶排序：待排元素均匀分配时，时间复杂度为O(n)
  - 基数排序：
    - 具有线性时间复杂度
    - 对整数与字符串有很高效率
    - 限于具体数据，效率与通用性不如基于“比较-交换”的排序

9.8内部排序方法的比较

排序方法	平均时间	最坏时间	最佳时间	辅助空间	稳定性
插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)(改进)	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
归并排序	O(n*logn)	O(n*logn)	O(n*logn)	O(n)	稳定
快速排序	O(n*logn)	$O(n^2)$	O(n*logn)	O(logn)	不稳定
堆排序	O(n*logn)	O(n*logn)	O(n*logn)	O(1)	不稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

选择排序的方法

- 当待排序记录数n较大时，若要求排序稳定，则采用归并排序。
- 当待排序记录数n较大，关键字分布随机，而且不要求稳定时，可采用快速排序；
- 当待排序记录数n较大，关键字会出现正、逆序情形，可采用堆排序（或归并排序）。
- 当待排序记录数n较小，记录已接近有序或随机分布时，可采用直接插入排序。

随堂练习

例1：设有5000个无序的元素，希望用最快速度挑出其中前10个最大的元素。在快速排序、堆排序、归并排序、基数排序和希尔排序方法中，采用哪一种方法最好？为什么？

例2：对由n个元素组成的线性表进行快速排序时，所需进行的比较次数与这n个元素的初始排列有关。

(1) 当n=7时，在最好情况下需进行多少次比较？

(2) 当n=7时，给出一个最好情况的初始排列的实例。

(3) 当n=7时，在最坏情况下需进行多少次比较？

(4) 当n=7时，给出一个最坏情况的初始排列的实例。

例3：某个待排序的序列是一个可变长度的字符串序列，这些字符串一个接一个地存储于唯一的字符数组中。请改写快速排序算法，对这个字符串序列进行排序。

例1：设有5000个无序的元素，希望用最快速度挑出其中前10个最大的元素。在快速排序、堆排序、归并排序、基数排序和希尔排序方法中，采用哪一种方法最好？为什么？

所列几种排序方法的速度都很快，但快速排序、归并排序、基数排序和希尔排序都是在排序结束之后才能确定数据元素的全部顺序，而无法知道排序过程中部分元素的有序性。但堆排序则每次输出一个最小（或最大）的元素，然后对堆进行调整，保证堆顶的元素是未排序元素中的最小（或最大）的。因此，选取前10个最大元素采用堆排序方法最好。

例2：对由n个元素组成的线性表进行快速排序时，所需进行的比较次数与这n个元素的初始排列有关。

(1) 在最好情况下，每一趟快速排序后均能划分出左、右两个相等的区间，即设线性表的长度$n=2k-1$，则第一趟快速排序后划分得到两个长度均为n/2的子表，第二快速排序后划分得到4个长度为n/4的 子表，以此类推，总共需进行$k=\log_2(n+1)$遍划分，即子表长度为1时排序完毕。因此，当n=7时k=3，也即在最好情况下第一个元素由两头向中间扫描到正中位置，即需与其余6个元素都进行比较后找到最终存储位置，因此需要比较6次。第二趟分别对左、由两个子表（长度均为3，即k=2）进行排序，与第一趟类似，需与子表中其余2个元素进行比较后找到其最终存储位置，也即两个子表共需比较4次，并且继续划分出的每个子表长度均为1，即排序完毕。故总共需比较10次。

例2：对由n个元素组成的线性表进行快速排序时，所需进行的比较次数与这n个元素的初始排列有关。

(2) 由(1)所知，每趟排序都应使第一个元素存储于表的正中位置，因此最好的初始排列的例子为：4, 7, 5, 6, 3, 1, 2。

(3) 快速排序最坏的情况是，每趟用来划分的基准元素总是定位于表的第一位置或最后一个位置，这样划分的左、右子表一个长度为0，另一个仅是原表长减1。这样，快速排序的效率蜕化为冒泡排序，其时间复杂度为$O(n^2)$，即比较次数为6+5+4+3+2+1=21次。

(4) 由(3)可知，快速排序最坏的情况是初始序列有序。所以，当n=7时，最坏情况的初始排列的例子为：1, 2, 3, 4, 5, 6, 7或者7, 6, 5, 4, 3, 2, 1。

9.9 stl的排序

Sort	对给定区间的所有元素进行排序	优化的快速排序
stable_sort	对给定区间的所有元素进行稳定排序	归并排序，若内存不足，则时间复杂度为O($nlog^2$ n)
partial_sort	对给定区间的所有元素部分排序	堆排序求前m个元素
partial_sort_copy	对给定区间复制并排序	
nth_element	找出给定区间的某个位置对应的元素	快速排序的partition方法
is_sorted	判断一个区间是否已经排好序	
partition	使得符合某个条件的元素放在前面	类似快速排序的partition方法
stable_partition	相对稳定的使得符合某个条件的元素放在前面	

常见题型

- 基本概念
- 各种排序算法
  - 稳定性，时间复杂度，空间复杂度
  - 给定一些特殊要求，设计排序算法

- **课程习题**
  - 笔做题——**9.14, 9.16, 9.21**
    （以作业形式提交）
  - 上机题——**9.15, 9.17**
  - 思考题——剩余的其它习题