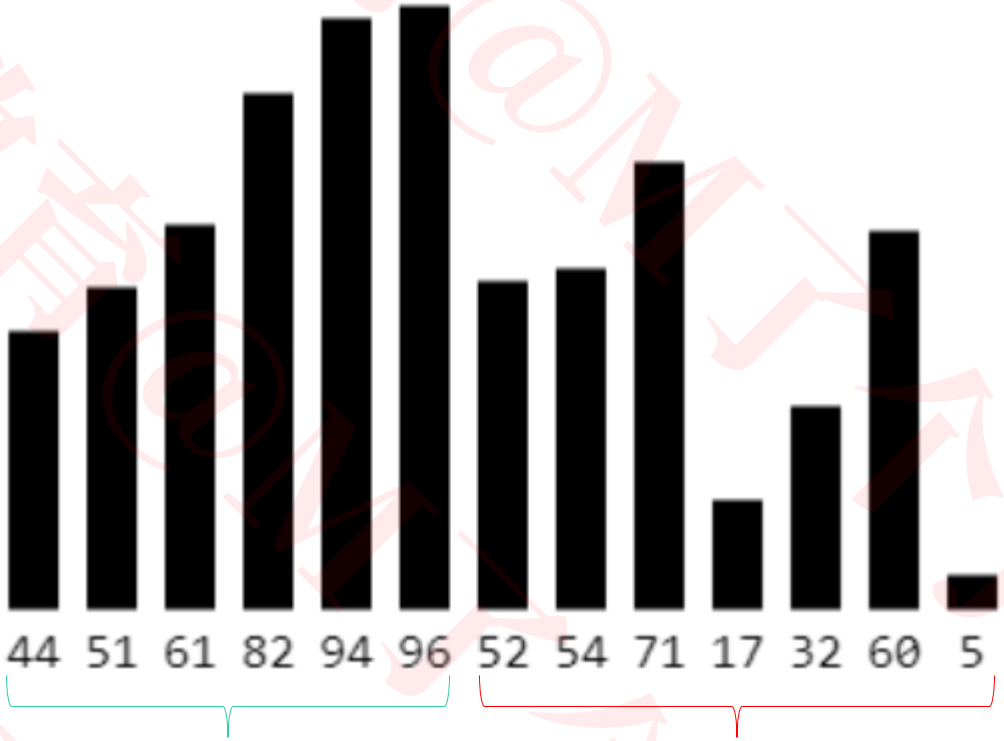
插入排序（Insertion Sort）

- 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

小码哥教育 SEEMYGO

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 <2,3,8,6,1> 的逆序对为：<2,1> <3,1> <8,1> <8,6> <6,1>，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 $O(nlogn)$ 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

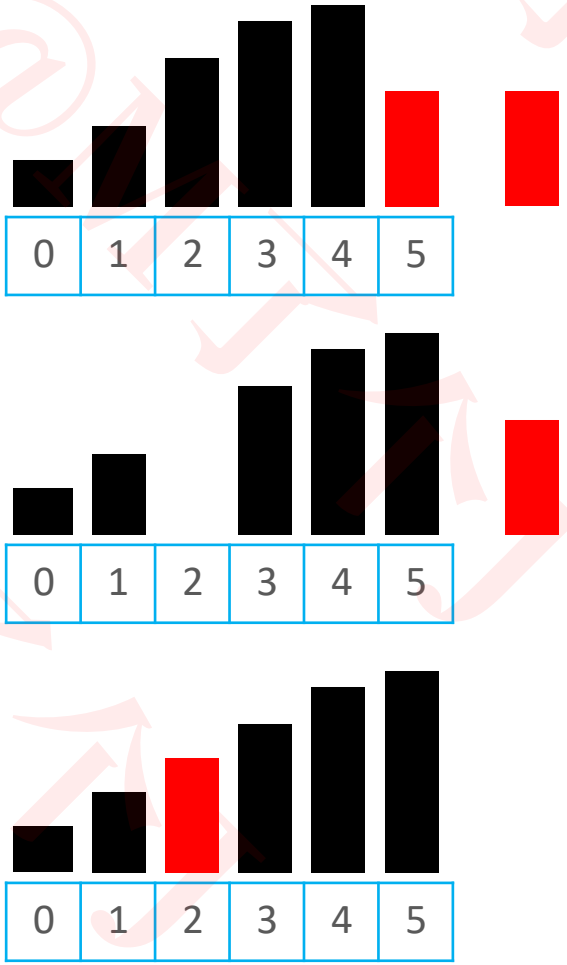
■ 思路是将【交换】转为【挪动】

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)

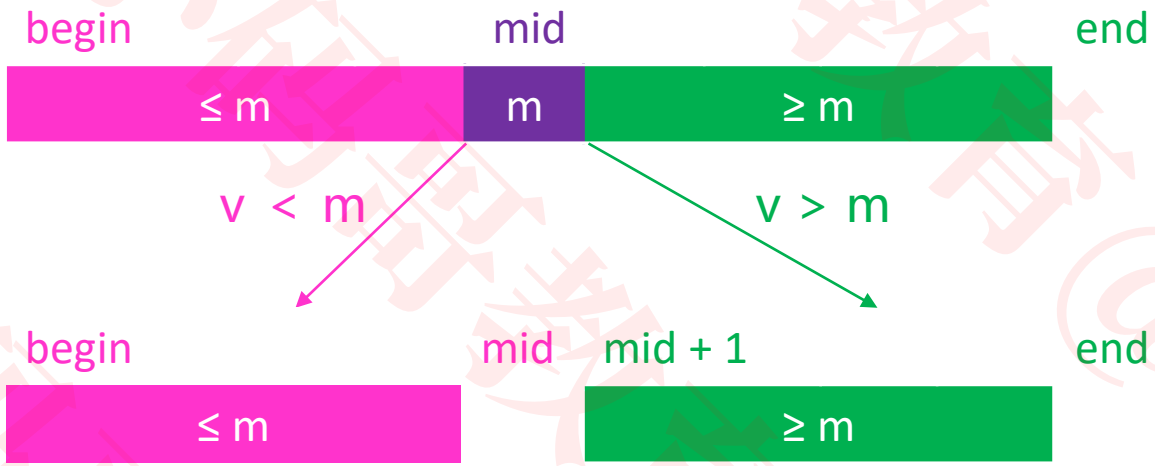
0	1	2	3	4	5	6	7	8	9
31	66	17	15	28	20	59	88	45	56

  - 如果是有序数组，可以使用二分搜索，最坏时间复杂度：O(logn)

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

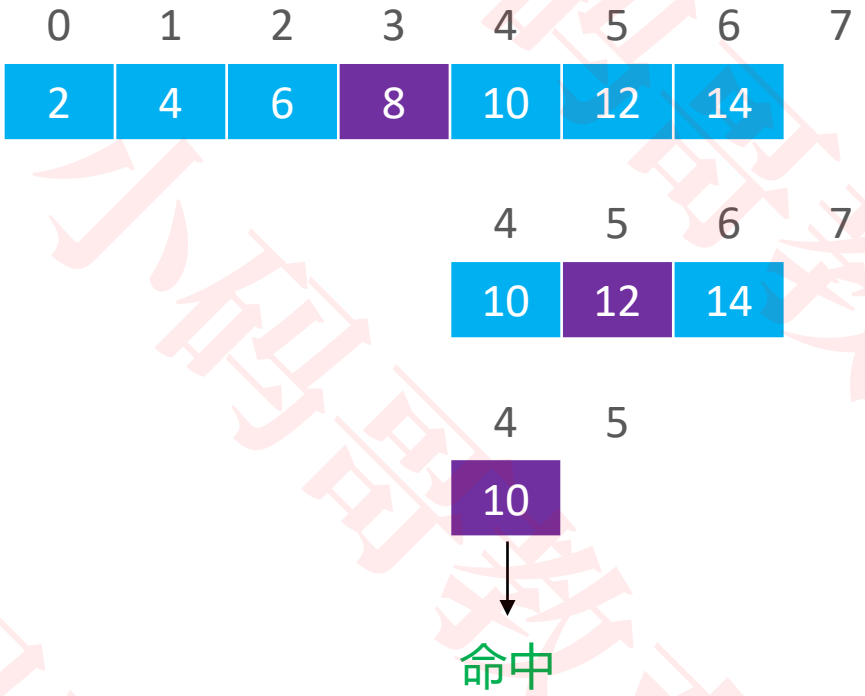
二分搜索 – 思路

- 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

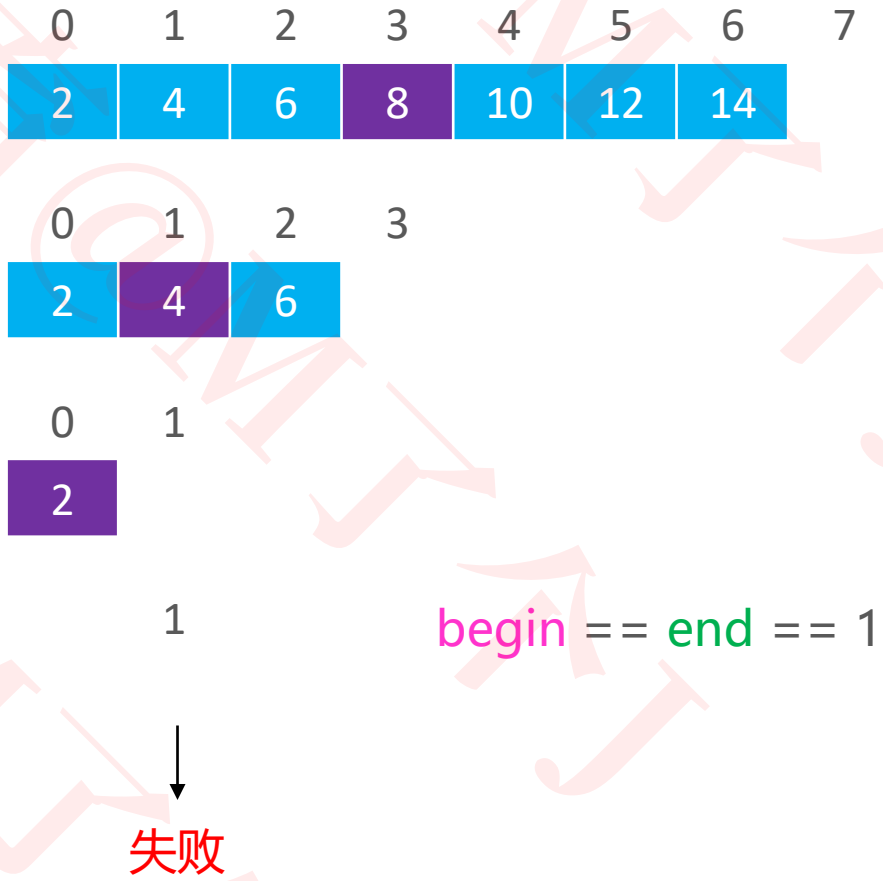


二分搜索 – 实例

■ 搜索 10



■ 搜索 3



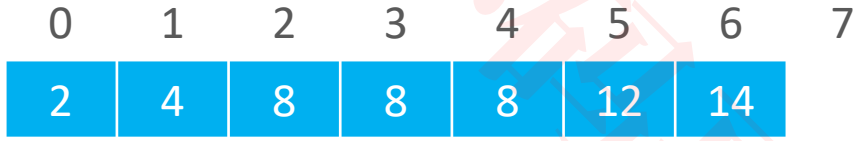
二分搜索 – 实现

```java
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

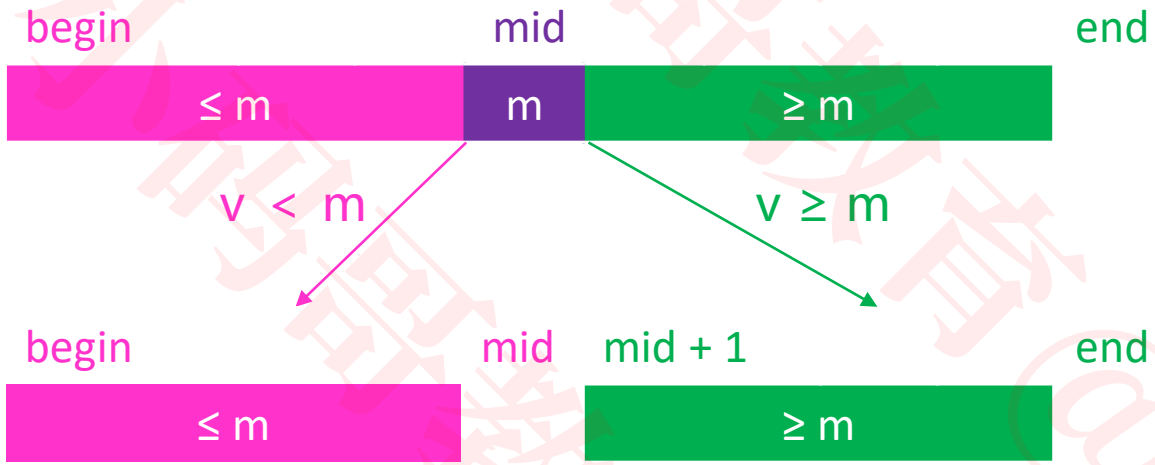
- 在元素 v 的插入过程中，可以先二分搜索出合适的插入位置，然后再将元素 v 插入



- 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

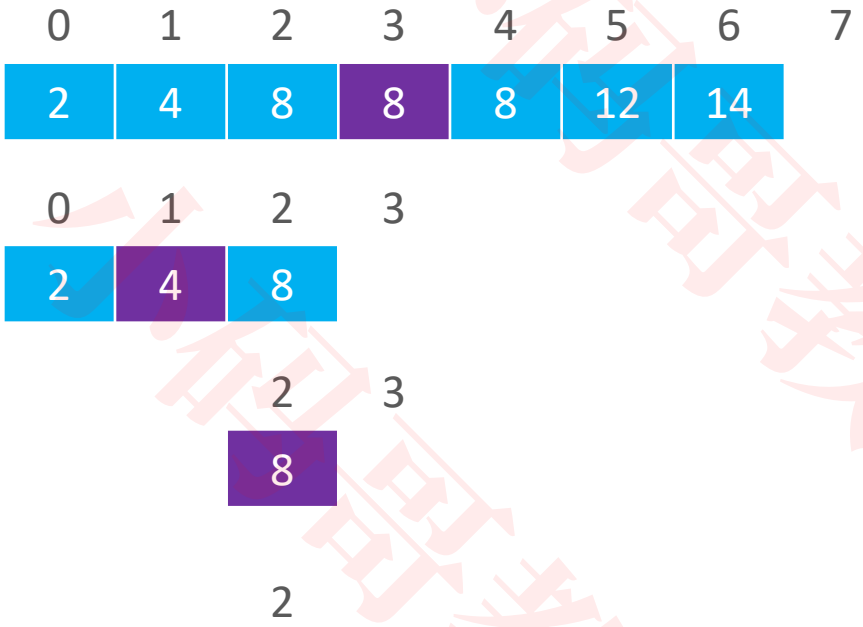
插入排序 – 二分搜索优化 – 思路

- 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 ≥ m，去 [mid + 1, end) 范围内二分搜索



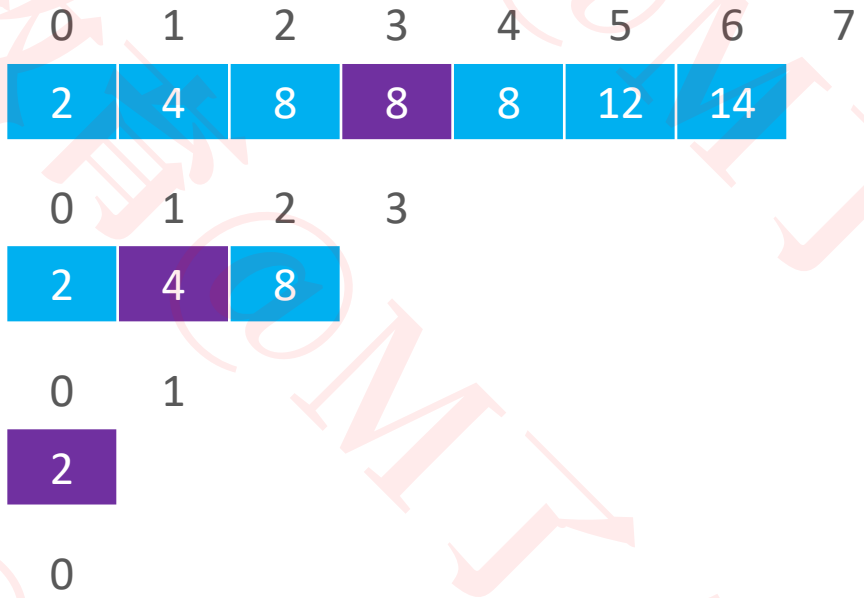
插入排序 – 二分搜索优化 – 实例

■ 搜索 5



begin == end == 2

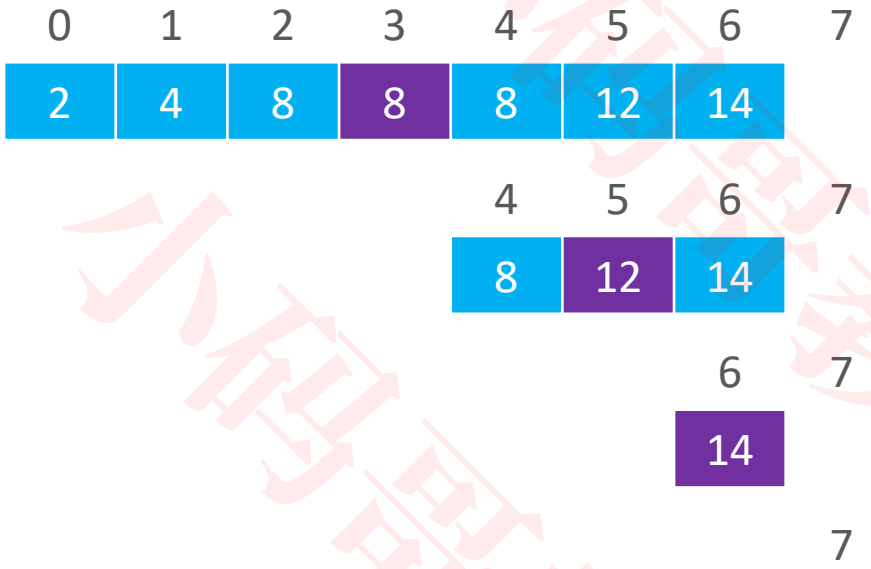
■ 搜索 1



begin == end == 0

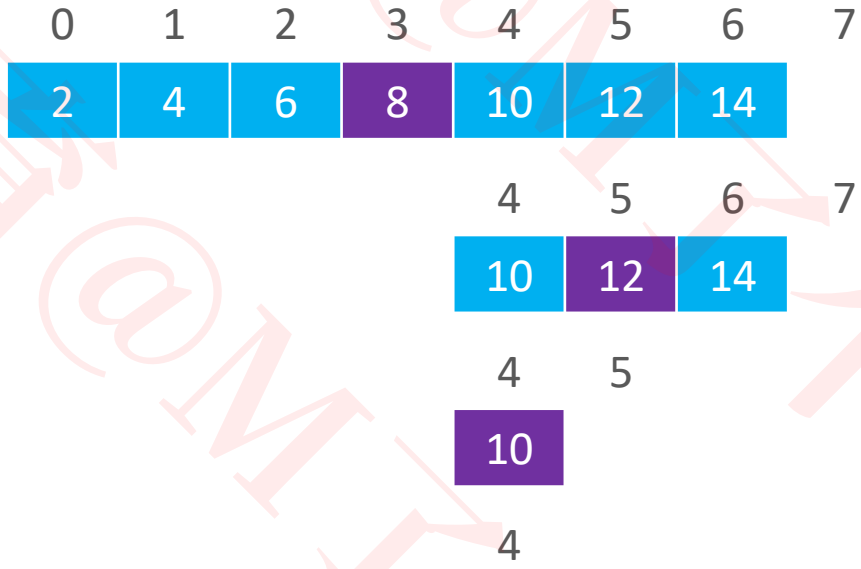
插入排序 – 二分搜索优化 – 实例

■ 搜索 15



begin == end == 7

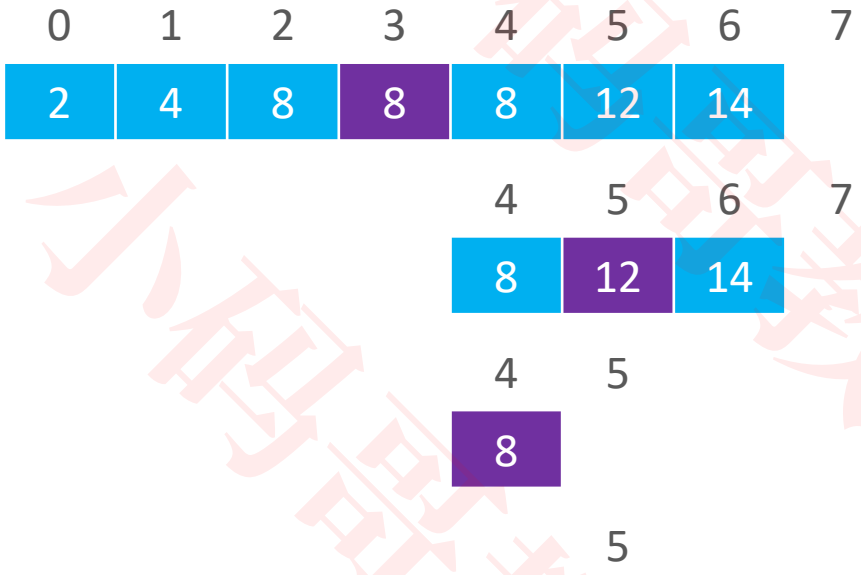
■ 搜索 8



begin == end == 4

插入排序 – 二分搜索优化 – 实例

■ 搜索 8



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 $O(n^2)$