

# 排序 (Sorting)

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

码拉松



实力IT教育 www.520it.com

# 初识排序

## ■ 什么叫排序?

□ 排序前: 3,1,6,9,2,5,8,4,7

□ 排序后: 1,2,3,4,5,6,7,8,9 (升序) 或者 9,8,7,6,5,4,3,2,1 (降序)

## ■ 排序的应用无处不在

排名	车型	所属厂商	2月销量
1	哈弗H6	长城汽车	25728
2	大众途观	上汽大众	15428
3	吉利博越	吉利汽车	15013
4	宝骏510	上汽通用五菱	12268
5	现代ix35	北京现代	12178
6	长安CS75	长安汽车	11297
7	哈弗F7	长城汽车	10665
8	长安CS55	长安汽车	10476
9	长安CS35	长安汽车	10353
10	奔驰GLC	北京奔驰	9450



# 10大排序算法

名称	时间复杂度			额外空间复杂度	In-place	稳定性
	最好	最坏	平均			
冒泡排序 (Bubble Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
选择排序 (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✗
插入排序 (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
归并排序 (Merge Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✗	✓
快速排序 (Quick Sort)	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$	✓	✗
希尔排序 (Shell Sort)	$O(n)$	$O(n^{4/3}) \sim O(n^2)$	取决于步长序列	$O(1)$	✓	✗
堆排序 (Heap Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	✓	✗
计数排序 (Counting Sort)	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	✗	✓
基数排序 (Radix Sort)	$O(d * (n + k))$	$O(d * (n + k))$	$O(d * (n + k))$	$O(n + k)$	✗	✓
桶排序 (Bucket Sort)	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + m)$	✗	✓

■ 以上表格是基于数组进行排序的一般性结论

■ 冒泡、选择、插入、归并、快速、希尔、堆排序，属于**比较排序** (Comparison Sorting)

# 冒泡排序 (Bubble Sort)

■ 冒泡排序也叫做起泡排序

■ 执行流程 (本课程统一以升序为例子)

① 从头开始比较每一对相邻元素, 如果第1个比第2个大, 就交换它们的位置

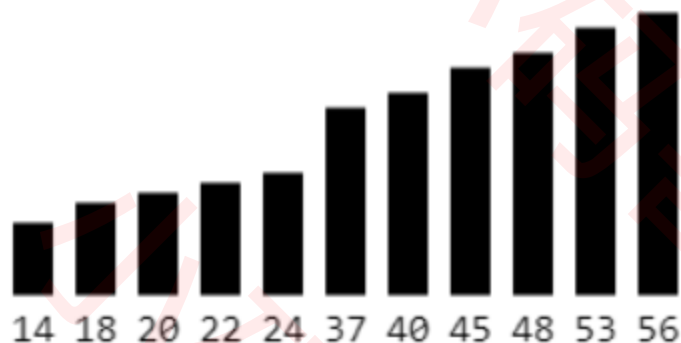
✓ 执行完一轮后, 最末尾那个元素就是最大的元素

② 忽略 ① 中曾经找到的最大元素, 重复执行步骤 ①, 直到全部元素有序

```
for (int end = array.length - 1; end > 0; end--) {  
    for (int begin = 1; begin <= end; begin++) {  
        if (cmp(begin, begin - 1) < 0) {  
            swap(begin, begin - 1);  
        }  
    }  
}
```

# 冒泡排序 – 优化①

- 如果序列已经完全有序，可以提前终止冒泡排序

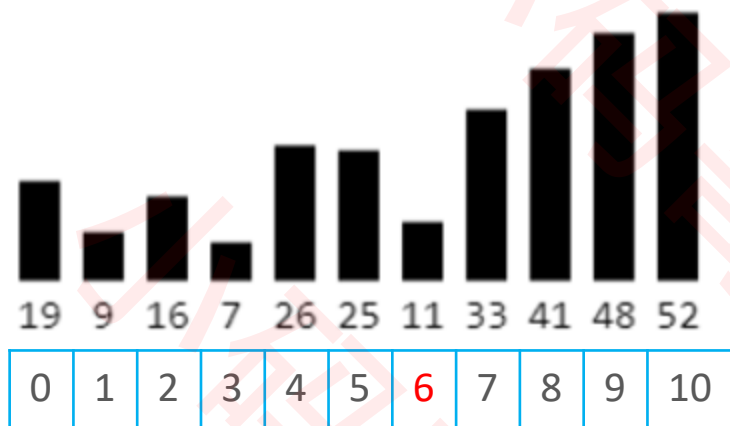


```
for (int end = array.length - 1; end > 0; end--) {  
    boolean sorted = true;  
    for (int begin = 1; begin <= end; begin++) {  
        if (cmp(begin, begin - 1) < 0) {  
            swap(begin, begin - 1);  
            sorted = false;  
        }  
    }  
    if (sorted) break;  
}
```



## 冒泡排序 – 优化②

- 如果序列尾部已经局部有序，可以记录最后1次交换的位置，减少比较次数



- 最后1次交换的位置是 6

```
for (int end = array.length - 1; end > 0; end--) {  
    int sortedIndex = 1;  
    for (int begin = 1; begin <= end; begin++) {  
        if (cmp(begin, begin - 1) < 0) {  
            swap(begin, begin - 1);  
            sortedIndex = begin;  
        }  
    }  
    end = sortedIndex;  
}
```

- 最坏、平均时间复杂度:  $O(n^2)$
- 最好时间复杂度:  $O(n)$
- 空间复杂度:  $O(1)$

# 排序算法的稳定性 (Stability)

■ 如果相等的2个元素，在排序前后的相对位置保持不变，那么这是**稳定**的排序算法

□ 排序前: 5, 1, 3<sub>a</sub>, 4, 7, 3<sub>b</sub>

□ 稳定的排序: 1, 3<sub>a</sub>, 3<sub>b</sub>, 4, 5, 7

□ 不稳定的排序: 1, 3<sub>b</sub>, 3<sub>a</sub>, 4, 5, 7

■ 对自定义对象进行排序时，稳定性会影响最终的排序效果

■ 冒泡排序属于稳定的排序算法

□ 稍有不慎，稳定的排序算法也能被写成不稳定的排序算法，比如下面的冒泡排序代码是不稳定的

```
for (int end = array.length - 1; end > 0; end--) {  
    for (int begin = 1; begin <= end; begin++) {  
        if (cmp(begin, begin - 1) <= 0) {  
            swap(begin, begin - 1);  
        }  
    }  
}
```

# 原地算法 (In-place Algorithm)

- 何为原地算法?
  - 不依赖额外的资源或者依赖少数的额外资源，仅依靠输出来覆盖输入
  - 空间复杂度为  $O(1)$  的都可以认为是原地算法
- 非原地算法，称为 Not-in-place 或者 Out-of-place
- 冒泡排序属于 In-place



# 选择排序 (Selection Sort)

## ■ 执行流程

① 从序列中找出最大的那个元素，然后与最末尾的元素交换位置

✓ 执行完一轮后，最末尾的那个元素就是最大的元素

② 忽略 ① 中曾经找到的最大元素，重复执行步骤 ①

```
for (int end = array.length - 1; end > 0; end--) {  
    int max = 0;  
    for (int begin = 1; begin <= end; begin++) {  
        if (cmp(max, begin) < 0) {  
            max = begin;  
        }  
    }  
    swap(max, end);  
}
```

## ■ 思考

□ 选择排序是否还有优化的空间?

✓ 使用堆来选择最大值

■ 选择排序的交换次数要远远少于冒泡排序，平均性能优于冒泡排序

■ 最好、最坏、平均时间复杂度： $O(n^2)$ ，空间复杂度： $O(1)$ ，属于不稳定排序

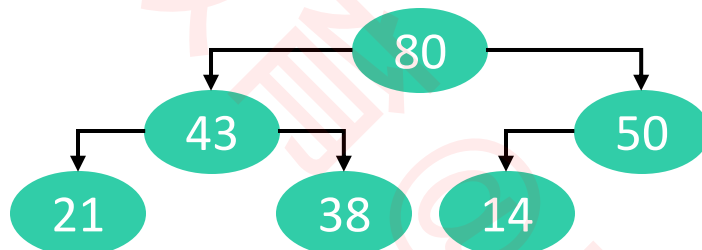
# 堆排序 (Heap Sort)

■ 堆排序可以认为是对选择排序的一种优化

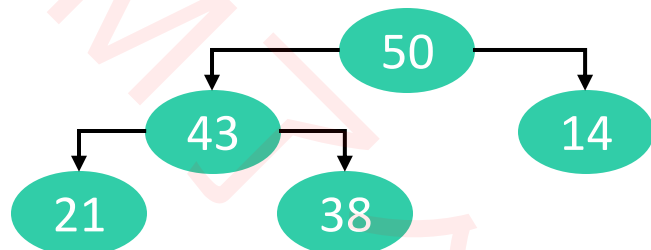
■ 执行流程

- ① 对序列进行原地建堆 (heapify)
- ② 重复执行以下操作，直到堆的元素数量为 1
  - ✓ 交换堆顶元素与尾元素
  - ✓ 堆的元素数量减 1
  - ✓ 对 0 位置进行 1 次 siftDown 操作

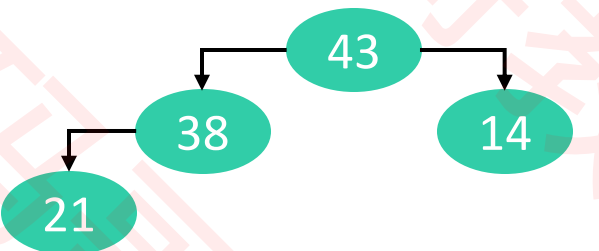
0	1	2	3	4	5
50	21	80	43	38	14



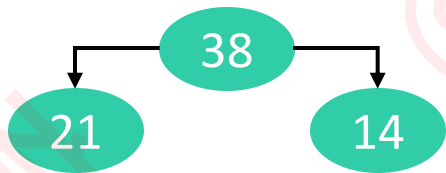
0	1	2	3	4	5
80	43	50	21	38	14



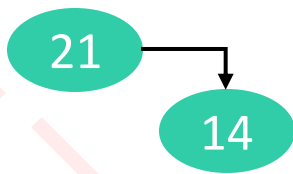
0	1	2	3	4	5
50	43	14	21	38	80



0	1	2	3	4	5
43	38	14	21	50	80



0	1	2	3	4	5
38	21	14	43	50	80



0	1	2	3	4	5
21	14	38	43	50	80



0	1	2	3	4	5
14	21	38	43	50	80

# 堆排序 - 实现

```
// heapify
heapSize = array.length;
for (int i = (heapSize >> 1) - 1; i >= 0; i--) {
    siftDown(i);
}

while (heapSize > 1) {
    swap(0, --heapSize);
    siftDown(0);
}
```

```
private void siftDown(int index) {
    T element = array[index];
    int half = heapSize >> 1;
    while (index < half) {
        int childIndex = (index << 1) + 1;
        T child = array[childIndex];
        int rightIndex = childIndex + 1;
        if (rightIndex < heapSize
            && cmp(array[rightIndex], child) > 0) {
            child = array[childIndex = rightIndex];
        }
        if (cmp(element, child) >= 0) break;
        array[index] = child;
        index = childIndex;
    }
    array[index] = element;
}
```

- 最好、最坏、平均时间复杂度： $O(n \log n)$ ，空间复杂度： $O(1)$ ，属于不稳定排序

# 插入排序 (Insertion Sort)

■ 插入排序非常类似于扑克牌的排序



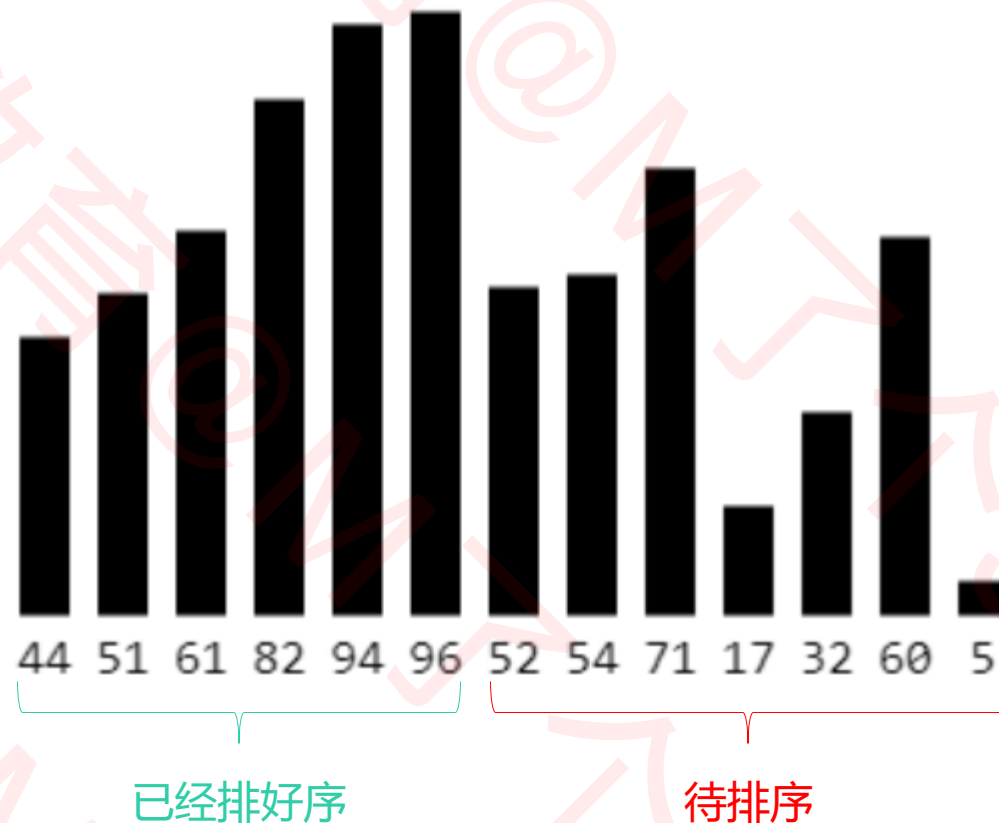
■ 执行流程

① 在执行过程中，插入排序会将序列分为2部分

✓ 头部是已经排好序的，尾部是待排序的

② 从头开始扫描每一个元素

✓ 每当扫描到一个元素，就将它插入到头部合适的位置，使得头部数据依然保持有序



# 插入排序 - 实现

```
for (int begin = 1; begin < array.length; begin++) {  
    int cur = begin;  
    while (cur > 0 && cmp(cur, cur - 1) < 0) {  
        swap(cur, cur - 1);  
        cur--;  
    }  
}
```

# 插入排序 – 逆序对 (Inversion)

## ■ 什么是逆序对?

□ 数组  $\langle 2, 3, 8, 6, 1 \rangle$  的逆序对为:  $\langle 2, 1 \rangle$   $\langle 3, 1 \rangle$   $\langle 8, 1 \rangle$   $\langle 8, 6 \rangle$   $\langle 6, 1 \rangle$ , 共5个逆序对

## ■ 插入排序的时间复杂度与逆序对的数量成正比关系

□ 逆序对的数量越多, 插入排序的时间复杂度越高

■ 最坏、平均时间复杂度:  $O(n^2)$

■ 最好时间复杂度:  $O(n)$

■ 空间复杂度:  $O(1)$

■ 属于稳定排序

■ 当逆序对的数量极少时, 插入排序的效率特别高

□ 甚至速度比  $O(n \log n)$  级别的快速排序还要快

■ 数据量不是特别大的时候, 插入排序的效率也是非常好的

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

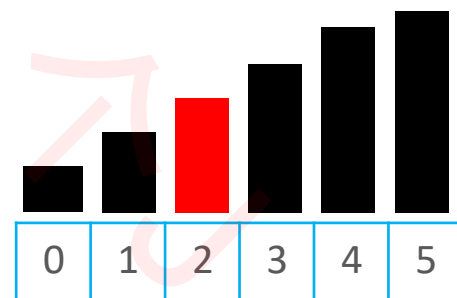
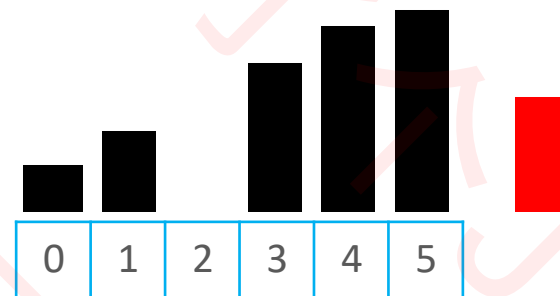
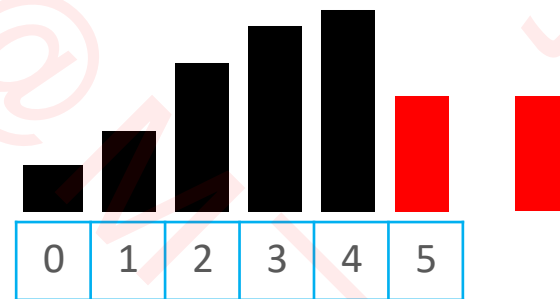


# 插入排序 - 优化

■ 思路是将【交换】转为【挪动】

- ① 先将待插入的元素备份
- ② 头部有序数据中比待插入元素大的，都朝尾部方向挪动1个位置
- ③ 将待插入元素放到最终的合适位置

```
for (int begin = 1; begin < array.length; begin++) {  
    int cur = begin;  
    T v = array[cur];  
    while (cur > 0 && cmp(v, array[cur - 1]) < 0) {  
        array[cur] = array[cur - 1];  
        cur--;  
    }  
    array[cur] = v;  
}
```



# 二分搜索 (Binary Search)

- 如何确定一个元素在数组中的位置？（假设数组里面全都是整数）
- 如果是无序数组，从第 0 个位置开始遍历搜索，平均时间复杂度： $O(n)$

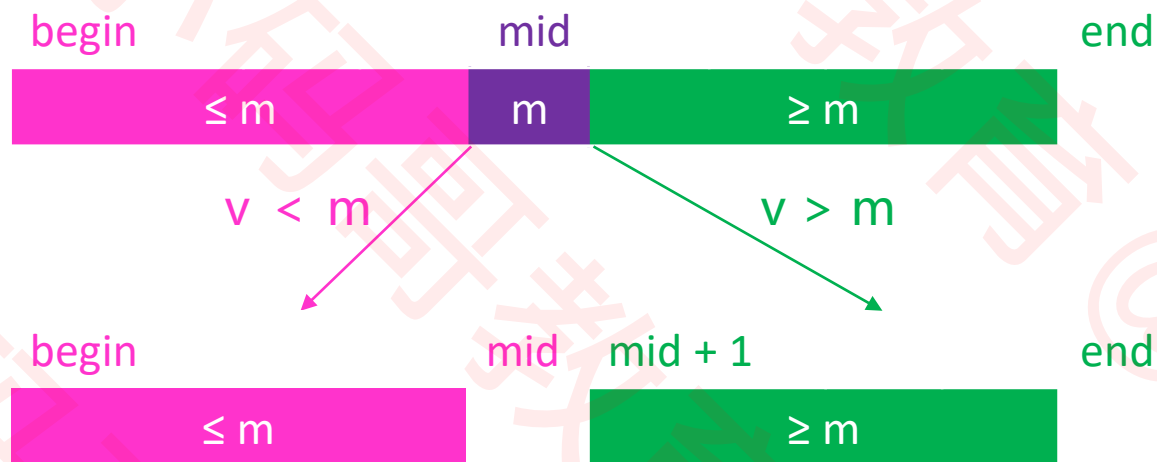
0	1	2	3	4	5	6	7	8	9
31	66	17	15	28	20	59	88	45	56

- 如果是有序数组，可以使用二分搜索，最坏时间复杂度： $O(\log n)$

0	1	2	3	4	5	6	7	8	9
15	17	20	28	31	45	56	59	66	88

# 二分搜索 - 思路

- 假设在  $[begin, end)$  范围内搜索某个元素  $v$ ,  $mid == (begin + end) / 2$
- 如果  $v < m$ , 去  $[begin, mid)$  范围内二分搜索
- 如果  $v > m$ , 去  $[mid + 1, end)$  范围内二分搜索
- 如果  $v == m$ , 直接返回  $mid$



# 二分搜索 - 实例

■ 搜索 10

0	1	2	3	4	5	6	7
2	4	6	8	10	12	14	

4	5	6	7
10	12	14	

4	5
10	

↓  
命中

■ 搜索 3

0	1	2	3	4	5	6	7
2	4	6	8	10	12	14	

0	1	2	3
2	4	6	

0	1
2	

1

↓  
失败

begin == end == 1

# 二分搜索 - 实现

```
public static int search(int[] array, int v) {  
    if (array == null || array.length == 0) return -1;  
    int begin = 0;  
    int end = array.length;  
    while (begin < end) {  
        int mid = (begin + end) >> 1;  
        if (v < array[mid]) {  
            end = mid;  
        } else if (v > array[mid]) {  
            begin = mid + 1;  
        } else {  
            return mid;  
        }  
    }  
    return -1;  
}
```

## ■ 思考

□ 如果存在多个重复的值，返回的是哪一个？

✓ 不确定

# 插入排序 – 二分搜索优化

- 在元素  $v$  的插入过程中，可以先二分搜索出合适的插入位置，然后再将元素  $v$  插入

0	1	2	3	4	5	6	7
2	4	8	8	8	12	14	

- 要求二分搜索返回的插入位置：第1个大于  $v$  的元素位置

- 如果  $v$  是 5，返回 2

- 如果  $v$  是 1，返回 0

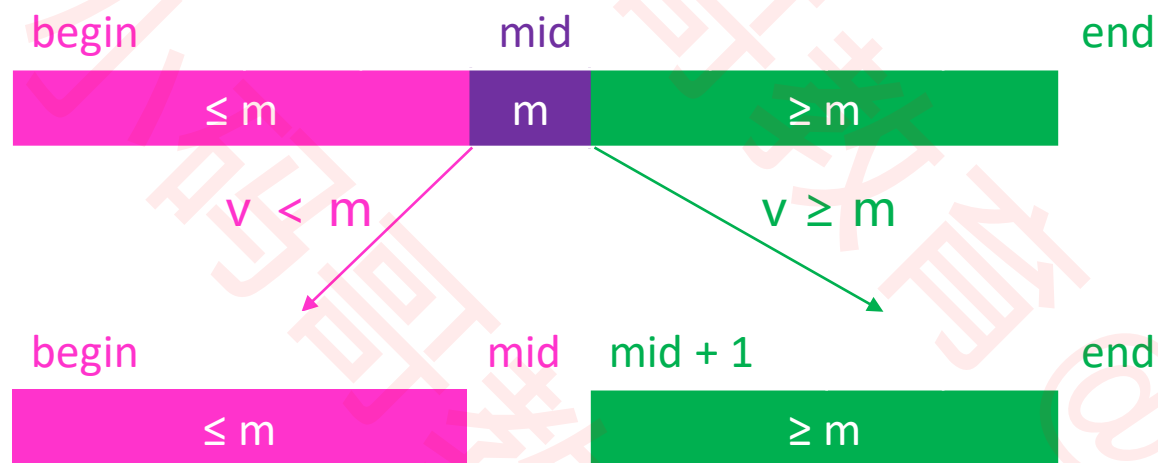
- 如果  $v$  是 15，返回 7

- 如果  $v$  是 8，返回 5



# 插入排序 - 二分搜索优化 - 思路

- 假设在  $[begin, end)$  范围内搜索某个元素  $v$ ,  $mid == (begin + end) / 2$
- 如果  $v < m$ , 去  $[begin, mid)$  范围内二分搜索
- 如果  $v \geq m$ , 去  $[mid + 1, end)$  范围内二分搜索



# 插入排序 - 二分搜索优化 - 实例

■ 搜索 5

0	1	2	3	4	5	6	7
2	4	8	8	8	12	14	

0	1	2	3
2	4	8	

2	3
8	

2

begin == end == 2

■ 搜索 1

0	1	2	3	4	5	6	7
2	4	8	8	8	12	14	

0	1	2	3
2	4	8	

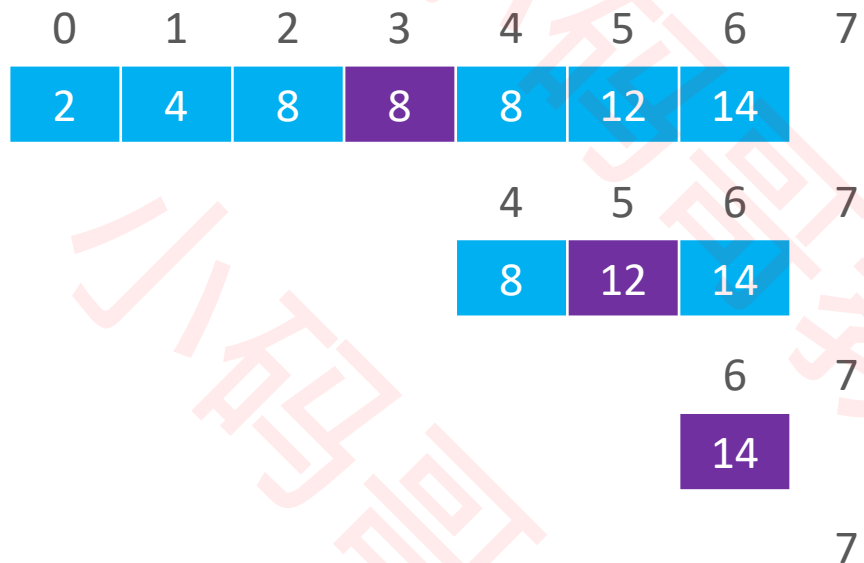
0	1
2	

0

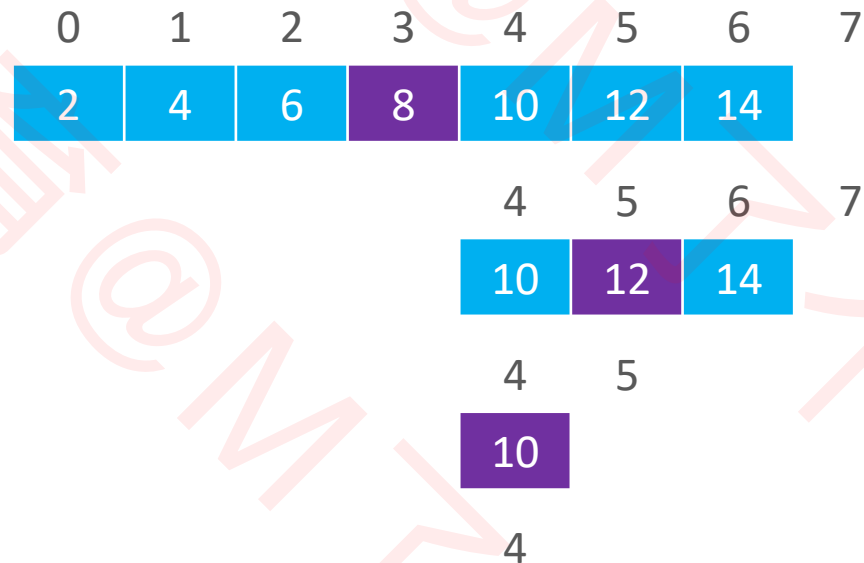
begin == end == 0

# 插入排序 - 二分搜索优化 - 实例

■ 搜索 15



■ 搜索 8



# 插入排序 - 二分搜索优化 - 实例

■ 搜索 8

0	1	2	3	4	5	6	7
2	4	8	8	8	12	14	

4	5	6	7
8	12	14	

4	5
8	

5

begin == end == 5

# 插入排序 - 二分搜索优化 - 实现

```
for (int i = 1; i < array.length; i++) {  
    insert(i, search(i));  
}
```

```
private void insert(int source, int dest) {  
    T v = array[source];  
    for (int i = source; i > dest; i--) {  
        array[i] = array[i - 1];  
    }  
    array[dest] = v;  
}
```

```
private int search(int index) {  
    int begin = 0;  
    int end = index;  
    while (begin < end) {  
        int mid = (begin + end) >> 1;  
        if (cmp(index, mid) < 0) {  
            end = mid;  
        } else {  
            begin = mid + 1;  
        }  
    }  
    return begin;  
}
```

■ 需要注意的是，使用了二分搜索后，只是减少了比较次数，但插入排序的平均时间复杂度依然是  $O(n^2)$

# 归并排序 (Merge Sort)

■ 1945年由约翰·冯·诺伊曼 (John von Neumann) 首次提出

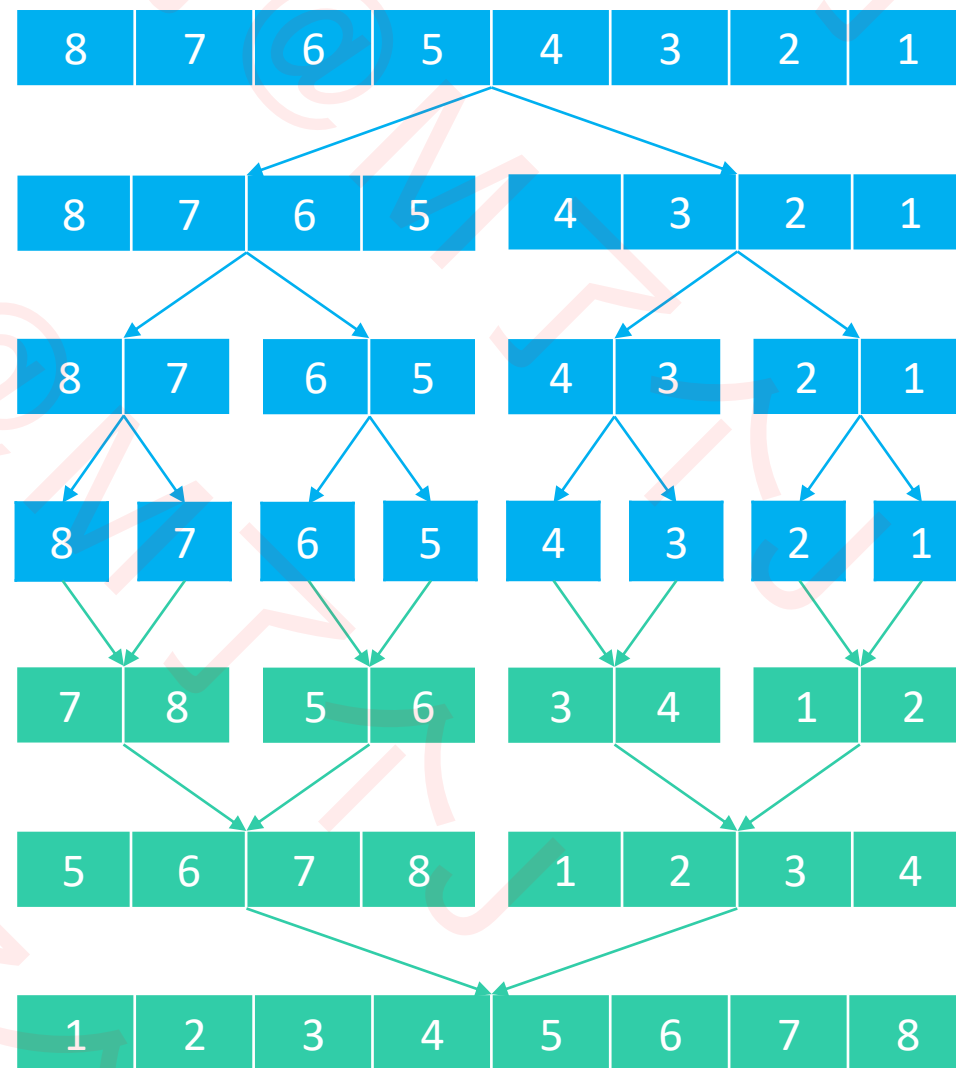


## ■ 执行流程

- ① 不断地将当前序列平均分割成2个子序列  
✓ 直到不能再分割 (序列中只剩1个元素)
- ② 不断地将2个子序列合并成一个有序序列  
✓ 直到最终只剩下1个有序序列

divide

merge





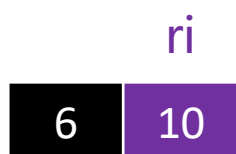
# 归并排序 – divide实现

```
// 准备一段临时的数组空间，在merge操作中使用
leftArray = (T[]) new Object[array.length >> 1];
sort(0, array.length);
```

```
/**
 * [begin, end)
 */
private void sort(int begin, int end) {
    // 至少要有2个元素
    if (end - begin < 2) return;

    int mid = (begin + end) >> 1;
    sort(begin, mid);
    sort(mid, end);
    merge(begin, mid, end);
}
```

# 归并排序 – merge

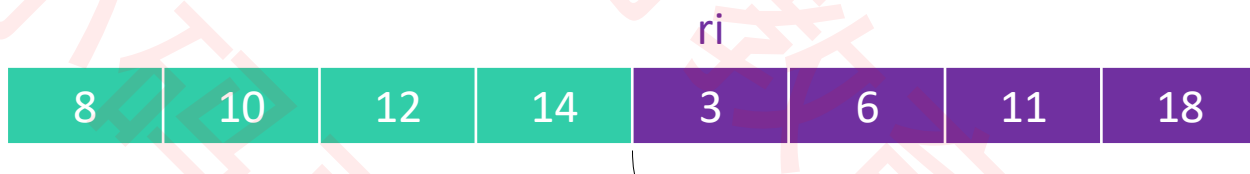


# 归并排序 – merge细节

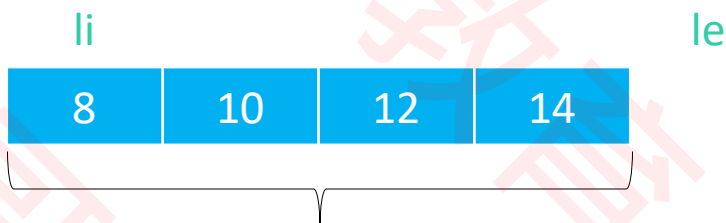
- 需要 merge 的 2 组序列存在于同一个数组中，并且是挨在一起的



- 为了更好地完成 merge 操作，最好将其中 1 组序列备份出来，比如  $[begin, mid)$



rightArray



leftArray

■  $li == 0, le == mid - begin$

■  $ri == mid, re == end$

# 归并排序 – merge

leftArray

li

3	8
---	---

li

3	8
---	---

li

3	8
---	---

li

3	8
---	---

array

ai

ri

3	8	6	10
---	---	---	----

ai

ri

3	8	6	10
---	---	---	----

ai

ri

3	6	6	10
---	---	---	----

ai

ri

3	6	8	10
---	---	---	----

# 归并排序 – merge – 左边先结束

leftArray

li

3	6
---	---

li

3	6
---	---

li

3	6
---	---

array

ai

ri

3	6	8	10
---	---	---	----

ai

ri

3	6	8	10
---	---	---	----

ai

ri

3	6	8	10
---	---	---	----

# 归并排序 – merge – 右边先结束

leftArray

li

8	10
---	----

li

8	10
---	----

li

8	10
---	----

li

8	10
---	----

li

8	10
---	----

array

ai

ri

8	10	3	6
---	----	---	---

ai

ri

3	10	3	6
---	----	---	---

ai

ri

3	6	3	6
---	---	---	---

ai

ri

3	6	8	6
---	---	---	---

ai

ri

3	6	8	10
---	---	---	----



# 归并排序 – merge实现

```
/* [begin, mid), [mid, end) */
private void merge(int begin, int mid, int end) {
    int li = 0, le = mid - begin; // 左边数组(基于leftArray)
    int ri = mid, re = end; // 右边数组(基于array)
    int ai = begin; // array的索引
    for (int i = li; i < le; i++) { // 拷贝左边数组到leftArray
        leftArray[i] = array[begin + i];
    }
    while (li < le) {
        if (ri < re && cmp(array[ri], leftArray[li]) < 0) {
            array[ai++] = array[ri++]; // 拷贝右边数组到array
        } else {
            array[ai++] = leftArray[li++]; // 拷贝左边数组到array
        }
    }
    // cmp位置改为 <= 会失去稳定性
}
```

# 归并排序 – 复杂度分析

## ■ 归并排序花费的时间

□  $T(n) = 2 * T(n/2) + O(n)$

□  $T(1) = O(1)$

□  $T(n)/n = T(n/2)/(n/2) + O(1)$

## ■ 令 $S(n) = T(n)/n$

□  $S(1) = O(1)$

□  $S(n) = S(n/2) + O(1) = S(n/4) + O(2) = S(n/8) + O(3) = S(n/2^k) + O(k) = S(1) + O(\log n) = O(\log n)$

□  $T(n) = n * S(n) = O(n \log n)$

■ 由于归并排序总是平均分割子序列，所以最好、最坏、平均时间复杂度都是  $O(n \log n)$ ，属于稳定排序

■ 从代码中不难看出：归并排序的空间复杂度是  $O(n/2 + \log n) = O(n)$

□  $n/2$  用于临时存放左侧数组， $\log n$  是因为递归调用

# 常见的递推式与复杂度

递推式	复杂度
$T(n) = T(n/2) + O(1)$	$O(\log n)$
$T(n) = T(n - 1) + O(1)$	$O(n)$
$T(n) = T(n/2) + O(n)$	$O(n)$
$T(n) = 2 * T(n/2) + O(1)$	$O(n)$
$T(n) = 2 * T(n/2) + O(n)$	$O(n \log n)$
$T(n) = T(n - 1) + O(n)$	$O(n^2)$
$T(n) = 2 * T(n - 1) + O(1)$	$O(2^n)$
$T(n) = 2 * T(n - 1) + O(n)$	$O(2^n)$

- 合并两个有序数组

- <https://leetcode-cn.com/problems/merge-sorted-array/>

- 合并两个有序链表

- <https://leetcode-cn.com/problems/merge-two-sorted-lists/comments/>

- 合并K个有序链表

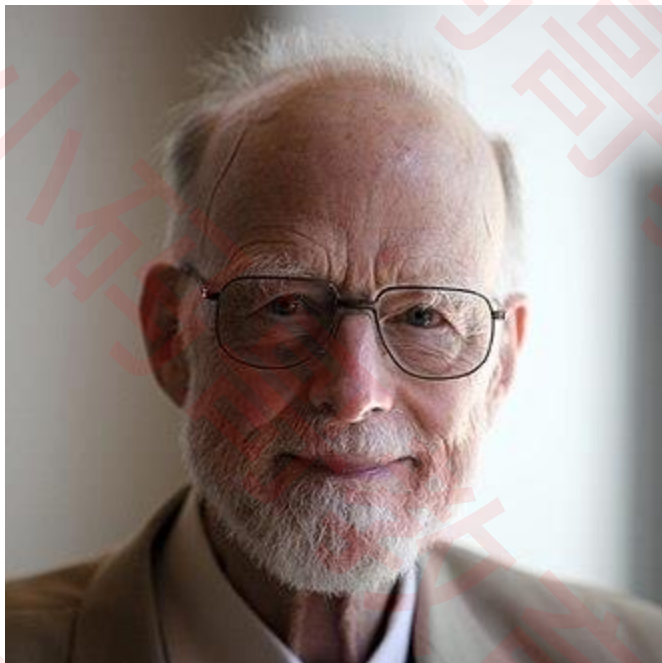
- <https://leetcode-cn.com/problems/merge-k-sorted-lists/>

- 解题教程

- <https://ke.qq.com/course/436549>

# 快速排序 (Quick Sort)

- 1960年由查尔斯·安东尼·理查德·霍尔 (Charles Antony Richard Hoare, 缩写为C. A. R. Hoare) 提出
- 昵称为东尼·霍尔 (Tony Hoare)



# 快速排序 – 执行流程

① 从序列中选择一个轴点元素 (pivot)

✓ 假设每次选择 0 位置的元素为轴点元素

② 利用 pivot 将序列分割成 2 个子序列

✓ 将小于 pivot 的元素放在pivot前面 (左侧)

✓ 将大于 pivot 的元素放在pivot后面 (右侧)

✓ 等于pivot的元素放哪边都可以

③ 对子序列进行 ① ② 操作

✓ 直到不能再分割 (子序列中只剩下1个元素)

6	11	8	2	9	4	1	5	7	10	3
---	----	---	---	---	---	---	---	---	----	---

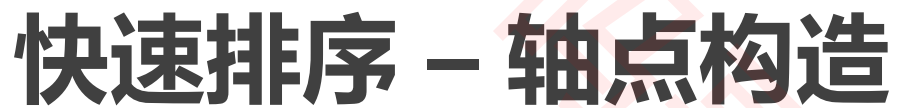
3	5	1	2	4	6	9	8	7	10	11
---	---	---	---	---	---	---	---	---	----	----

2	1	3	5	4	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

■ 快速排序的本质

□ 逐渐将每一个元素都转换成轴点元素





# 快速排序 – 时间复杂度

■ 在轴点左右元素数量比较均匀的情况下，同时也是最好的情况

□  $T(n) = 2 * T(n/2) + O(n) = O(n \log n)$

■ 如果轴点左右元素数量极度不均匀，最坏情况

□  $T(n) = T(n - 1) + O(n) = O(n^2)$

■ 为了降低最坏情况的出现概率，一般采取的做法是

□ 随机选择轴点元素

■ 最好、平均时间复杂度：  $O(n \log n)$

■ 最坏时间复杂度：  $O(n^2)$

■ 由于递归调用的缘故，空间复杂度：  $O(\log n)$

■ 属于不稳定排序

7	1	2	3	4	5	6
---	---	---	---	---	---	---

6	1	2	3	4	5	7
---	---	---	---	---	---	---

5	1	2	3	4	6	7
---	---	---	---	---	---	---

4	1	2	3	5	6	7
---	---	---	---	---	---	---

3	1	2	4	5	6	7
---	---	---	---	---	---	---

2	1	3	4	5	6	7
---	---	---	---	---	---	---

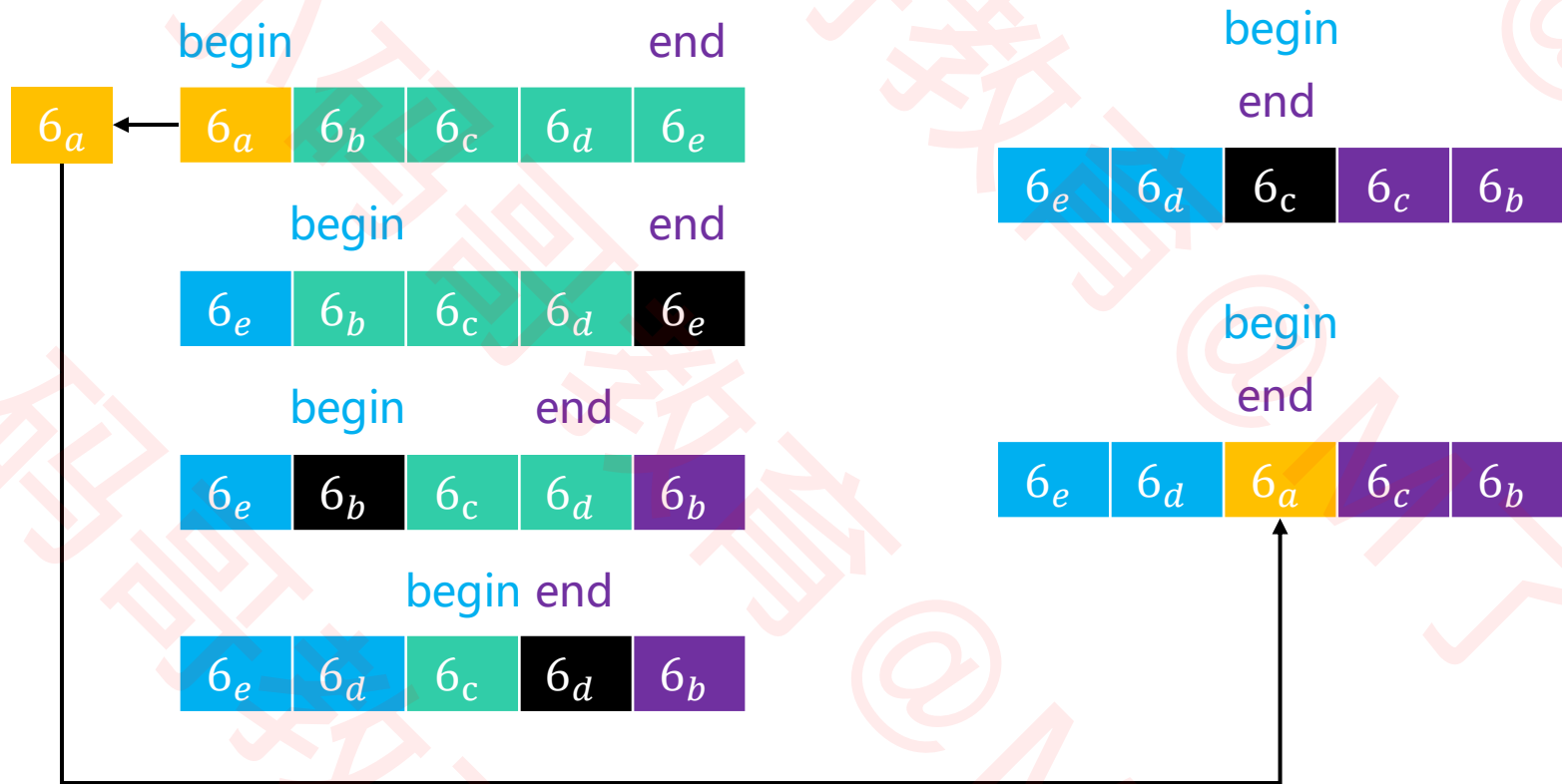
1	2	3	4	5	6	7
---	---	---	---	---	---	---

# 快速排序 – 实现

```
/* [begin, end) */  
private void sort(int begin, int end) {  
    // 至少要有2个元素  
    if (end - begin < 2) return;  
  
    int middle = pivotIndex(begin, end);  
    sort(begin, middle);  
    sort(middle + 1, end);  
}
```

```
private int pivotIndex(int begin, int end) {  
    // 随机交换begin位置的元素  
    swap(begin, begin + (int)(Math.random() * (end - begin)));  
    T pivot = array[begin];  
    end--; // end指向最后1个元素  
    while (begin < end) {  
        while (begin < end) {  
            if (cmp(pivot, array[end]) < 0) {  
                end--;  
            } else {  
                array[begin++] = array[end];  
                break;  
            }  
        }  
        while (begin < end) {  
            if (cmp(pivot, array[begin]) > 0) {  
                begin++;  
            } else {  
                array[end--] = array[begin];  
                break;  
            }  
        }  
    }  
    array[begin] = pivot;  
    return begin;  
}
```

# 快速排序 – 与轴点相等的元素



- 如果序列中的所有元素都与轴点元素相等，利用目前的算法实现，轴点元素可以将序列分割成 2 个均匀的子序列

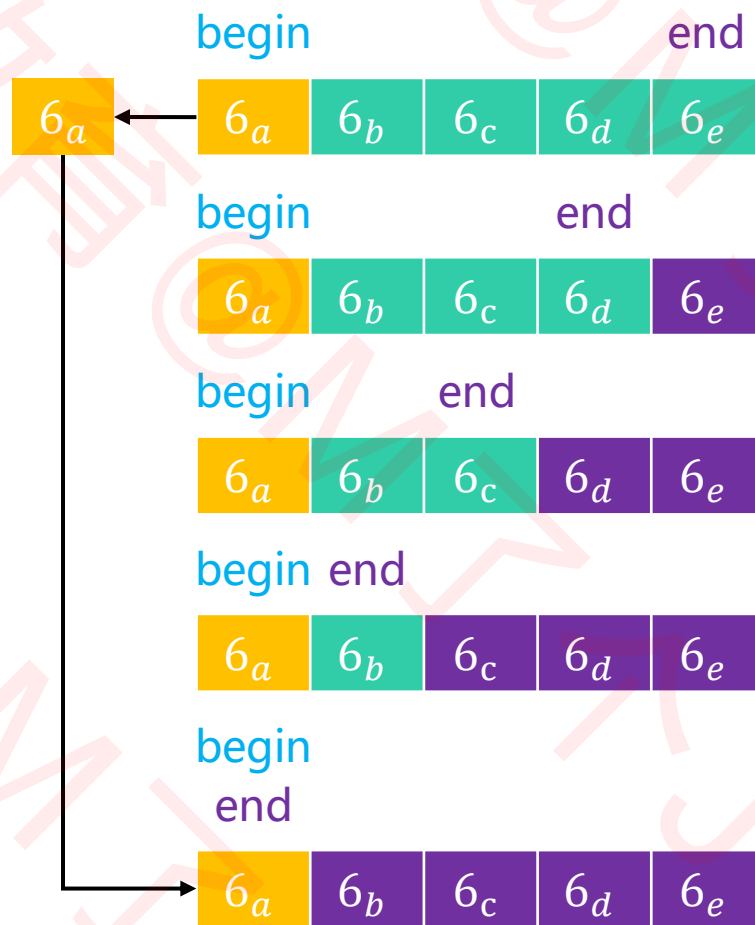
# 快速排序 – 与轴点相等的元素

■ 思考：cmp 位置的判断分别改为  $\leq$ 、 $\geq$  会起到什么效果？

```
while (begin < end) {
    if (cmp(pivot, array[end]) <= 0) {
        end--;
    } else {
        array[begin++] = array[end];
        break;
    }
}
while (begin < end) {
    if (cmp(pivot, array[begin]) >= 0) {
        begin++;
    } else {
        array[end--] = array[begin];
        break;
    }
}
```

■ 轴点元素分割出来的子序列极度不均匀

□ 导致出现最坏时间复杂度  $O(n^2)$



# 希尔排序 (Shell Sort)

- 1959年由唐纳德·希尔 (Donald Shell) 提出
- 希尔排序把序列看作是一个矩阵，分成  $m$  列，逐列进行排序
  - $m$  从某个整数逐渐减为1
  - 当  $m$  为1时，整个序列将完全有序
- 因此，希尔排序也被称为递减增量排序 (Diminishing Increment Sort)
- 矩阵的列数取决于步长序列 (step sequence)
  - ✓ 比如，如果步长序列为{1,5,19,41,109,...}，就代表依次分成109列、41列、19列、5列、1列进行排序
  - ✓ 不同的步长序列，执行效率也不同

# 希尔排序 – 实例

- 希尔本人给出的步长序列是  $n/2^k$ ，比如  $n$  为16时，步长序列是{1, 2, 4, 8}

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---

- 分成8列进行排序

16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1



8	7	6	5	4	3	2	1
16	15	14	13	12	11	10	9



8	7	6	5	4	3	2	1	16	15	14	13	12	11	10	9
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	---

# 希尔排序 - 实例

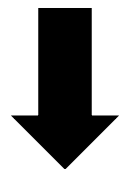
## ■ 分成4列进行排序

8	7	6	5	4	3	2	1	16	15	14	13	12	11	10	9
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	---

8	7	6	5	4	3	2	1
4	3	2	1	16	15	14	13
16	15	14	13	12	11	10	9
12	11	10	9	8	7	6	5



4	3	2	1	8	7	6	5
8	7	6	5	12	11	10	9
12	11	10	9	16	15	14	13
16	15	14	13	4	3	2	1



4	3	2	1	8	7	6	5	12	11	10	9	16	15	14	13
---	---	---	---	---	---	---	---	----	----	----	---	----	----	----	----

# 希尔排序 - 实例

■ 分成2列进行排序

4	3	2	1	8	7	6	5	12	11	10	9	16	15	14	13
---	---	---	---	---	---	---	---	----	----	----	---	----	----	----	----

4	3
2	1
8	7
6	5
12	11
10	9
16	15
14	13



2	1
4	3
6	5
8	7
10	9
12	11
14	13
16	15



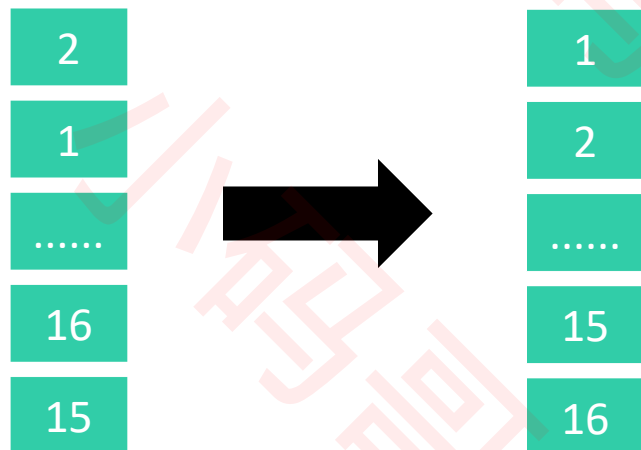
2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15
---	---	---	---	---	---	---	---	----	---	----	----	----	----	----	----



# 希尔排序 - 实例

## ■ 分成1列进行排序

2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15
---	---	---	---	---	---	---	---	----	---	----	----	----	----	----	----



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

■ 不难看出，从8列 变为 1列的过程中，逆序对的数量在逐渐减少

□ 因此希尔排序底层一般使用插入排序对每一列进行排序，也很多资料认为希尔排序是插入排序的改进版

# 希尔排序 - 实例

- 假设有11个元素，步长序列是{1, 2, 5}

11	10	9	8	7	6	5	4	3	2	1
----	----	---	---	---	---	---	---	---	---	---

11	10	9	8	7
----	----	---	---	---

6	5	4	3	2
---	---	---	---	---

1
---



1	5	4	3	2
---	---	---	---	---

6	10	9	8	7
---	----	---	---	---

11
----

- 假设元素在第 col 列、第 row 行，步长（总列数）是 step

- 那么这个元素在数组中的索引是  $col + row * step$

- 比如 9 在排序前是第 2 列、第 0 行，那么它排序前的索引是  $2 + 0 * 5 = 2$

- 比如 4 在排序前是第 2 列、第 1 行，那么它排序前的索引是  $2 + 1 * 5 = 7$

# 希尔排序 - 实现

```
List<Integer> stepSequence = sedgewickStepSequence();  
for (Integer step : stepSequence) {  
    sort(step);  
}
```

```
private void sort(int step) {  
    for (int col = 0; col < step; col++) {  
        for (int begin = col + step; begin < array.length; begin += step) {  
            int cur = begin;  
            while (cur > col && cmp(cur, cur - step) < 0) {  
                swap(cur, cur - step);  
                cur -= step;  
            }  
        }  
    }  
}
```

- 最好情况是步长序列只有1，且序列几乎有序，时间复杂度为  $O(n)$
- 空间复杂度为  $O(1)$ ，属于不稳定排序

# 希尔排序 - 步长序列

- 希尔本人给出的步长序列，最坏情况时间复杂度是  $O(n^2)$

```
private List<Integer> shellStepSequence() {  
    List<Integer> stepSequence = new ArrayList<>();  
    int step = array.length;  
    while ((step >>= 1) > 0) {  
        stepSequence.add(step);  
    }  
    return stepSequence;  
}
```

# 希尔排序 - 步长序列

■ 目前已知的最好的步长序列，最坏情况时间复杂度是  $O(n^{4/3})$ ，1986年由Robert Sedgewick提出

$$\begin{cases} 9 \left( 2^k - 2^{\frac{k}{2}} \right) + 1 & k \text{ even,} \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$$

1, 5, 19, 41, 109, ...

```
private List<Integer> sedgewickStepSequence() {  
    List<Integer> stepSequence = new LinkedList<>();  
    int k = 0, step = 0;  
    while (true) {  
        if (k % 2 == 0) {  
            int pow = (int) Math.pow(2, k >> 1);  
            step = 1 + 9 * (pow * pow - pow);  
        } else {  
            int pow1 = (int) Math.pow(2, (k - 1) >> 1);  
            int pow2 = (int) Math.pow(2, (k + 1) >> 1);  
            step = 1 + 8 * pow1 * pow2 - 6 * pow2;  
        }  
        if (step >= array.length) break;  
        stepSequence.add(0, step);  
        k++;  
    }  
    return stepSequence;  
}
```

# 计数排序 (Counting Sort)

- 之前学习的冒泡、选择、插入、归并、快速、希尔、堆排序，都是基于比较的排序
- 平均时间复杂度目前最低是  $O(n \log n)$
- 计数排序、桶排序、基数排序，都不是基于比较的排序
- 它们是典型的用空间换时间，在某些时候，平均时间复杂度可以比  $O(n \log n)$  更低
- 计数排序于1954年由Harold H. Seward提出，适合对一定范围内的整数进行排序
- 计数排序的核心思想
- 统计每个整数在序列中出现的次数，进而推导出每个整数在有序序列中的索引

# 计数排序 – 最简单的实现

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	存放所有整数出现的次数								
索引	0	1	2	3	4	5	6	7	8
次数				1	1	2	1	2	1

3	4	5	5	6	7	7	8
---	---	---	---	---	---	---	---

- 这个版本的实现存在以下问题
- 无法对负整数进行排序
- 极其浪费内存空间
- 是个不稳定的排序
- .....

```
int max = array[0]; // 最大值
for (int i = 1; i < array.length; i++) {
    if (array[i] > max) {
        max = array[i];
    }
}
// 统计元素出现的次数
int[] counts = new int[max + 1];
for (int i = 0; i < array.length; i++) {
    counts[array[i]]++;
}
// 按顺序赋值
int index = 0;
for (int i = 0; i < counts.length; i++) {
    while (counts[i]-- > 0) {
        array[index++] = i;
    }
}
```

# 计数排序 – 改进思路

array

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	从索引0开始依次存放3~8出现的次数					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	2	1	2	1

	每个次数累加上其前面的所有次数 得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	2	4	5	7	8

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

- 假设array中的最小值是 min
- array中的元素 k 对应的 counts 索引是  $k - \text{min}$
- array中的元素 k 在有序序列中的索引
  - $\text{counts}[k - \text{min}] - p$
  - p 代表着是倒数第几个 k
- 比如元素 8 在有序序列中的索引
  - $\text{counts}[8 - 3] - 1$ , 结果为 7
- 倒数第 1 个元素 7 在有序序列中的索引
  - $\text{counts}[7 - 3] - 1$ , 结果为 6
- 倒数第 2 个元素 7 在有序序列中的索引
  - $\text{counts}[7 - 3] - 2$ , 结果为 5



# 计数排序 – 改进思路

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素 得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	2	4	5	7	8

0	1	2	3	4	5	6	7

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素 得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	2	3	5	7	8

0	1	2	3	4	5	6	7
			5				

# 计数排序 – 改进思路

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	3	5	7	8

0	1	2	3	4	5	6	7
	4		5				

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	3	5	6	8

0	1	2	3	4	5	6	7
	4		5			7	

# 计数排序 – 改进思路

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	3	4	6	8

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

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	3	4	6	7

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

# 计数排序 – 改进思路

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	1	1	2	4	6	7

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

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	0	1	2	4	6	7

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

# 计数排序 – 改进思路

7	3	5	8	6	7	4	5
---	---	---	---	---	---	---	---

	每个元素累加上其前面的所有元素得到的就是元素在有序序列中的位置信息					
元素	3	4	5	6	7	8
索引	0	1	2	3	4	5
次数	0	1	2	4	5	7

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

# 计数排序 – 改进实现

```
int max = array[0]; // 最大值
int min = array[0]; // 最小值
for (int i = 1; i < array.length; i++) {
    if (array[i] > max) {
        max = array[i];
    }
    if (array[i] < min) {
        min = array[i];
    }
}
```

```
// 用于存放排好序的数据
int[] output = new int[array.length];
for (int i = array.length - 1; i >= 0; i--){
    output[--counts[array[i] - min]] = array[i];
}

for (int i = 0; i < array.length; i++){
    array[i] = output[i];
}
```

```
// 用于计数
int[] counts = new int[max - min + 1];
for (int i = 0; i < array.length; i++) {
    counts[array[i] - min]++;
}
for (int i = 1; i < counts.length; i++) {
    counts[i] += counts[i - 1];
}
```

- 最好、最坏、平均时间复杂度:  $O(n + k)$
- 空间复杂度:  $O(n + k)$
- $k$  是整数的取值范围
- 属于稳定排序

# 计数排序 – 对自定义对象进行排序

- 如果自定义对象可以提供用以排序的整数类型，依然可以使用计数排序

```
private static class Person {  
    int age;  
    String name;  
    Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
    @Override  
    public String toString() {  
        return "Person [age=" + age  
            + ", name=" + name + "];"  
    }  
}
```

```
Person[] persons = new Person[] {  
    new Person(20, "A"),  
    new Person(-13, "B"),  
    new Person(17, "C"),  
    new Person(12, "D"),  
    new Person(-13, "E"),  
    new Person(20, "F")  
};
```

```
int max = persons[0].age;  
int min = persons[0].age;  
for (int i = 1; i < persons.length; i++) {  
    if (persons[i].age > max) {  
        max = persons[i].age;  
    }  
    if (persons[i].age < min) {  
        min = persons[i].age;  
    }  
}
```

# 计数排序 – 对自定义对象进行排序

```
// 用于计数
int[] counts = new int[max - min + 1];
for (int i = 0; i < persons.length; i++) {
    counts[persons[i].age - min]++;
}
for (int i = 1; i < counts.length; i++) {
    counts[i] += counts[i - 1];
}

// 用于存放排好序的数据
Person[] output = new Person[persons.length];
for (int i = persons.length - 1; i >= 0; i--) {
    output[--counts[persons[i].age - min]] = persons[i];
}
```

## ■ 排序之后的结果

- ① Person [age=-13, name=B]
- ② Person [age=-13, name=E]
- ③ Person [age=12, name=D]
- ④ Person [age=17, name=C]
- ⑤ Person [age=20, name=A]
- ⑥ Person [age=20, name=F]



# 基数排序 (Radix Sort)

- 基数排序非常适合用于整数排序（尤其是非负整数），因此本课程只演示对非负整数进行基数排序
- 执行流程：依次对个位数、十位数、百位数、千位数、万位数...进行排序（从低位到高位）

126	69	593	23	6	89	54	8
-----	----	-----	----	---	----	----	---



593	23	54	126	6	8	69	89
-----	----	----	-----	---	---	----	----



6	8	23	126	54	69	89	593
---	---	----	-----	----	----	----	-----



6	8	23	54	69	89	126	593
---	---	----	----	----	----	-----	-----

- 个位数、十位数、百位数的取值范围都是固定的0~9，可以使用计数排序对它们进行排序
- 思考：如果先对高位排序，再对低位排序，是否可行？

# 基数排序 - 实现

```
int max = array[0]; // 最大值
for (int i = 1; i < array.length; i++) {
    if (array[i] > max) {
        max = array[i];
    }
}

int output[] = new int[array.length];
int counts[] = new int[10];
for (int divider = 1; divider <= max; divider *= 10) {
    // 对每一位进行计数排序
    countingSort(divider, output, counts);
}
```

# 基数排序 - 实现

```
private void countingSort(int divider, int[] output, int[] counts) {  
    for (int i = 0; i < counts.length; i++) {  
        counts[i] = 0;  
    }  
    for (int i = 0; i < array.length; i++) {  
        counts[array[i] / divider % 10]++;  
    }  
    for (int i = 1; i < counts.length; i++) {  
        counts[i] += counts[i - 1];  
    }  
    for (int i = array.length - 1; i >= 0; i--) {  
        output[--counts[array[i] / divider % 10]] = array[i];  
    }  
    for (int i = 0; i < array.length; i++) {  
        array[i] = output[i];  
    }  
}
```

- 最好、最坏、平均时间复杂度:  $O(d * (n + k))$ ,  $d$  是最大值的位数,  $k$  是进制。属于稳定排序
- 空间复杂度:  $O(n + k)$ ,  $k$  是进制

# 基数排序 - 另一种思路

126	69	593	23	6	89	54	8
-----	----	-----	----	---	----	----	---



0	1	2	3	4	5	6	7	8	9
			593	54		126		8	69
			23			6			89

593	23	54	126	6	8	69	89
-----	----	----	-----	---	---	----	----



0	1	2	3	4	5	6	7	8	9
6		23			54	69		89	593
8		126							

6	8	23	126	54	69	89	593
---	---	----	-----	----	----	----	-----



0	1	2	3	4	5	6	7	8	9
6	126				593				
8									
23									
54									
69									
89									

6	8	23	54	69	89	126	593
---	---	----	----	----	----	-----	-----

# 基数排序 - 另一种思路的实现

```
int max = array[0]; // 最大值
for (int i = 1; i < array.length; i++) {
    if (array[i] > max) {
        max = array[i];
    }
}
```

```
// 桶数组
int[][] buckets = new int[10][array.length];
// 每个桶的元素数量
int[] bucketSizes = new int[buckets.length];
for (int divider = 1; divider <= max; divider *= 10) {
    for (int i = 0; i < array.length; i++) {
        int no = array[i] / divider % 10;
        buckets[no][bucketSizes[no]++] = array[i];
    }
    int index = 0;
    for (int i = 0; i < buckets.length; i++) {
        for (int j = 0; j < bucketSizes[i]; j++) {
            array[index++] = buckets[i][j];
        }
        bucketSizes[i] = 0;
    }
}
```

- 空间复杂度是  $O(kn + k)$ ，时间复杂度是  $O(dn)$
- $d$  是最大值的位数， $k$  是进制

# 桶排序 (Bucket Sort)

## ■ 执行流程

- ① 创建一定数量的桶 (比如用数组、链表作为桶)
- ② 按照一定的规则 (不同类型的数据, 规则不同), 将序列中的元素均匀分配到对应的桶
- ③ 分别对每个桶进行单独排序
- ④ 将所有非空桶的元素合并成有序序列

## ■ 元素在桶中的索引

□ 元素值 \* 元素数量

0.34	0.47	0.29	0.84	0.45	0.38	0.35	0.76
------	------	------	------	------	------	------	------

0

1

2

3

4

5

6

7

0.34	0.29	0.35
0.47	0.45	0.38

0.84	0.76
------	------

0

1

2

3

4

5

6

7

0.29	0.34	0.35
0.38	0.45	0.47

0.76	0.84
------	------

0.29	0.34	0.35	0.38	0.45	0.47	0.76	0.84
------	------	------	------	------	------	------	------

# 桶排序 - 实现

```
double[] array = {0.34, 0.47, 0.29, 0.84, 0.45, 0.38, 0.35, 0.76};
```

```
// 桶数组
List<Double>[] buckets = new List[array.length];
for (int i = 0; i < array.length; i++) {
    int bucketIndex = (int) (array[i] * array.length);
    List<Double> bucket = buckets[bucketIndex];
    if (bucket == null) {
        bucket = new LinkedList<>();
        buckets[bucketIndex] = bucket;
    }
    bucket.add(array[i]);
}
```

```
// 对每个桶进行排序
int index = 0;
for (int i = 0; i < buckets.length; i++) {
    if (buckets[i] == null) continue;
    buckets[i].sort(null);
    for (Double d : buckets[i]) {
        array[index++] = d;
    }
}
```

■ 空间复杂度:  $O(n + m)$ ,  $m$  是桶的数量

■ 时间复杂度:  $O(n) + m * O\left(\frac{n}{m} * \log \frac{n}{m}\right) = O\left(n + n * \log \frac{n}{m}\right) = O(n + n * \log n - n * \log m)$

□ 因此为  $O(n + k)$ ,  $k$  为  $n * \log n - n * \log m$

□ 属于稳定排序

# 史上“最强”排序 – 休眠排序

```
private static class SortThread extends Thread {  
    private int value;  
    public SortThread(int value) {  
        this.value = value;  
    }  
    public void run() {  
        try {  
            Thread.sleep(value);  
            System.out.println(value);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    int[] array = {10, 100, 50, 30, 60};  
    for (int i = 0; i < array.length; i++) {  
        new SortThread(array[i]).start();  
    }  
}
```