



# Data Structures

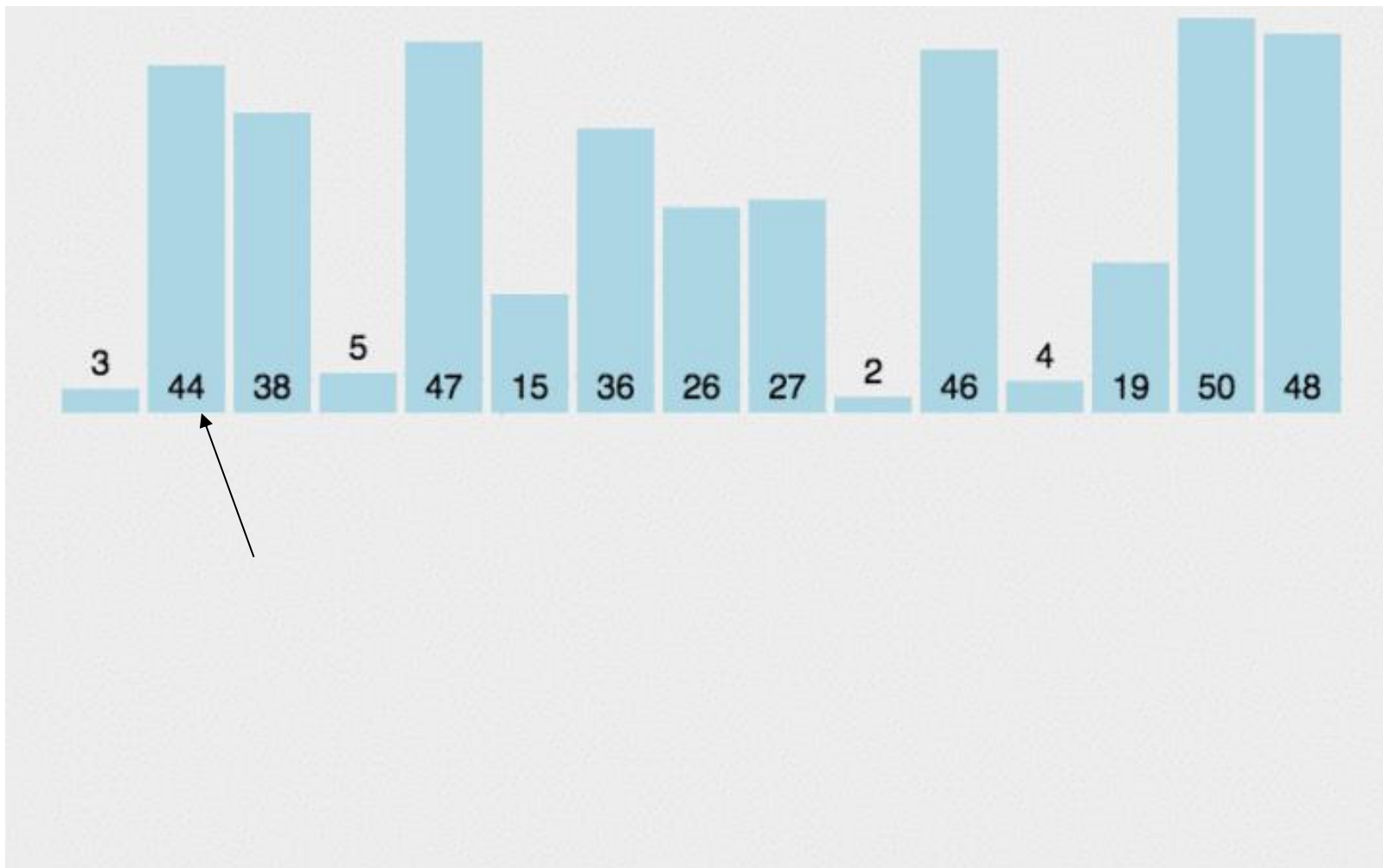
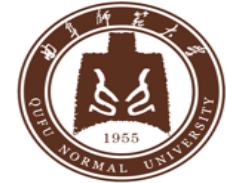
Ch8

# 排序 Sort

2024 年 12 月13 日

- ➡ 8.1 概述
- ➡ 8.2 插入排序：直接插入排序、希尔排序
- ➡ **8.3 交换排序：起泡、快速排序**
- ➡ 8.4 选择排序：简单选择、堆排序
- ➡ 8.5 归并排序：二路归并排序
- ➡ 8.6 各种排序方法比较
- ➡ 8.7 扩展与提高

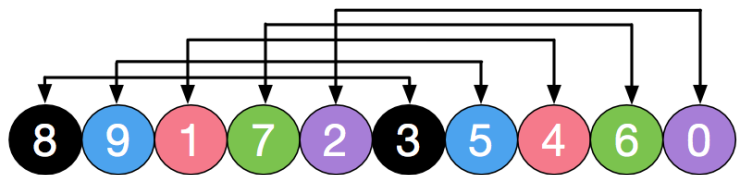
# 复习：直接插入排序



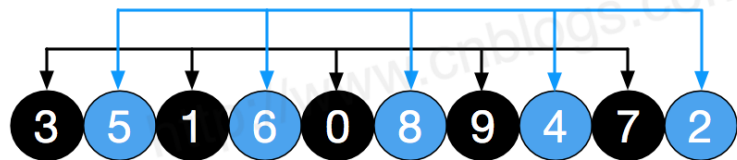
原始数组 以下数据元素颜色相同为一组



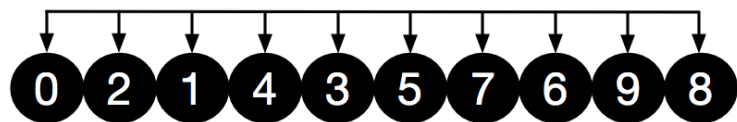
初始增量  $gap=length/2=5$ , 意味着整个数组被分为5组, [8,3] [9,5] [1,4] [7,6] [2,0]



对这5组分别进行直接插入排序, 结果如下, 可以看到, 像3, 5, 6这些小元素都被调到前面了, 然后缩小增量  $gap=5/2=2$ , 数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序, 结果如下, 可以看到, 此时整个数组的有序程度更进一步啦。再缩小增量  $gap=2/2=1$ , 此时, 整个数组为1组[0,2,1,4,3,5,7,6,9,8], 如下

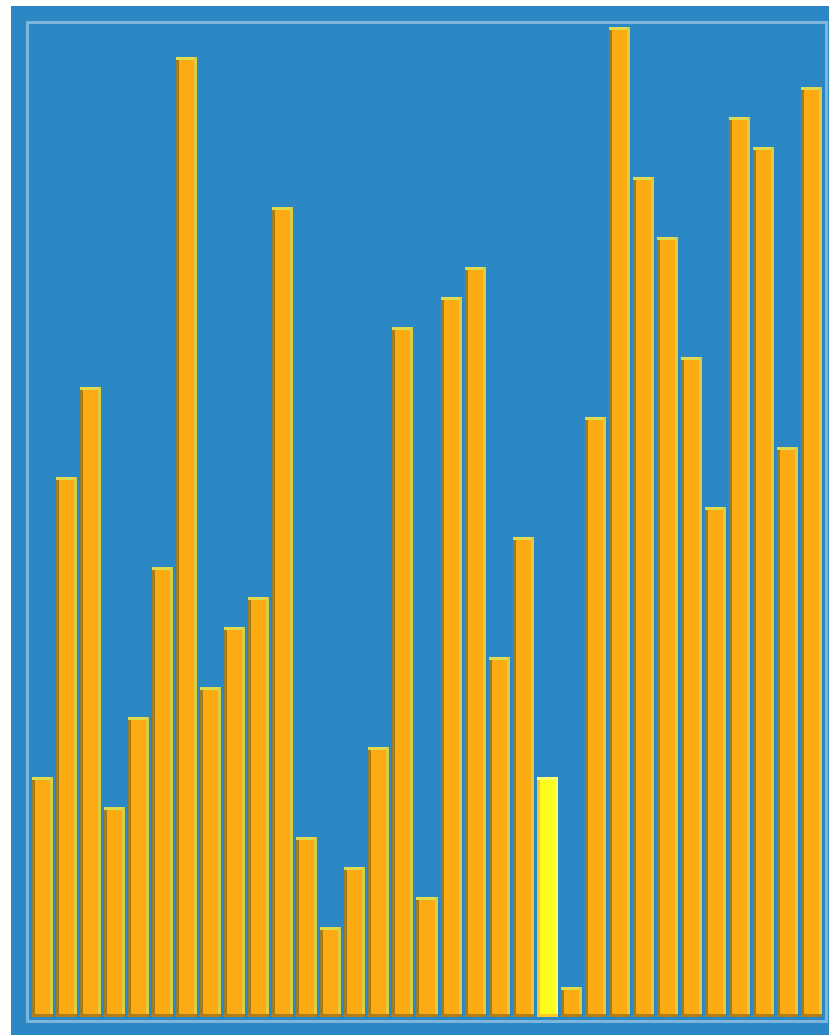


经过上面的“宏观调控”, 整个数组的有序化程度成果喜人。

此时, 仅仅需要对以上数列简单微调, 无需大量移动操作即可完成整个数组的排序。



# 复习：希尔排序



## 8.3 交换排序

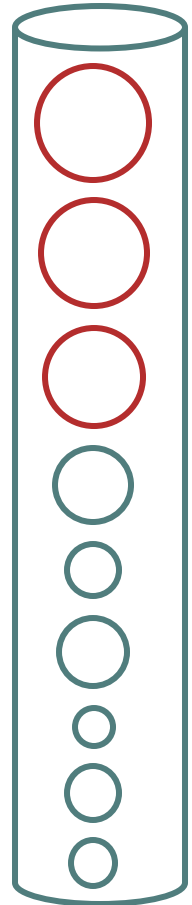
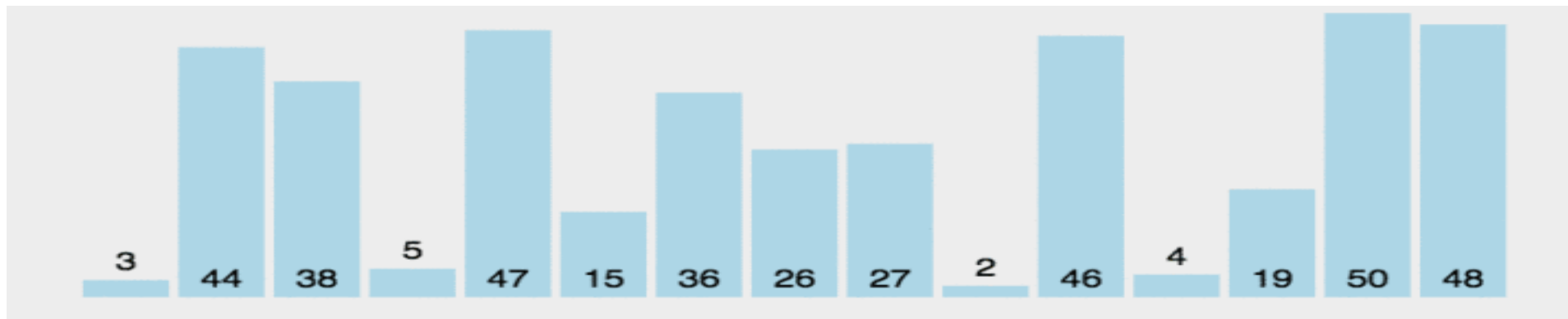
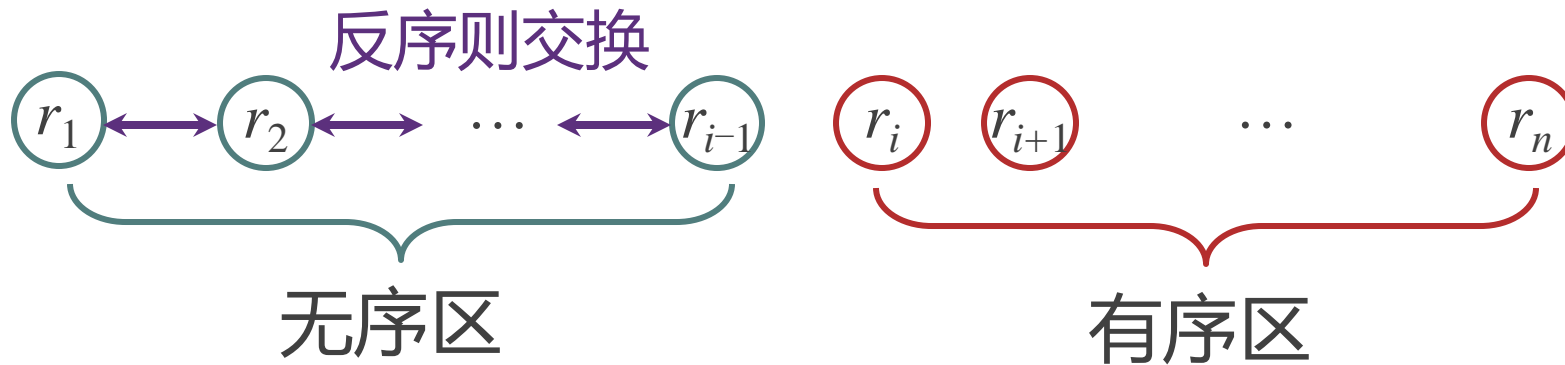
### 8-3-1 起泡排序

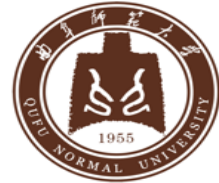
## 8.3 交换排序

### 8-3-1 起泡排序

#### 1. 起泡排序基本思想

📢 起泡排序的基本思想：两两比较**相邻**记录，如果**反序**则交换，直到没有反序的记录为止。





## 2. 起泡排序实例

**基本思想：**每趟不断将记录两两比较，并按“前小后大”规则交换

21, 25, 49, 25\*, 16, 08

21, 25, 25\*, 16, 08, 49

21, 25, 16, 08, 25\*, 49

21, 16, 08, 25, 25\*, 49

16, 08, 21, 25, 25\*, 49



08, 16, 21, 25, 25\*, 49

```
for(j=1;j<=n-1;j++)  
    for(i=0;i<n-j;i++)  
        if(a[i]>a[i+1])  
            {t=a[i];a[i]=a[i+1];a[i+1]=t;}//交换
```

## 8.3 交换排序

### 8-3-1 起泡排序

#### 2. 起泡排序实例

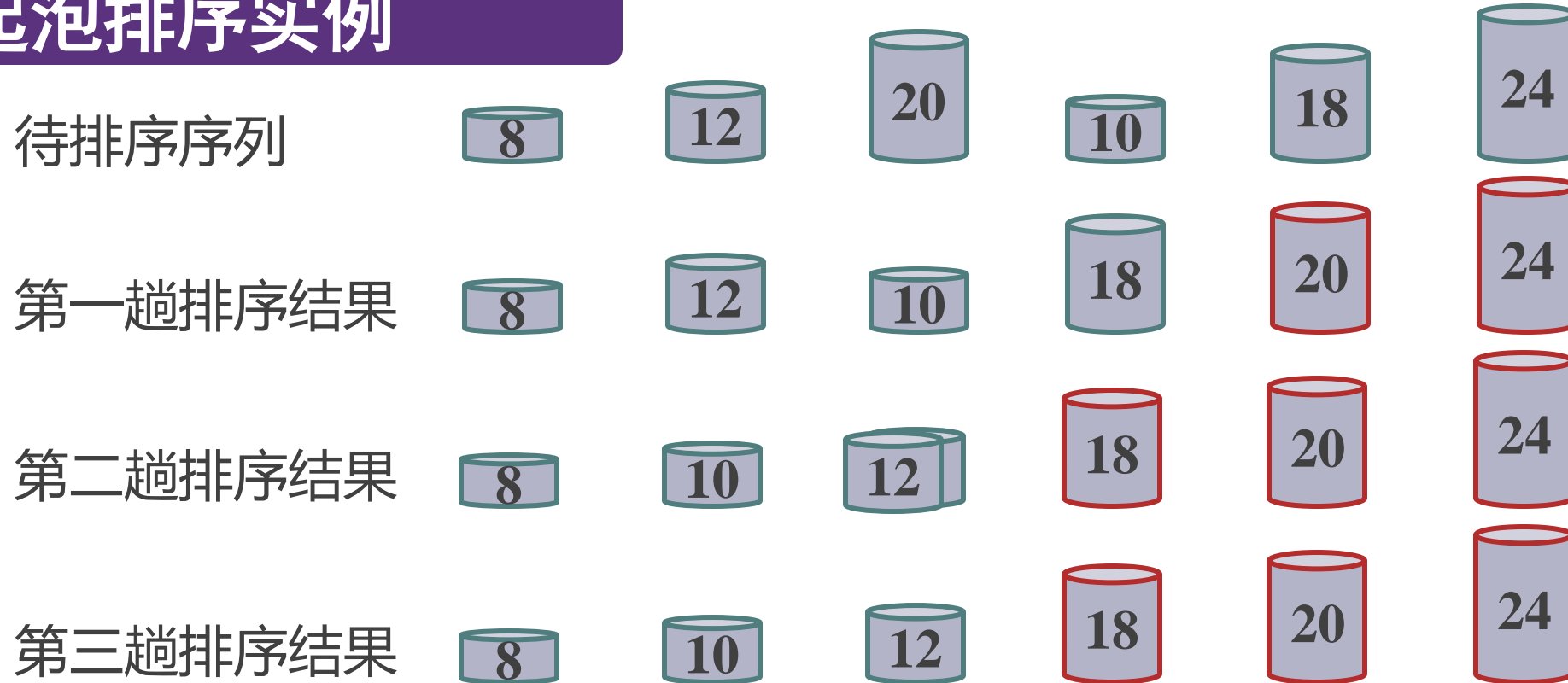
待排序序列	8	12	20	10	18	24		
第一趟排序结果	8	12	10	18	20	24	第二趟排序有必要吗？	第五趟排序有必要吗？
第二趟排序结果	8	10	12	18	20	24		
第三趟排序结果	8	10	12	18	20	24		
第四趟排序结果	8	10	12	18	20	24		
第五趟排序结果	8	10	12	18	20	24		



## 8.3 交换排序

### 8-3-1 起泡排序

#### 2. 起泡排序实例



一趟起泡排序可以确定多个记录的最终位置



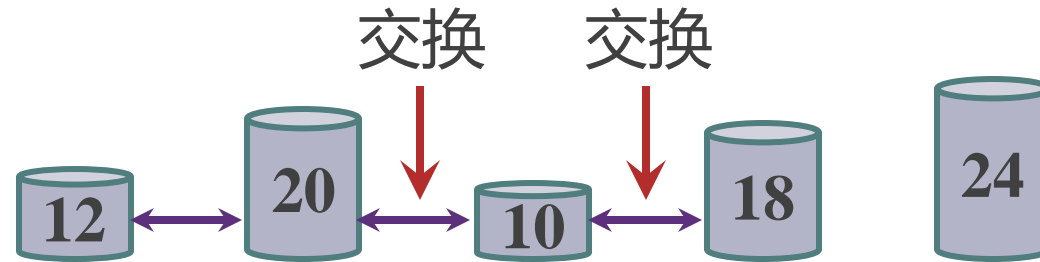
一趟起泡排序没有记录交换，则结束排序过程

## 8.3 交换排序

### 8-3-1 起泡排序

#### 3. 关键问题

待排序序列



第一趟排序结果



解决方法：设置变量exchange记载交换的位置，一趟排序后exchange记载的就是最后交换的位置，从exchange之后的记录不参加下一趟排序。

算法描述：

```
if (data[j] > data[j+1]){  
    temp = r[j]; r[j] = r[j+1]; r[j+1] = temp;  
    exchange = j;  
}
```

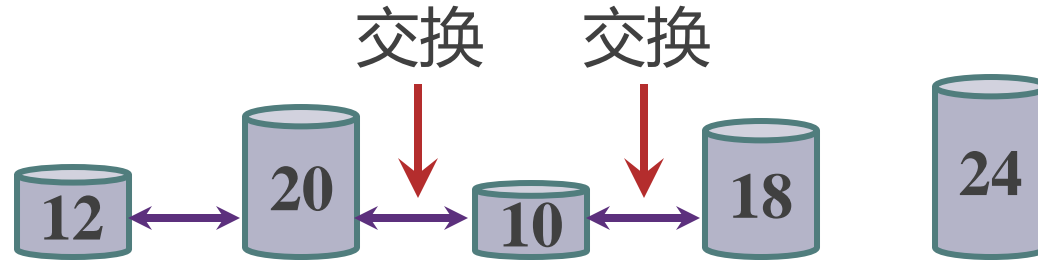
如果有多个记录位于最终位置，如何不参加下一趟排序？

## 8.3 交换排序

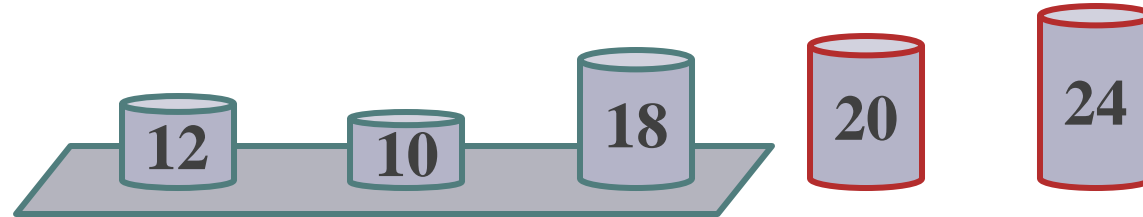
### 8-3-1 起泡排序

#### 3. 关键问题

待排序序列



第一趟排序结果



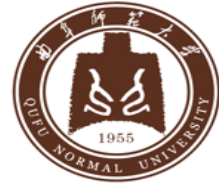
解决方法：设置变量bound表示一趟起泡排序的范围 $[1, \text{bound}]$ ，并且bound与上一趟起泡排序的最后交换的位置exchange之间的关系是 $\text{bound} = \text{exchange}$ 。

算法描述：

```
bound = exchange; exchange = 0;
for (j = 0; j < bound; j++)
    if (data[j] > data[j+1]){
        temp = r[j]; r[j] = r[j+1]; r[j+1] = temp;
        exchange = j;
    }
```



下一趟排序的范围是多少？



#### 4. 算法性能

```
void Sort :: BubbleSort( )
{
    int j, exchange, bound, temp;
    exchange = length - 1;          //第一趟起泡排序的区间是[0~length-1]
    while (exchange != 0)
    {
        bound = exchange; exchange = 0;
        for (j = 0; j < bound; j++) //一趟起泡排序的区间是[0~bound]
            if (data[j] > data[j+1]) {
                temp = data[j]; data[j] = data[j+1]; data[j+1] = temp;
                exchange = j;        //记载每一次记录交换的位置
            }
    }
}
```

## 8.3 交换排序

### 8-3-1 起泡排序

#### 4. 算法性能

```
void Sort :: BubbleSort( )
```

```
{
```

```
    int j, exchange, bound, temp;
```

```
    exchange = length - 1;
```

```
    while (exchange != 0)
```

```
    {
```

```
        bound = exchange; exchange = 0;
```

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

```
            if (data[j] > data[j+1]) {
```

```
                temp = data[j]; data[j] = data[j+1]; data[j+1] = temp;
```

```
                exchange = j;            }
```

```
    }
```

```
}
```



比较语句？ 执行次数？



移动语句？ 执行次数？



取决于待排序序列的初始状态

## 8.3 交换排序



### 8-3-1 起泡排序

#### 4. 算法性能

📜 最好情况：正序  $O(n)$

📎 比较次数： $n-1$ 次

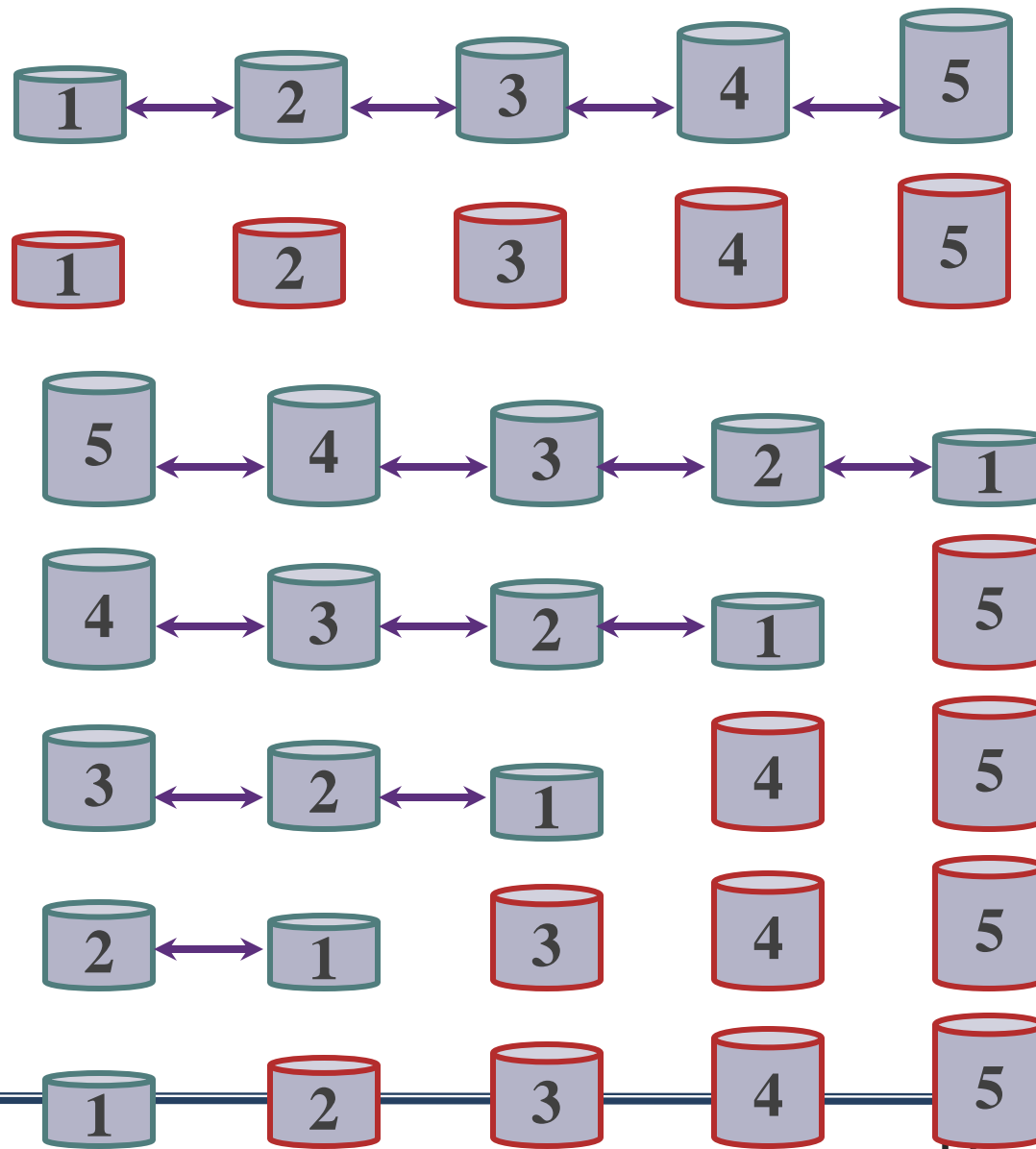
📎 移动次数：0次

📜 最坏情况：逆序  $O(n^2)$

📎 比较次数： $\sum_{i=1}^{n-1} n-i = \frac{n(n-1)}{2}$  次

📎 移动次数： $\sum_{i=1}^{n-1} 3(n-i) = \frac{3n(n-1)}{2}$  次

📜 平均情况：随机排列， $O(n^2)$



## 8.3 交换排序

### 8-3-1 起泡排序

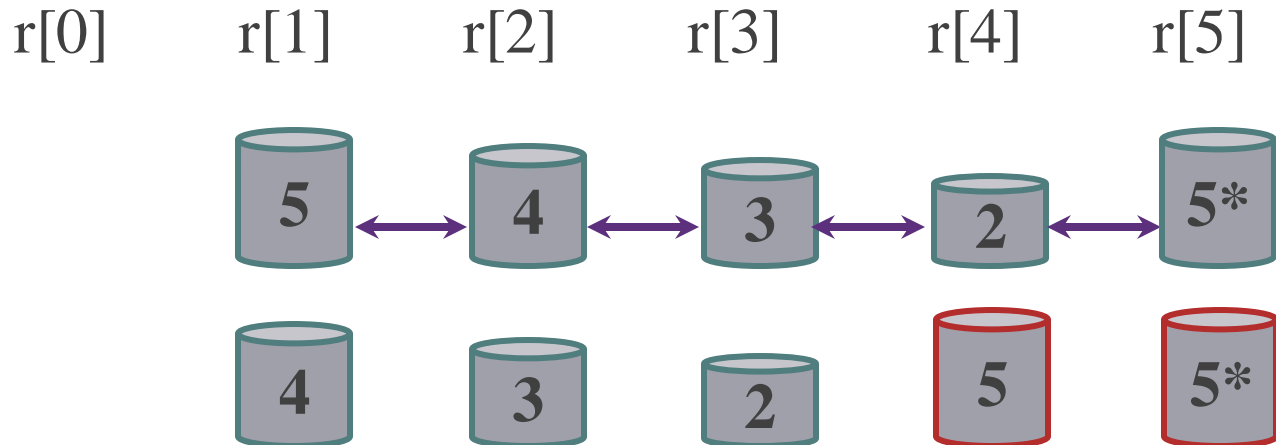


#### 4. 算法性能



$r[0]$ 作用是什么?

暂存单元



空间性能:  $O(1)$



稳定性: 稳定

```
if (data[j] > data[j+1]) {  
    temp = r[j]; r[j] = r[j+1]; r[j+1] = temp  
    exchange = j;  
}
```

## 8.3 交换排序

### 8-3-2 快速排序



## 8.3 交换排序

### 8-3-2 快速排序

#### 起泡排序的改进思路

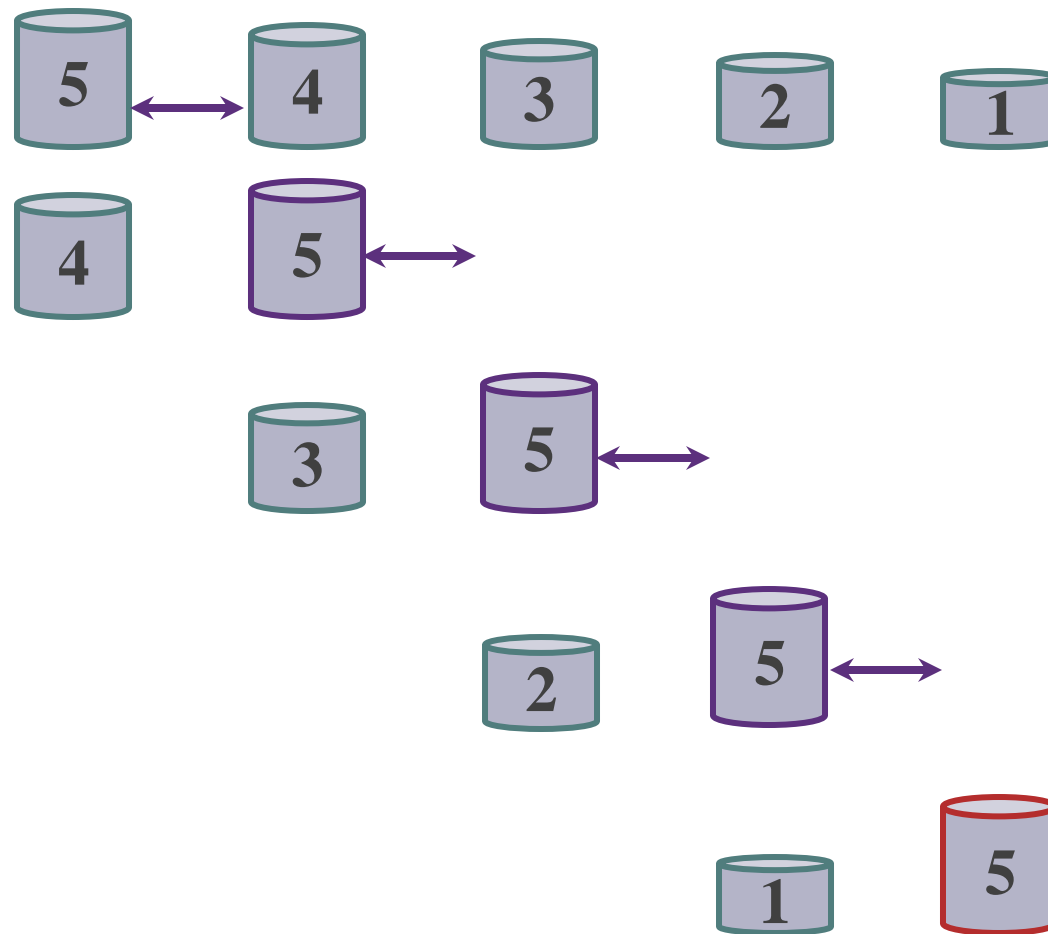
记录的比较在相邻单元中进行



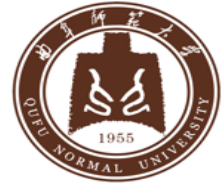
每次交换只能右移一个单元




总的比较次数和移动次数较多

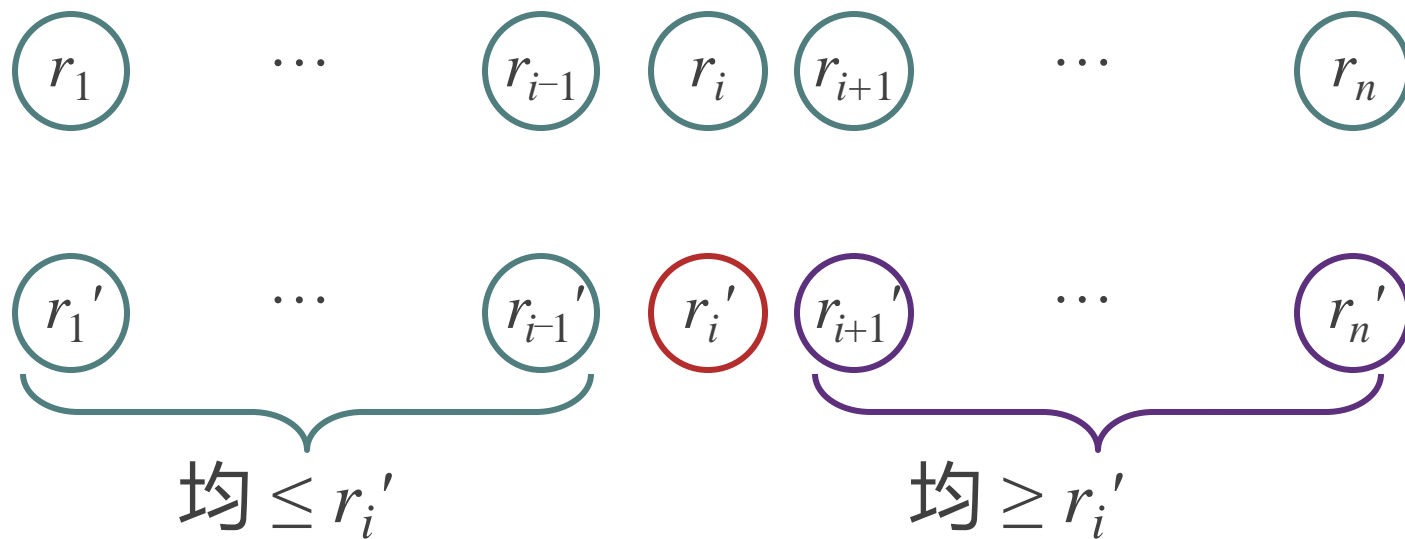


较大记录从前面直接移到后面，较小记录从后面直接移到前面？



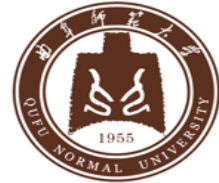
#### 1. 快速排序

 快速排序的基本思想：选一个轴值，将待排序记录划分成两部分，左侧记录均小于或等于轴值，右侧记录均大于或等于轴值，然后分别对这两部分重复上述过程，直到整个序列有序。



## 8.3 交换排序

### 8-3-2 快速排序



#### 运行实例

待排序序列



第一趟排序结果



第二趟排序结果

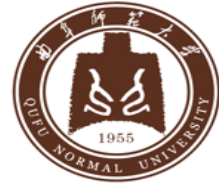


第三趟排序结果



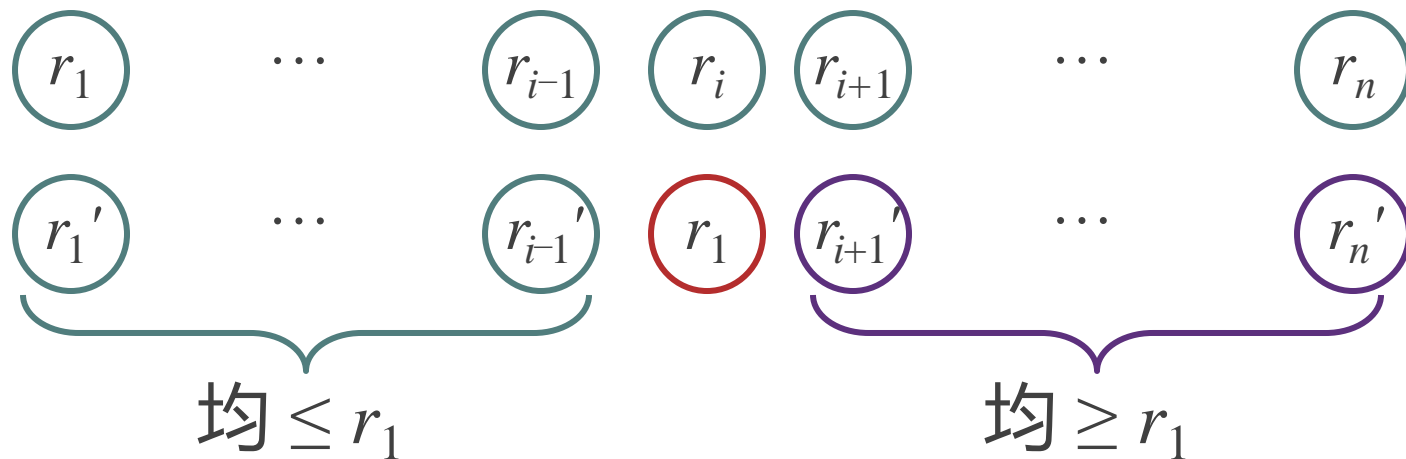
最终排序结果





#### 1. 快速排序

✦ 一次划分：以轴值为基准将无序序列划分为两部分，即：

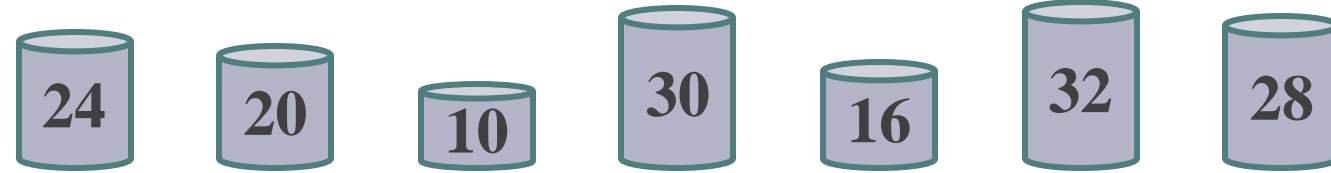


## 8.3 交换排序

### 8-3-2 快速排序

#### 2. 关键问题

待排序序列



一次划分结果



解决方法:

- (1) 第一个记录;
- (2) 随机选取;
- (3) 比较三个记录取值居中者;

简单起见，取第一个记录作为轴值

决定排序的时间性能

决定两个子序列的长度



如何选择轴值——比较的基准？选取不同轴值有什么后果？



## 2. 关键问题

待排序序列



一次划分结果



减少了总的比较次数和移动次数

较小的记录一次就能从后面移到前面（较大的记录？）

记录的比较和移动从两端向中间进行



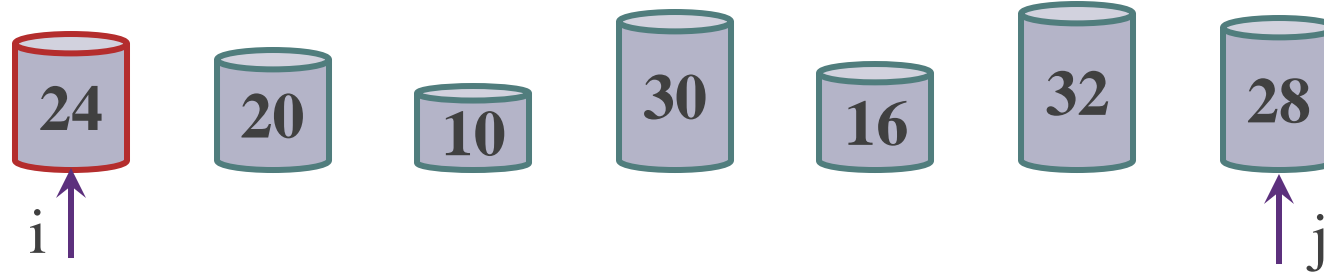
如何实现一次划分——较大的记录移到后面，较小记录移到前面？

## 8.3 交换排序

### 8-3-2 快速排序

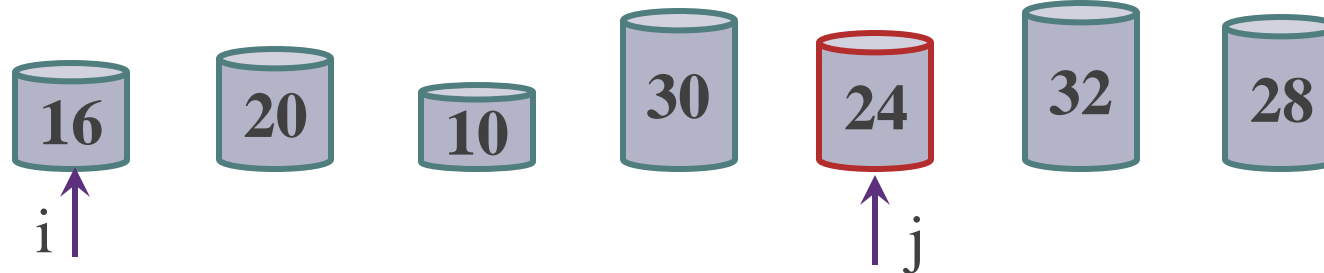
#### 3. 一次划分

待排序序列



$j$  从后向前扫描  
直到  $r[j] < r[i]$

交换  $r[j]$  和  $r[i]$   
 $i++$



## 8.3 交换排序

### 8-3-2 快速排序



#### 3. 一次划分

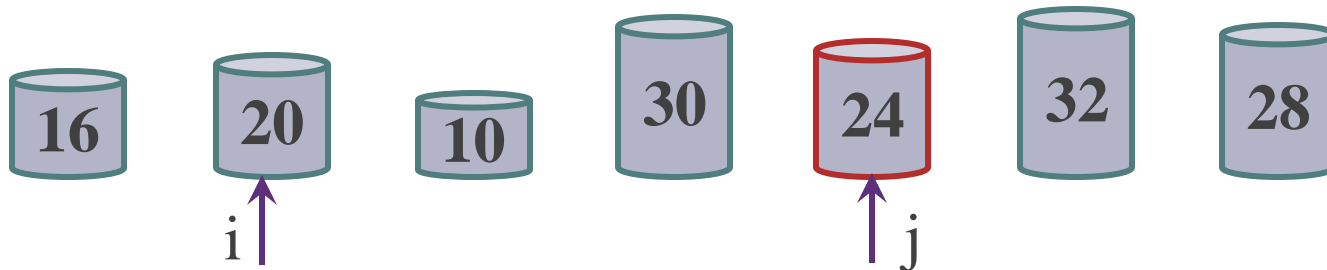
待排序序列



$j$  从后向前扫描  
直到  $r[j] < r[i]$

交换  $r[j]$  和  $r[i]$

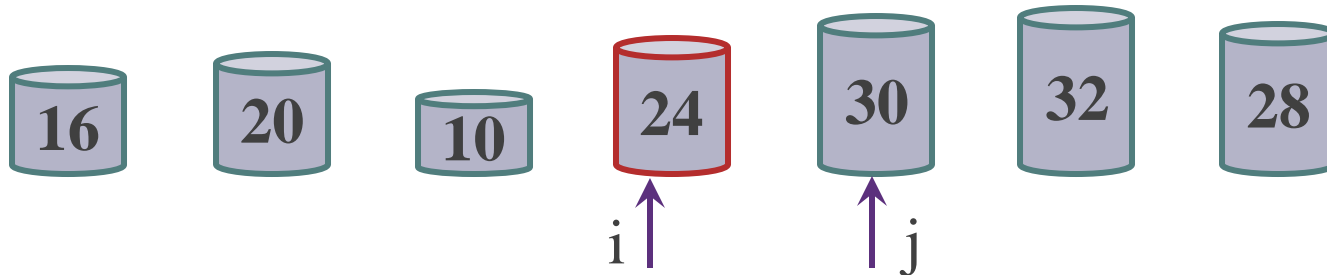
$i++$



$i$  从前向后扫描  
直到  $r[j] < r[i]$

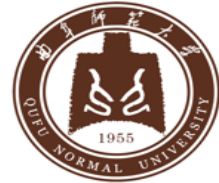
交换  $r[j]$  和  $r[i]$

$j--$



重复上述过程，直到  $i$  等于  $j$





### 3. 一次划分实现

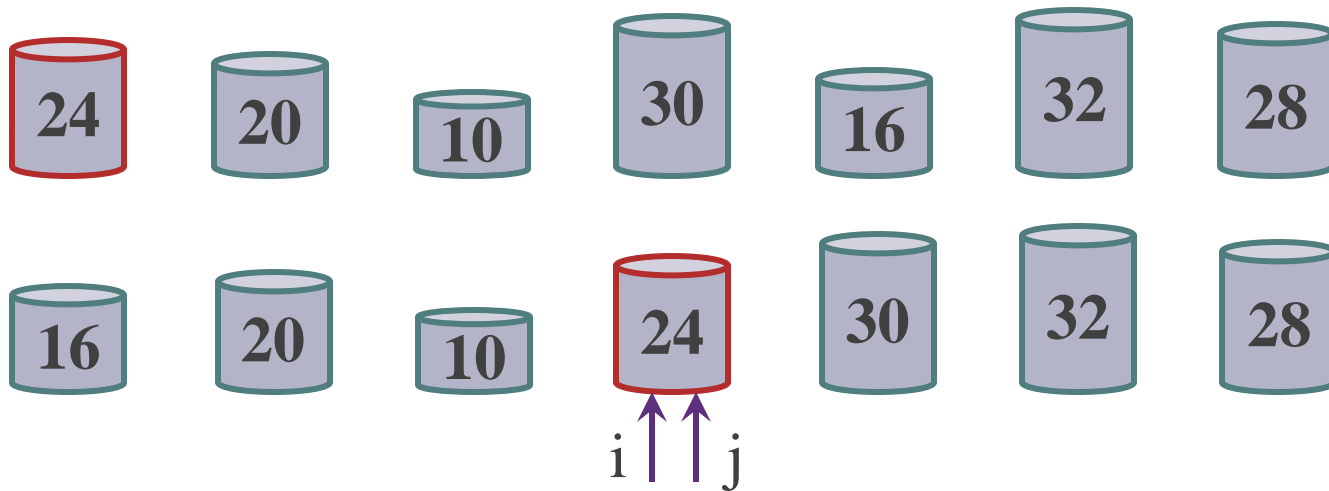
```
int Sort :: Partition(int first, int last)
```

```
{  
    int i = first, j = last, temp;  
    while (i < j)  
    {
```



为什么设置形参first和last? 表示什么?

表示待划分区间[first, last], 是变化的



```
}
```

```
return i;
```

```
}
```

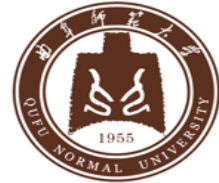
/\*i为轴值记录的最终位置\*/



### 3. 一次划分实现

```
int Sort :: Partition(int first, int last)
{
    int i = first, j = last, temp;
    while (i < j)
    {
        while (i < j && data[i] <= data[j]) j--;           /*右侧扫描*/
        if (i < j) {
            temp = data[i]; data[i] = data[j]; data[j] = temp; i++;
        }
        while (i < j && data[i] <= data[j]) i++;           /*左侧扫描*/
        if (i < j) {
            temp = data[i]; data[i] = data[j]; data[j] = temp; j--;
        }
    }
    return i;
}
```

/\*i为轴值记录的最终位置\*/



### 3. 一次划分实现

```
int Sort :: Partition(int first, int last)
```

```
{
```

```
    int i = first, j = last, temp;
```

```
    while (i < j)
```

```
    {
```

```
        while (i < j && data[i] <= data[j]) j--;
```

```
        if (i < j) {
```

```
            temp = data[i]; data[i] = data[j]; data[j] = temp; i++;
```

```
        }
```

```
        while (i < j && data[i] <= data[j]) i++;
```

```
        if (i < j) {
```

```
            temp = data[i]; data[i] = data[j]; data[j] = temp; j--;
```

```
        }
```

```
    }
```

```
    return i;
```

```
}
```



时间复杂度是多少？

下标 i 和 j 共同将数组扫描一遍， $O(n)$

/\*右侧扫描\*/

/\*左侧扫描\*/

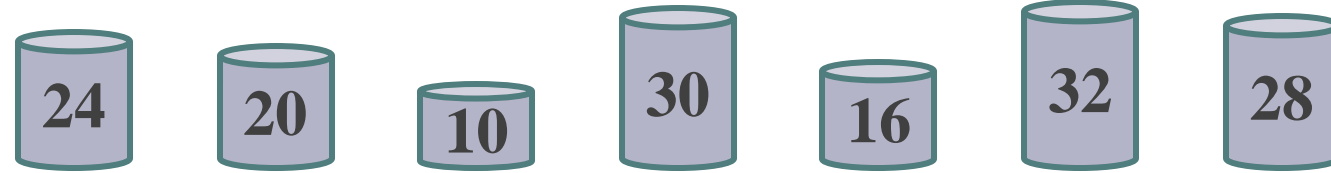
/\*i为轴值记录的最终位置\*/

## 8.3 交换排序

### 8-3-2 快速排序

#### 4. 关键问题

待排序序列



一次划分结果



解决方法:

递归执行快速排序

```
void Sort :: QuickSort (int first, int last )  
{  
    int pivot = Partition (first, last);  
    QuickSort (first, pivot-1);  
    QuickSort (pivot+1, last );  
}
```

算法描述:



如何处理一次划分得到的两个待排序子序列?

## 8.3 交换排序

### 8-3-2 快速排序

#### 运行实例

待排序序列



第一趟排序结果



第二趟排序结果



解决方法:

若待排序序列只有一个记录，即待划分区间长度为 1

`if (first == last) return;`



递归何时结束?



## 8.3 交换排序

### 8-3-2 快速排序

#### 5. 算法描述

##### 基本思想:

任取待排序对象序列中的某个对象 (例如取第一个对象) 作为基准 (枢轴), 按照该对象的关键字大小, 将整个对象序列划分为左右两个子序列:

左侧子序列中所有对象的关键字都小于或等于基准对象的关键字

右侧子序列中所有对象的关键字都大于基准对象的关键字。

每一趟排序, 确定基准 (枢轴) 记录的位置。

左侧子序列和右侧子序列分别做快速排序。 <递归思想>

## 8.3 交换排序

### 8-3-2 快速排序

#### 5. 算法描述

##### 基本思想:

任取待排序对象序列中的某个对象 (例如取第一个对象) 作为基准 (枢轴), 按照该对象的关键字大小, 将整个对象序列划分为左右两个子序列:

```
void Sort :: QuickSort(int first, int last)
{
    if (first == last) return; /*区间长度为1, 递归结束*/
    else {
        int pivot = Partition(first, last);
        QuickSort(first, pivot-1);
        QuickSort(pivot+1, last);
    }
}
```

1. 每一趟排序, 确定基准或枢轴记录的位置。
2. 左侧子序列和右侧子序列分别做快速排序。  
<递归思想>

#### 5. 算法描述

```
int Sort :: Partition(int first, int last)
```

```
{
```

```
    int i = first, j = last, temp;
```

```
    while (i < j)
```

```
    {
```

```
        while (i < j && data[i] <= data[j]) j--;
```

```
        if (i < j) {
```

```
            temp = data[i]; data[i] = data[j]; data[j] = temp; i++;
```

```
        }
```

```
        while (i < j && data[i] <= data[j]) i++;
```

```
        if (i < j) {
```

```
            temp = data[i]; data[i] = data[j]; data[j] = temp; j--;
```

```
        }
```

```
    }
```

```
    return i;
```

```
}
```



比较语句？ 执行次数？

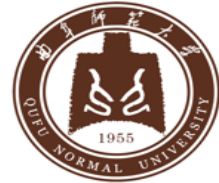


移动语句？ 执行次数？



取决于待排序序列的初始状态





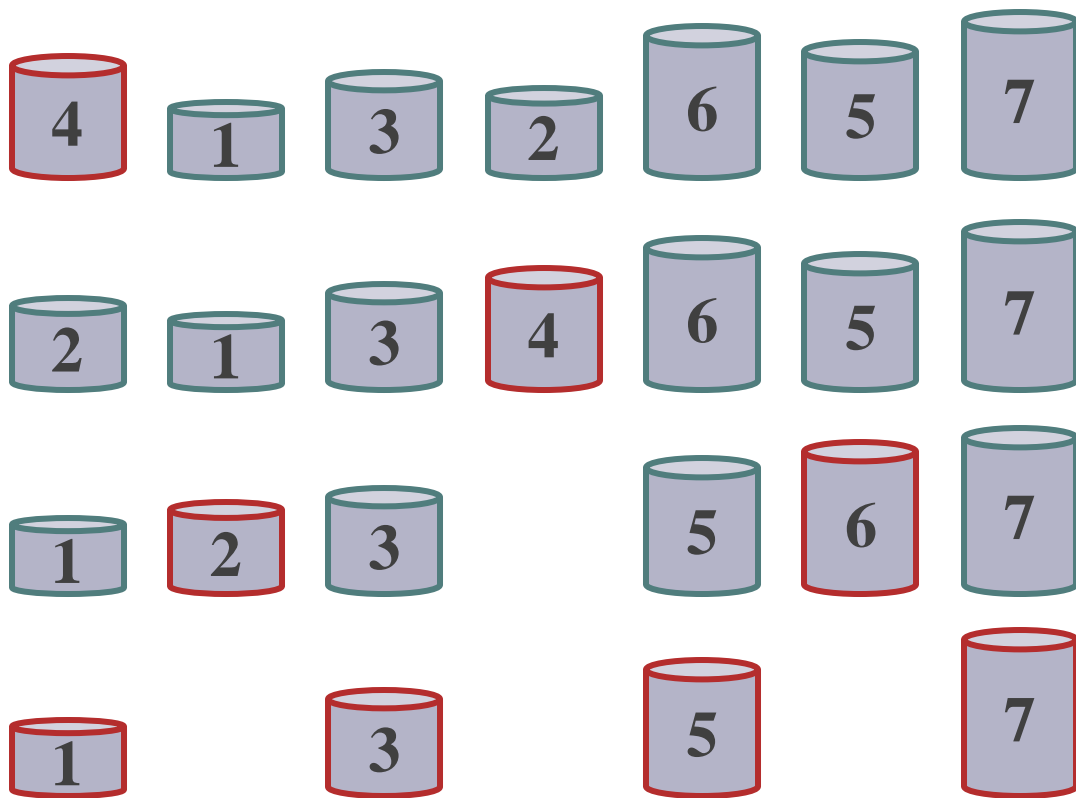
## 6. 算法时间性能分析

📌 最好情况：每次划分的轴值均是中值

$$O(n \log_2 n)$$

📌 排序趟数： $\log_2 n$

📌 一趟排序： $O(n)$



## 8.3 交换排序

### 8-3-2 快速排序

#### 6. 算法时间性能分析

📜 最好情况：每次划分的轴值均是中值

$$O(n\log_2 n)$$

📎 排序趟数： $\log_2 n$

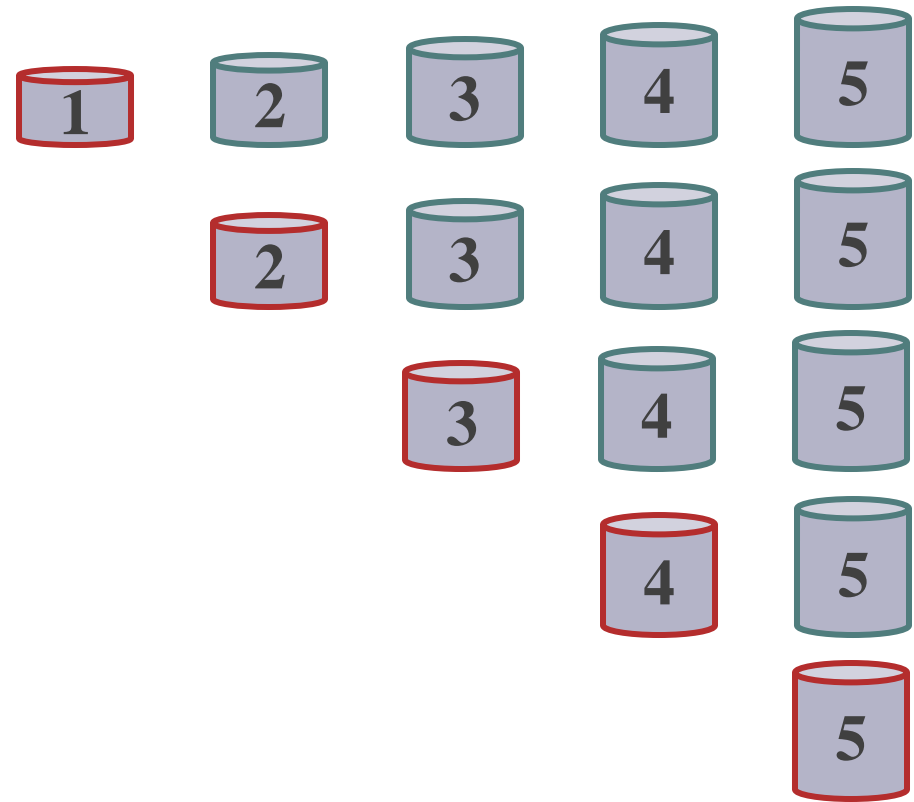
📎 一趟排序： $O(n)$

📜 最坏情况：正序、逆序  $O(n^2)$

📎 排序趟数： $n-1$

📎 一趟排序： $O(n)$

📜 平均情况： $O(n\log_2 n)$



## 8.3 交换排序

### 8-3-2 快速排序

#### 6. 算法空间性能分析

 空间性能:  $O(\log_2 n) \sim O(n)$

 一次划分:  $O(1)$

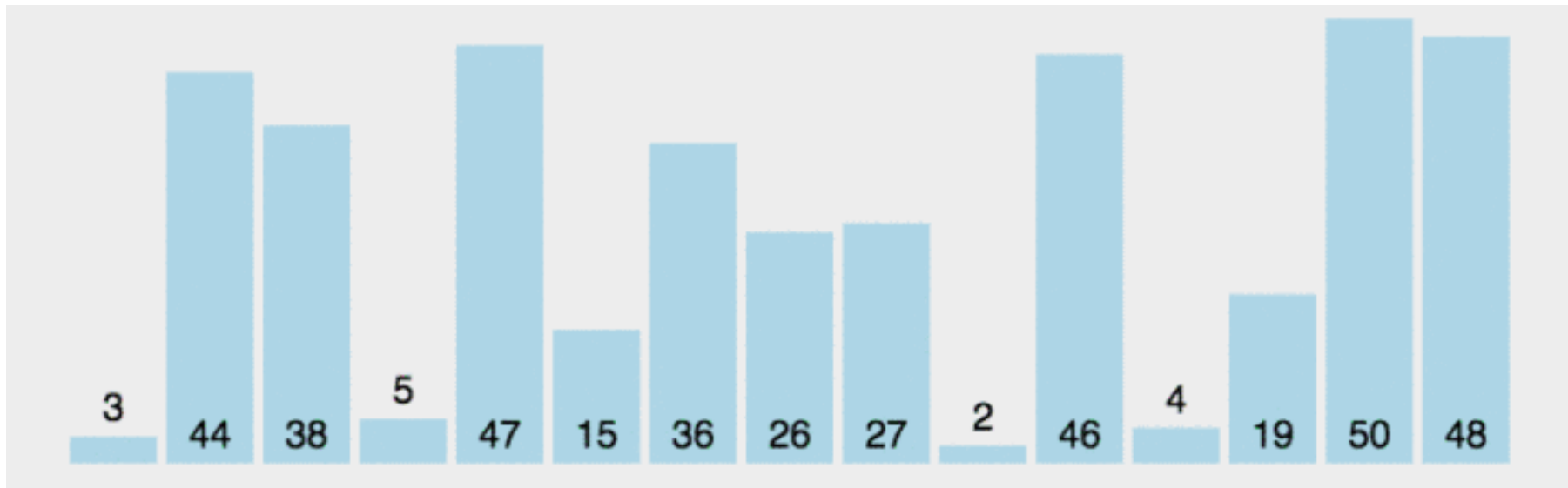
 递归深度:  $O(\log_2 n) \sim O(n)$

 稳定性: 不稳定

```
void Sort :: QuickSort (int first, int last)
{
    int pivot = Partition (first, last);
    QuickSort (first, pivot-1);
    QuickSort (pivot+1, last );
}
```

## 小结

1. 理解**交换排序**的基本思想
2. 掌握**起泡排序**的思想和实现方法
3. 掌握**快速排序**的基本思想
4. 掌握**一次划分**方法和**快速排序**的递归实现

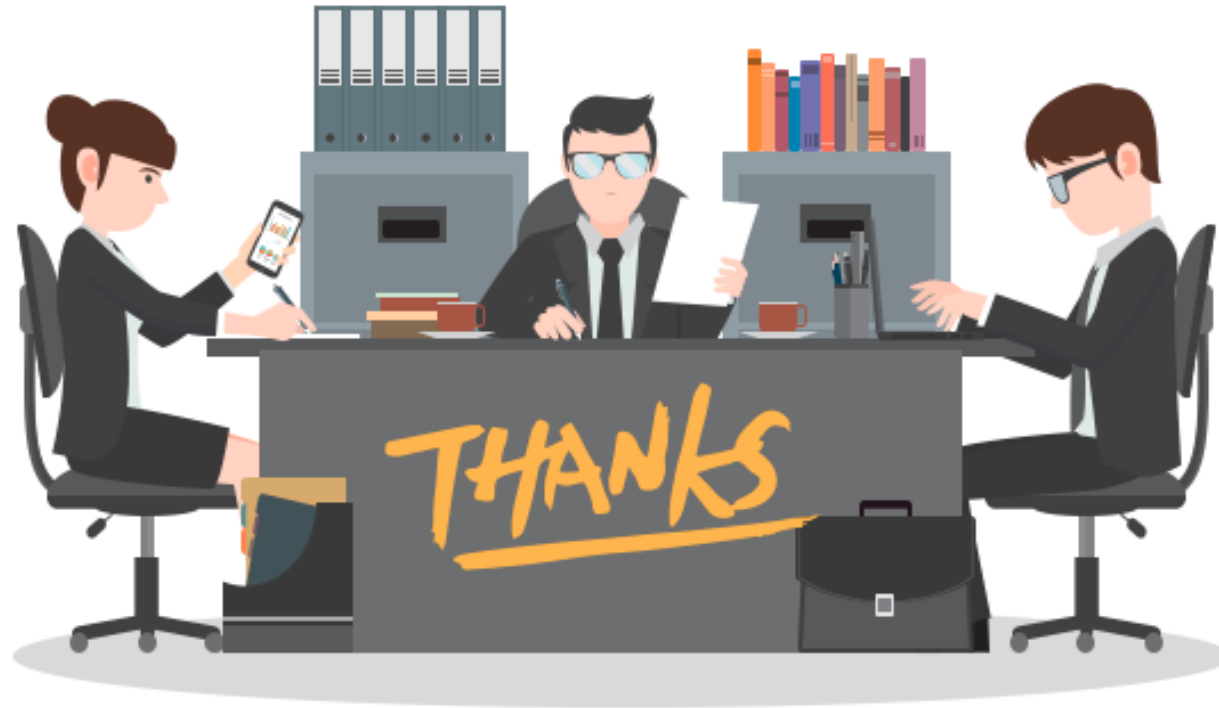


快速排序



# 作业

1. 设计C++类，并完成**起泡排序和快速排序**成员函数的设计与测试。



*Thank You !*

*Q & A*