计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：张羽丰

湖南大学 信息科学与工程学院

第 10 章

排序

高等教育出版社

提纲

10.1 问题引入

排序是指将数据按照关键字重新排列为升序（从小到大）或降序（从大到小）的处理。

- 升序：关键字从小到大
- 降序：关键字从大到小

示例：

待排序列：34 12 34’ 08 96
升序： 08 12 34’ 34 96
降序： 96 34 34’ 12 08

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。

稳定的排序算法有：

- 插入排序、冒泡排序、归并排序

不稳定的排序算法：

- 选择排序、快速排序、堆排序

示例：

待排序列：	34	12	34’	08	96
稳定：	08	12	34	34’	96
不稳定：	08	12	34’	34	96

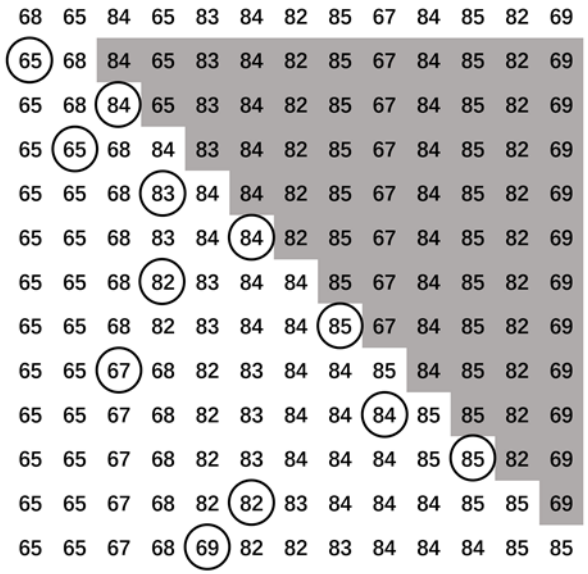
排序的分类

- **内部排序、外部排序**（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
- **比较、分配**（根据排序实现手段）
  - 基于“比较”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：通过将元素进行分配和收集进行排序
    - 基数排序、桶排序
- 根据实现的**难易**程度：
  - 基本排序：插入排序、冒泡排序、选择排序、……
  - 高级排序：快速排序、归并排序、堆排序、基数排序、……

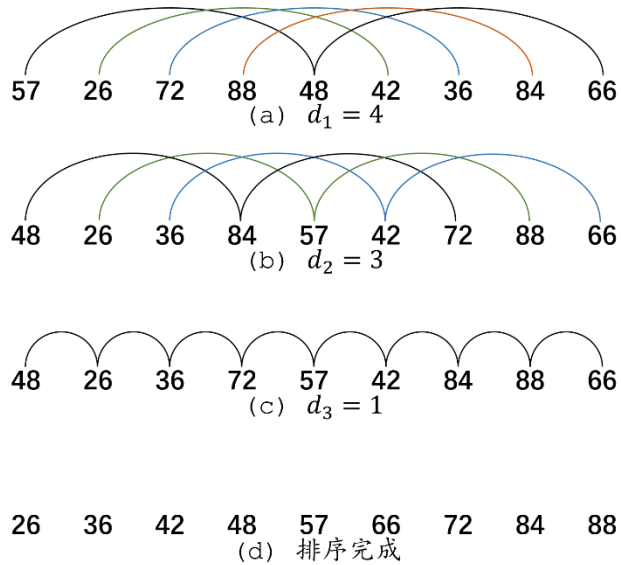
10.2 插入排序

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

折半插入排序和希尔排序是插入排序的两种改进



直接插入排序



希尔排序

10.2.1 直接插入排序

排序步骤：假设 a_0，a_1，…，a_{i-1} 已排序 $(a_0 < a_1 < \cdots < a_{i-1})$，对于$t = \boldsymbol{a_i}$，将他与$\boldsymbol{a_{i-1}, a_{i-2}, \ldots, a_0}$依次进行比较。

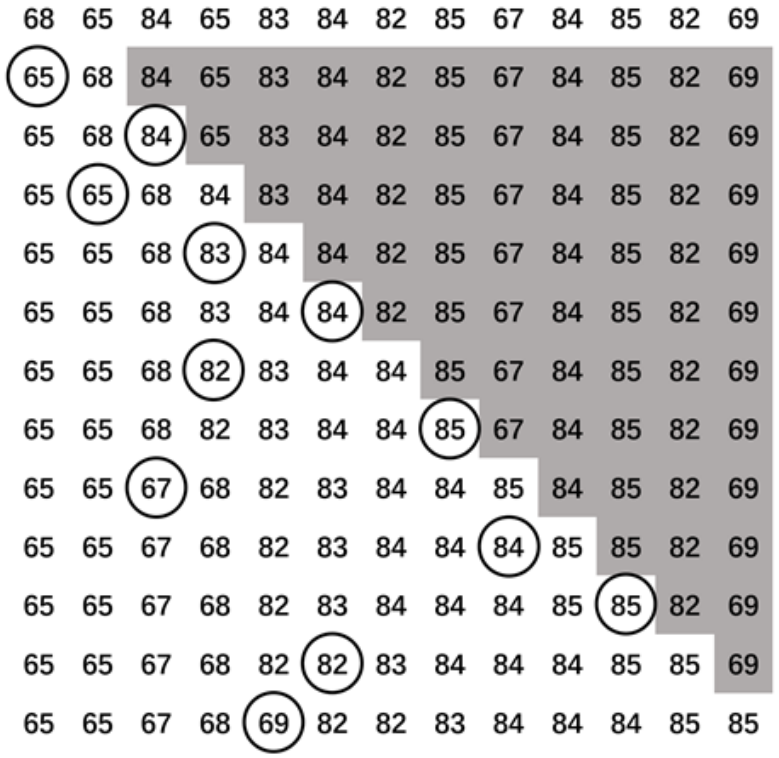
若 $\boldsymbol{a_j} > t$，则将 $\boldsymbol{a_j}$ 向后移动一位。直到发现某个$j(0 \le j \le i-1)$；

若 $\boldsymbol{a_j} \le t$，则令 $\boldsymbol{a_{j+1}} = t$，此时完成了将 $\boldsymbol{a_i}$ 插入有序数组的过程；

如果这样的a_j不存在，那么在比较过程中，$a_{i-1}, a_{i-2}, \ldots, a_0$都依次后移一个位置，此时令$a_0 = t$。

直接插入排序过程

对关键字序列{68,65,84,65,83,84,82,85,67,84,85,82,69}进行升序排列



直接插入排序伪代码

算法10-1 插入排序 InsertionSort(a, l, r)

输入： 序列a，左端点下标l，右端点下标r

输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1   for i ← l + 1 to r do //从左边界开始，依次获取每个记录
2   | t ← a_i
3   | for j ← i − 1 downto l do
4   | | if a_j > t then
5   | | | a_{j+1} ← a_j //若当前记录小，则把前面的记录向后移一个位置
6   | | else
7   | | | a_{j+1} ← t //将最初获取的记录复制到相应位置
8   | | | break
9   | | end
10  | end
11  | if a_l > t then //如果a_l > t，将最初获取的记录置于序列开始
12  | | a_l ← t
13  | end
14  end
```

直接插入排序性能分析

时间复杂度：

最佳情况（有序）：$\boldsymbol{O(n)}$

- $n-1$ 次比较，0 次移动.

最坏情况（逆序）：$\boldsymbol{O(n^2)}$

- 插入第 i 个元素时，每次都需要比较前 $i-1$ 个元素才能找到插入位置。
- $\frac{n(n-1)}{2}$ 次比较，$\frac{(n+2)(n-1)}{2}$ 次移动.

平均情况：$\boldsymbol{O(n^2)}$

- 比较次数和记录移动次数约为$\frac{n^2}{4}$
- $O(n^2)$ 次比较，$O(n^2)$ 次移动

直接插入排序性能分析

空间复杂度：$O(1)$

对于每次排序，仅申请一个临时变量用于存储第i个元素，空间复杂度为$O(1)$

稳定性分析：**稳定**

对于值相同的元素，可以选择将后面出现的元素，插入到前面出现元素的后面，这样就可以保持原有的前后顺序不变，所以插入排序是**稳定的**排序算法。

适用场景：节点个数少。

10.2.2 折半插入排序

折半插入排序是对直接插入排序的一种改进。当待排序序列较长时，如果采用折半查找的方法，可以更快地寻找插入位置，减少关键码的比较次数。

折半插入排序虽然可以减少关键码的比较次数，但是并不减少排序元素的移动次数。比较次数为$O(nlogn)$，移动次数为$O(n^2)$

10.2.3 希尔排序

基本思想：相较于插入排序，对位置相隔较大距离的元素进行比较，使得元素在比较后能够一次跨过较大的距离。

排序步骤：

- 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

希尔排序伪代码

算法10-2 希尔排序 $\text{ShellSort}(a, l, r)$

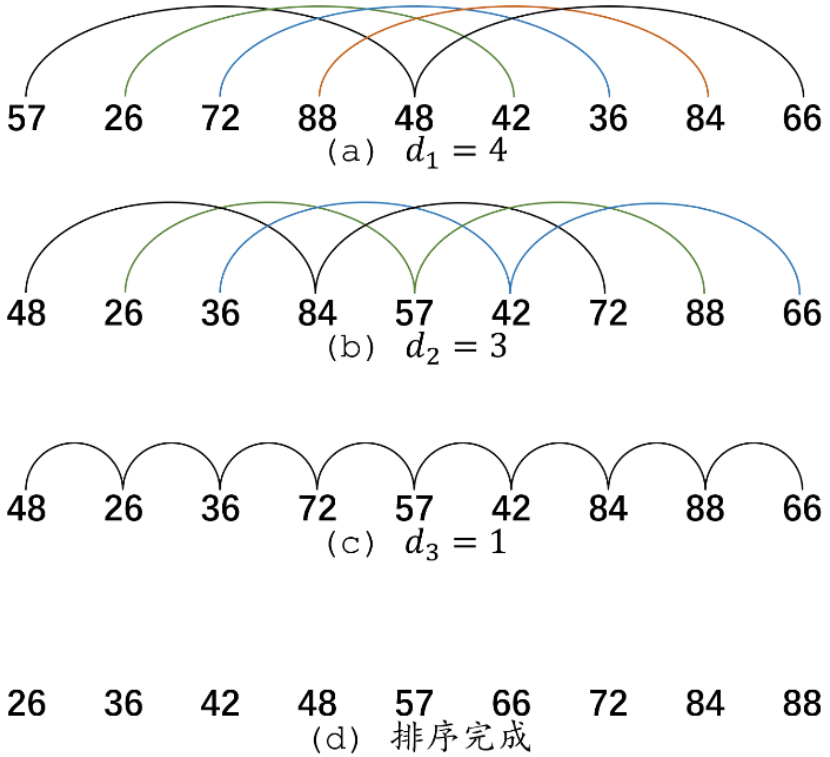
输入： 序列a，左端点下标l，右端点下标r

输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1   d ← {d_1, d_2, ..., d_t}  //希尔排序时使用的步长，根据不同步长循环进行插入排序
2   for m ← 1 to |d| do  //步长的个数
3   |  for i ← l + d_m to r do  //每个元素都往前插入排序一次
4   |  |  v ← a_i
5   |  |  for j ← i downto l + d_m step d_m do  //对序列a_j, a_{j-d_m}, ...进行插入排序，本组内
6   |  |  |  if a_{j-d_m} > v then
7   |  |  |  |  a_j ← a_{j-d_m}
8   |  |  |  else
9   |  |  |  |  a_j ← v
10  |  |  |  |  break
12  |  |  |  end
13  |  |  end
14  |  end
15  end
```

希尔排序过程

对关键字序列{57,26,72,88,48,42,36,84,66}进行升序排列，设置增量序列为$d = \{4,3,1\}$



希尔排序性能分析

时间复杂度：

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)
$2^k - 1$	$(n^{3/2})$
$2^i 3^i$	$\boldsymbol{(n\log^2 n)}$

空间复杂度： $\boldsymbol{O(1)}$

稳定性分析： **不稳定**

10.3 选择排序

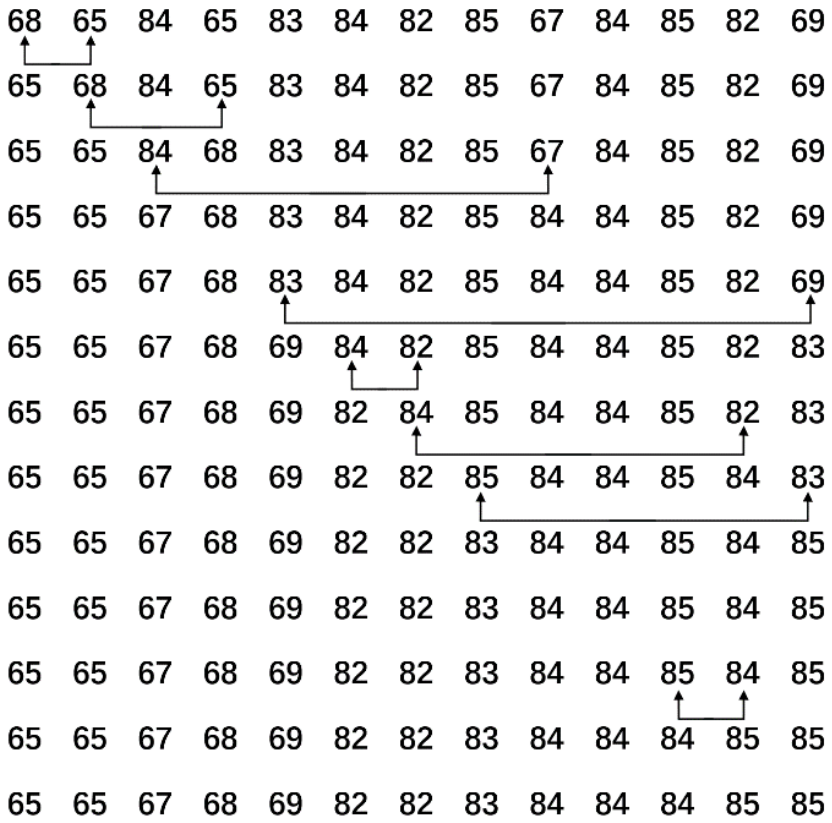
基本思想：首先选出键值**最小**的项，将它与**第一个项**交换位置；然后选出键值**次小**的项，将其与**第二个项**交换位置；…；直到整个序列完成排序。

步骤：

- 假设待排序的序列为$a_0, a_1, a_2, \cdots, a_{n-1}$
- 依次对$i = 0, 1, \cdots, n-2$执行如下步骤：
  - 在$a_i, a_{i+1}, \cdots, a_{n-1}$中选择一个键值最小的项$a_k$
  - 将a_i与a_k交换。

10.3.1 简单选择排序

简单选择排序在从待排序序列中选出键值最小的项时，所用的策略是简单的逐个枚举法。



简单选择排序过程

简单选择排序伪代码

算法10-3 简单选择排序 $SelectionSort(a,l,r)$

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1     for i ← l to r − 1 do //依次从剩余未排序序列中选取一个最小的记录
2     |  min ← i
3     |  for j ← i + 1 to r do
4     |  |  if a_j < a_min then
5     |  |  |  min ← j
6     |  |  end
7     |  |  t ← a_i //将当前的最小记录放入已排序好的队列的末尾
8     |  |  a_i ← a_min
9     |  |  a_min ← t
10    |  end
11    end
```

简单选择排序性能分析

时间复杂度：

最佳情况（有序）：$\boldsymbol{O(n^2)}$

- $\frac{n(n-1)}{2}$ 次比较，0 次移动.

最坏情况（逆序）：$\boldsymbol{O(n^2)}$

- $\frac{n(n-1)}{2}$ 次比较，$n-1$ 次移动.

平均情况：$\boldsymbol{O(n^2)}$

- $O(n^2)$ 次比较，$O(n)$ 次移动

简单选择排序性能分析

空间复杂度：$O(1)$

排序时仅交换未排序序列中最值与未排序序列起始位置，不涉及额外空间的使用。

稳定性分析：不稳定

交换操作时可能会破坏具有相同key值元素的相对位置。因此选择排序为不稳定排序。

运行时间与记录顺序关系很小

交换次数很少

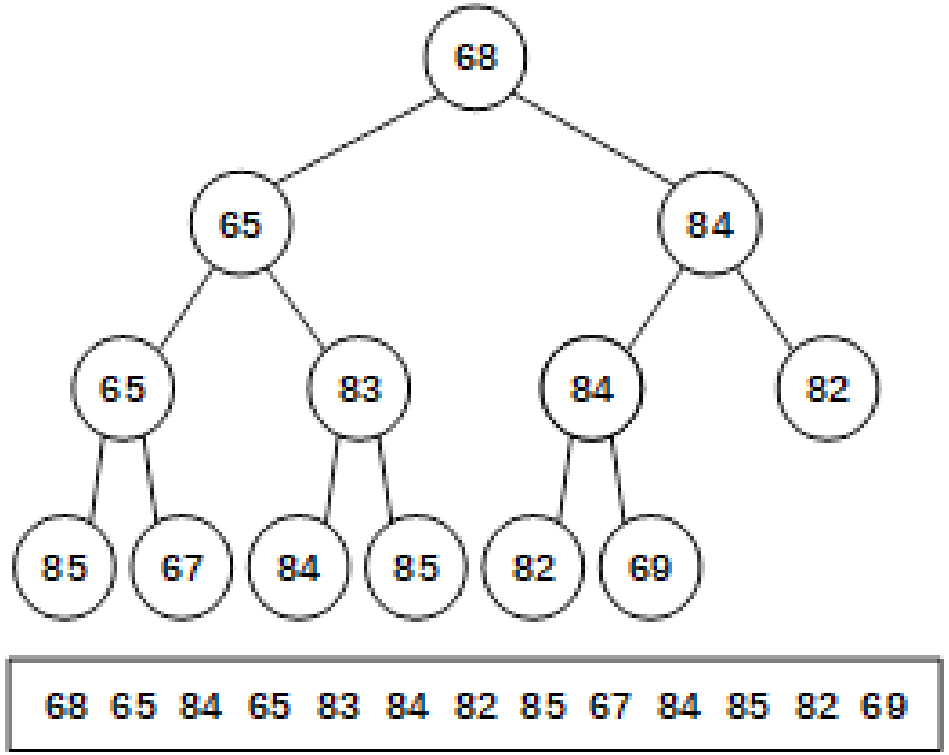
10.3.2 堆排序

基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

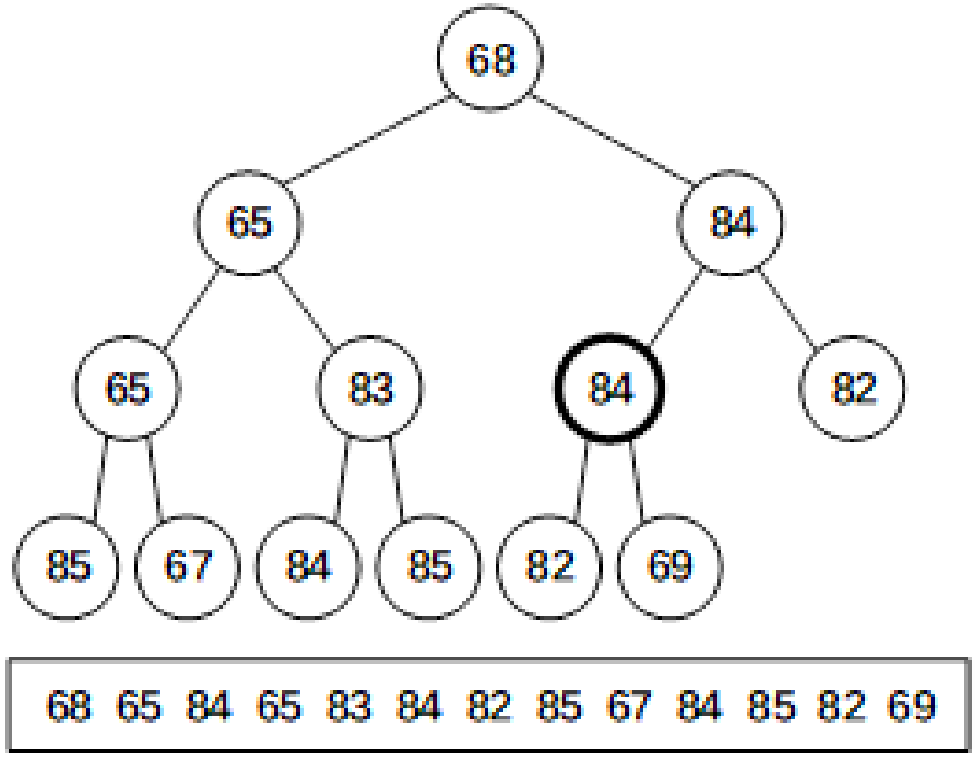
排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。

初始堆建立过程

下沉方法建堆



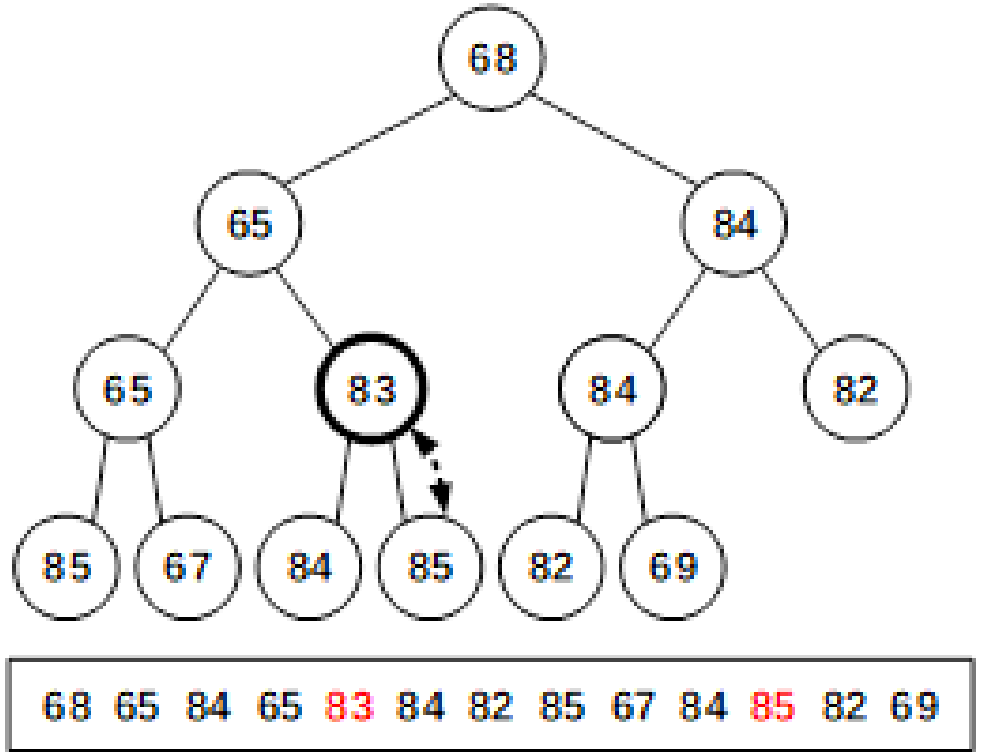
（0）初始状态



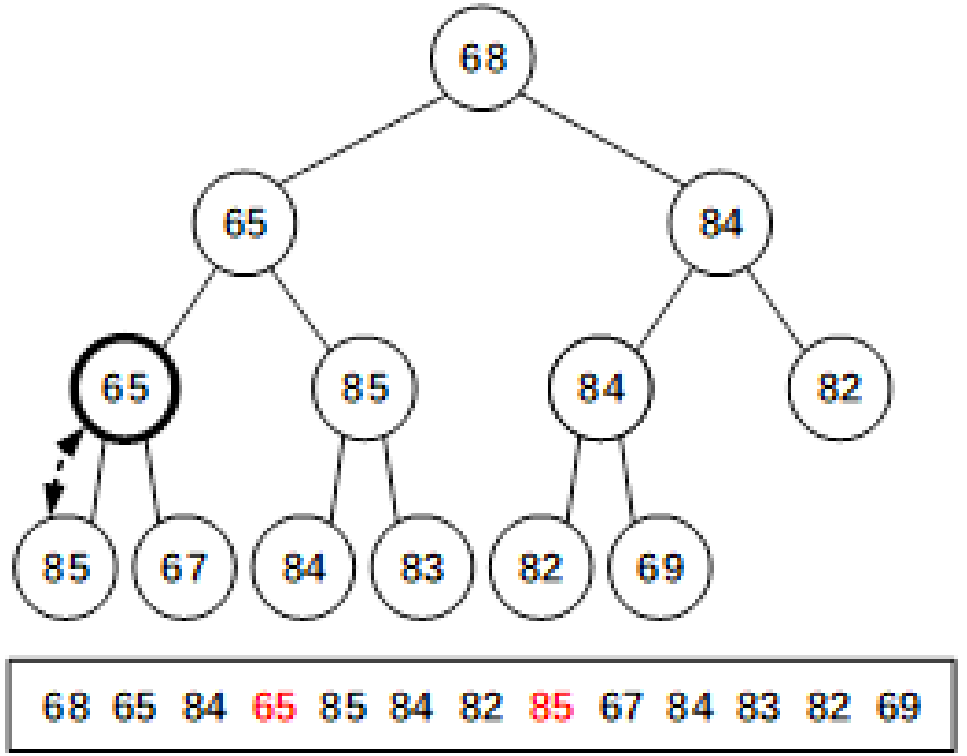
（1）堆修复：无动作

初始堆建立过程

下沉方法建堆



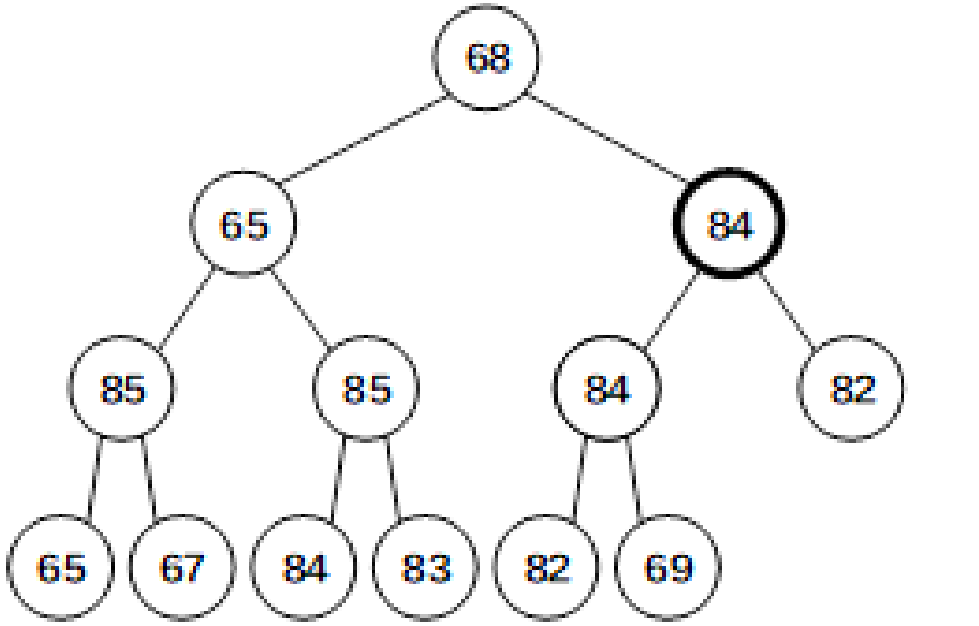
（2）堆修复：调整堆顶



（3）堆修复：调整堆顶

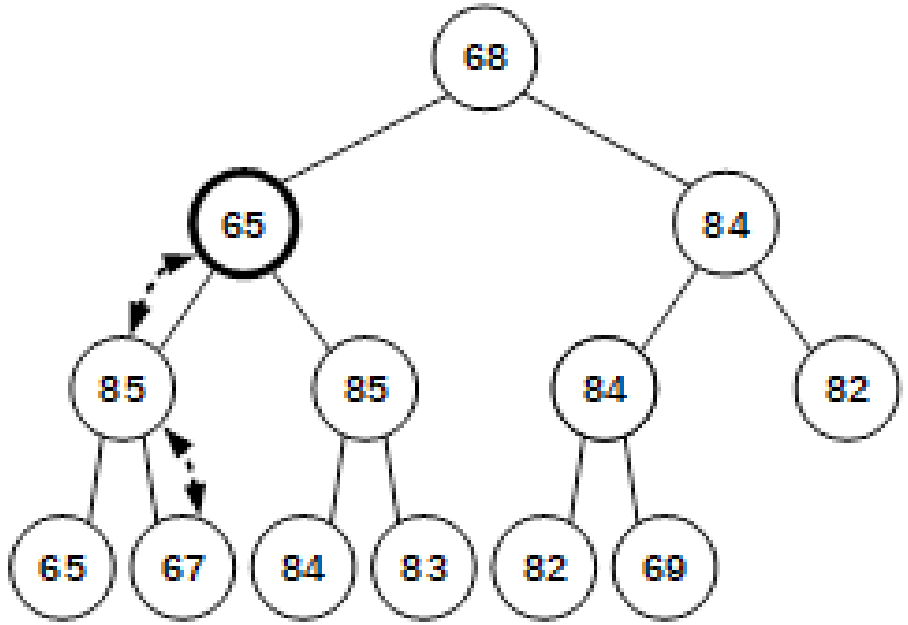
初始堆建立过程

下沉方法建堆



68 65 84 85 85 84 82 65 67 84 83 82 69

（4）堆修复：无动作

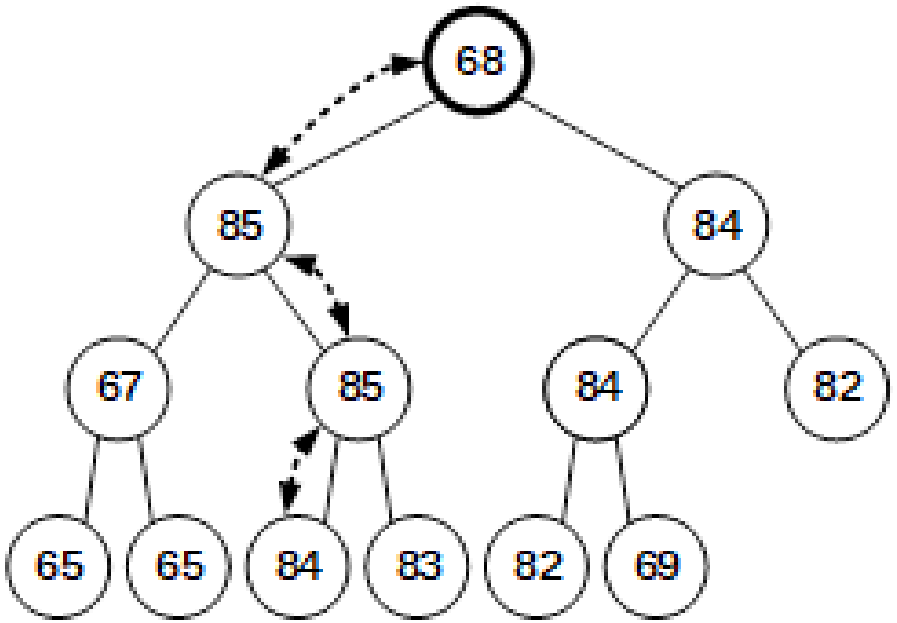


68 65 84 85 85 84 82 65 67 84 83 82 69

（5）堆修复：调整堆顶和堆内修复

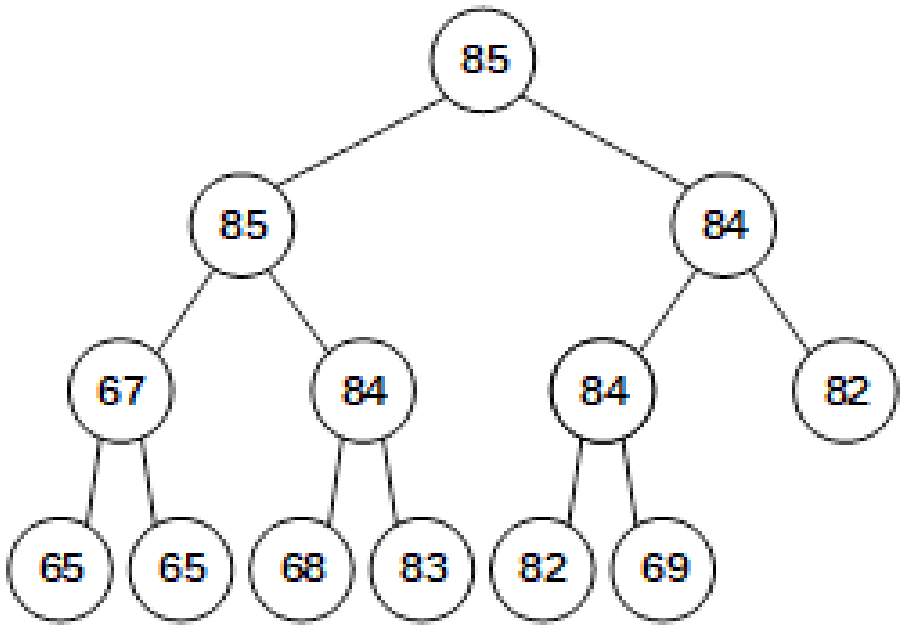
初始堆建立过程

下沉方法建堆



68 85 84 67 85 84 82 65 65 84 83 82 69

（6）堆修复：调整堆顶和堆内修复

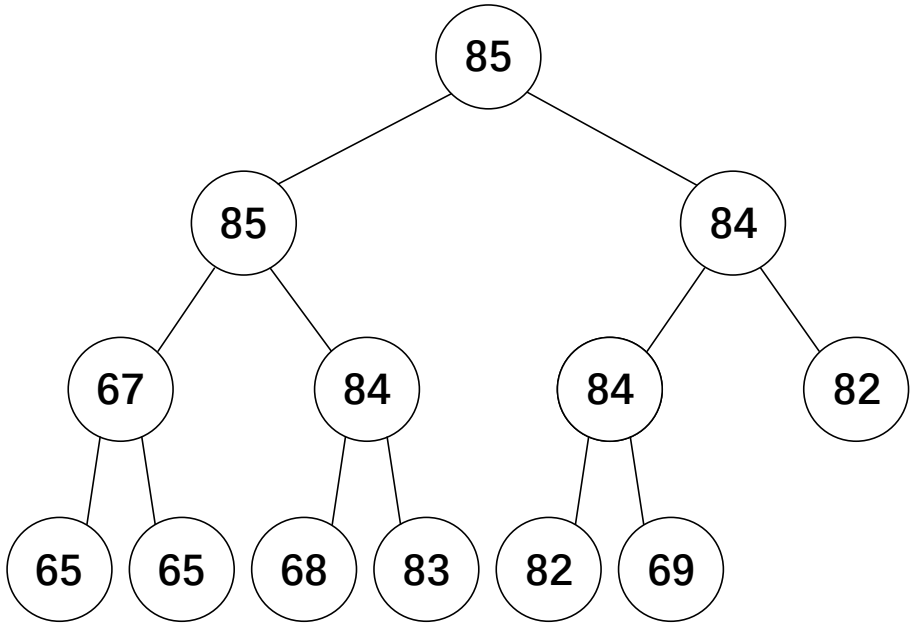


85 85 84 67 84 84 82 65 65 68 83 82 69

（7）修复完成，获得最终状态

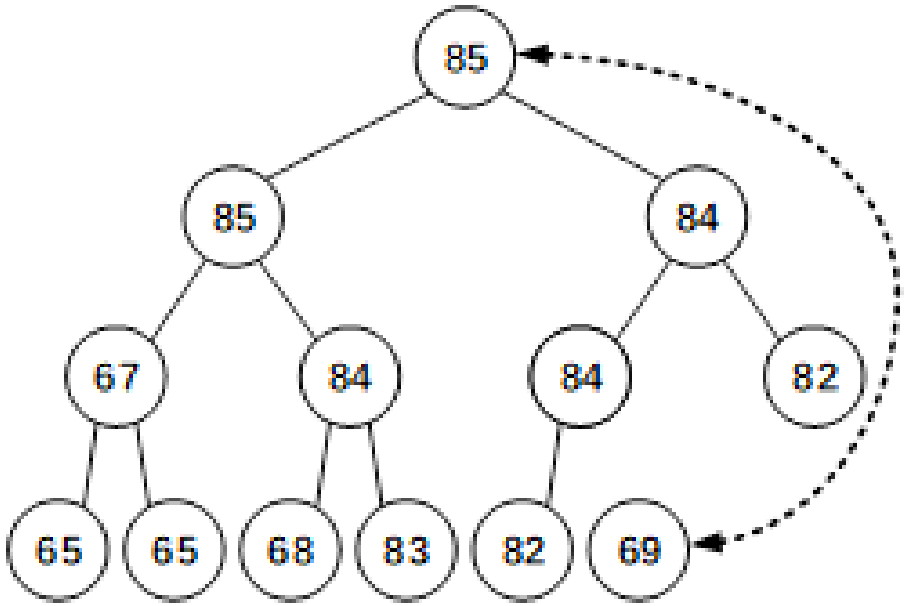
堆排序过程

取出当前堆中最大值，放到最后面



85 85 84 67 84 84 82 65 65 68 83 82 69

（0）初始状态

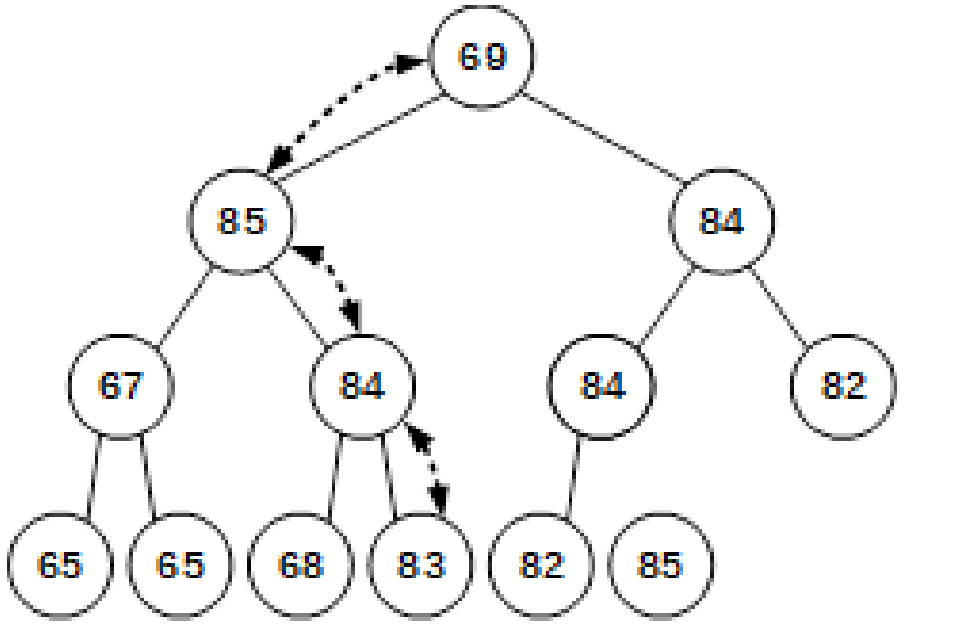


85 85 84 67 84 84 82 65 65 68 83 82 69

（1）堆顶调整

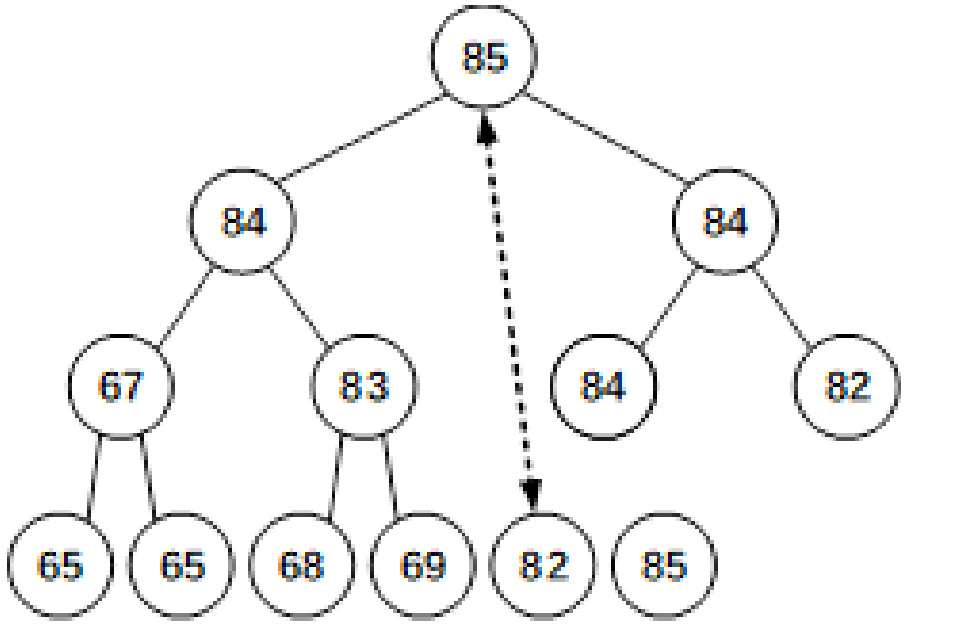
堆排序过程

取出当前堆中最大值，放到最后面



69 85 84 67 84 84 82 65 65 68 83 82 85

（2）堆修复 范围$1 \sim N-1$

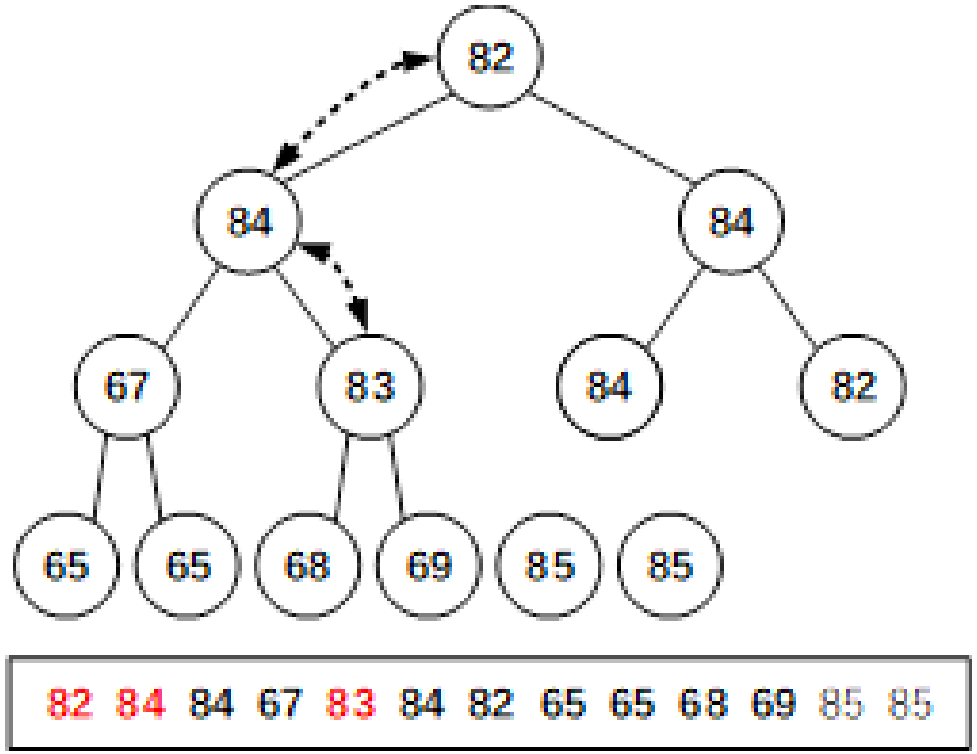


85 84 84 67 83 84 82 65 65 68 69 82 85

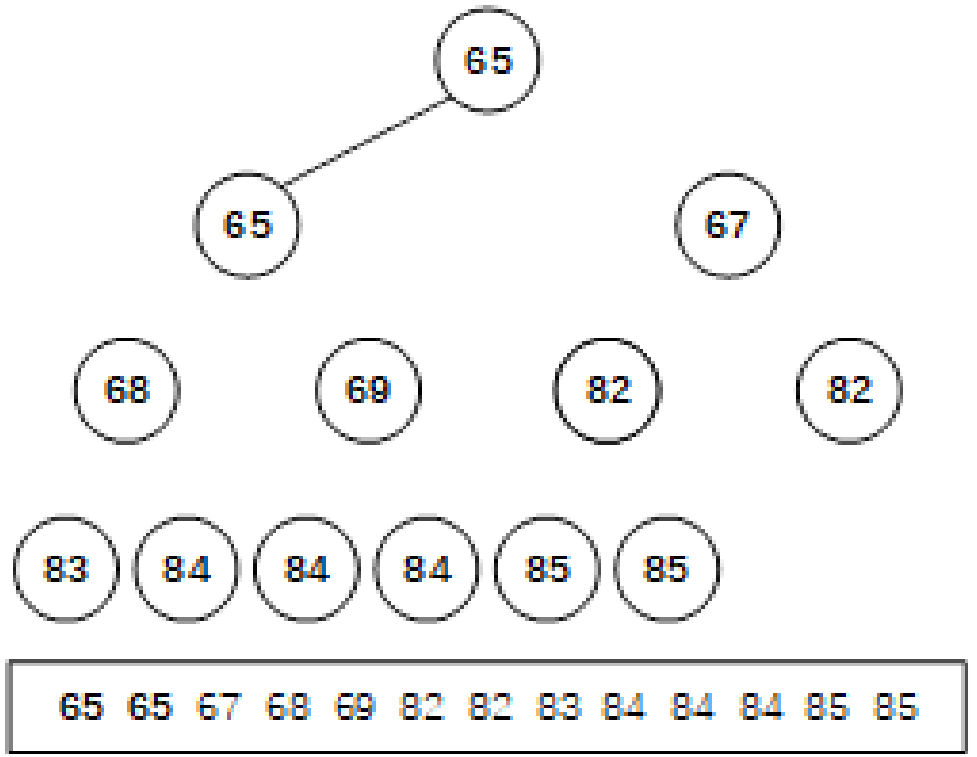
（3）堆顶调整

堆排序过程

取出当前堆中最大值，放到最后面



（4）堆修复 范围$1\sim N-2$



（5）堆修复 范围$1\sim 2$

堆排序伪代码

算法10-4 堆排序 $\mathrm{HeapSort}(a, l, r)$

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1  n ← r − l + 1
2  MakeHeapDown(⟨a_l, ..., a_r⟩)      //建最大堆
3  while n > 1 do  //基于堆的排序
4  |  t ← a_l  //堆顶
5  |  a_l ← a_{l+n−1}  //堆顶与要放到后面的位置的元素交换
6  |  a_{l+n−1} ← t
7  |  n ← n − 1
8  |  SiftDown(⟨a_l, ..., a_{l+n−1}⟩, l)  //交换后的堆顶下沉
9  end
```

堆排序性能分析

时间代价：$O(n \log n)$

对n个元素进行堆排序，建立初始堆需要线性时间，即时间复杂度为$O(n)$，排序过程中需要的比较次数至多为$2n \log n$，因为每下沉一次，要与两个子节点中较大的交换，因而比较两次，下沉次数为log n。因此总的时间复杂度为$O(n \log n)$。

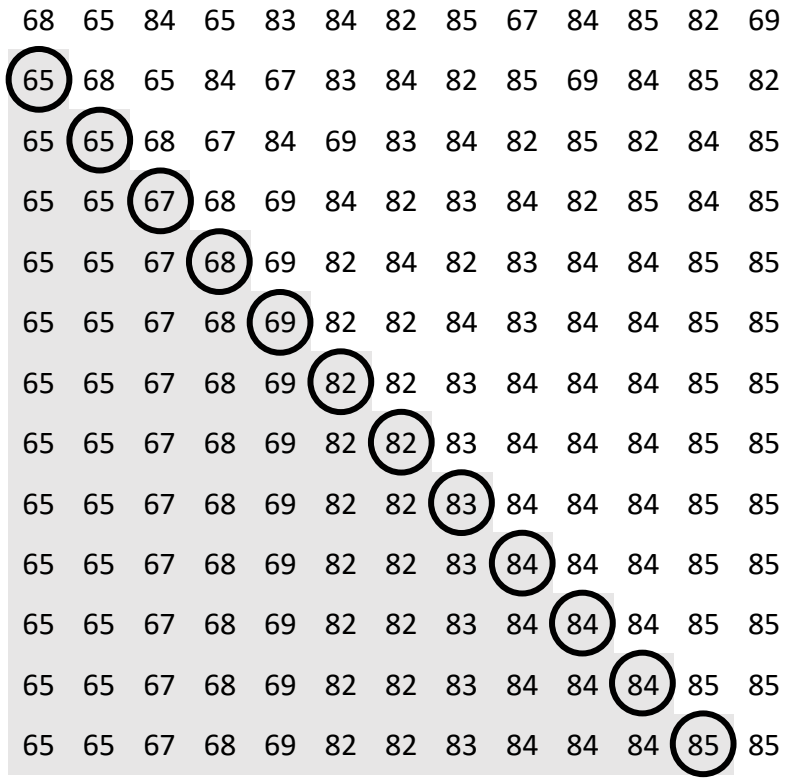
空间代价：$O(1)$

稳定性：不稳定

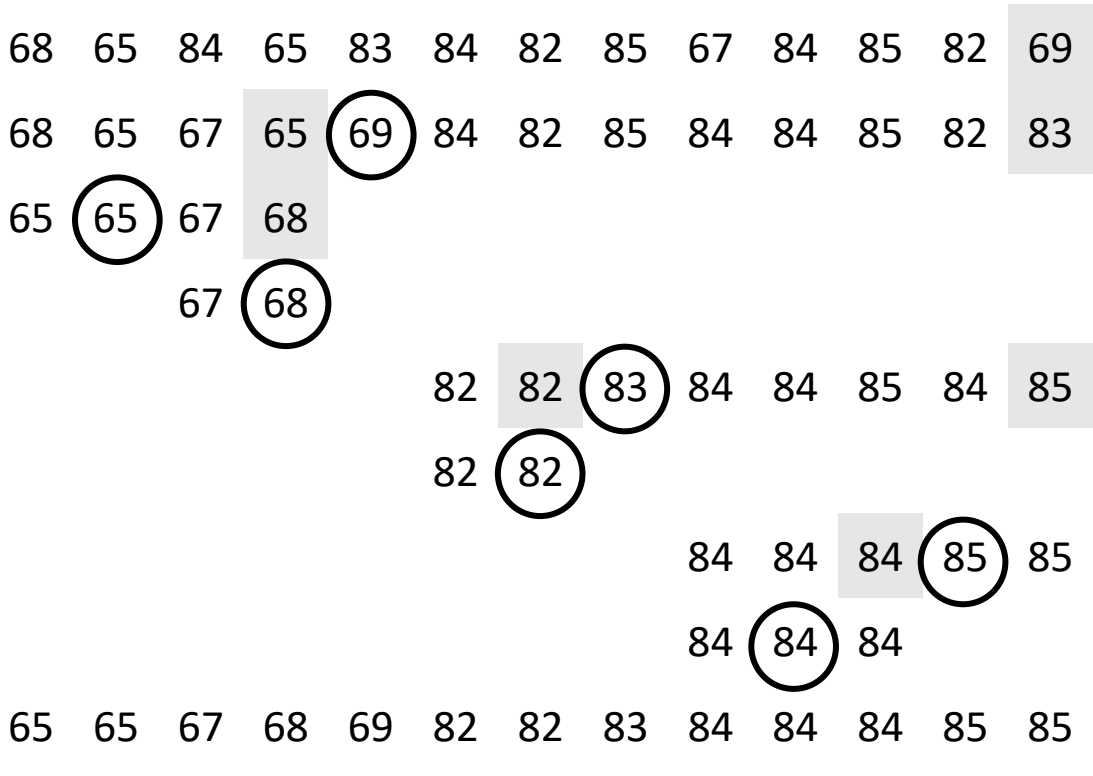
在堆排序中，会将**堆顶元素与数组末尾元素**进行**交换**，这一操作会**破坏**数组的稳定性，故堆排序是不稳定排序。

10.4 交换排序

核心思路：对序列中的元素进行多次两两交换，从而使序列元素有序。



冒泡排序



快速排序

10.4.1 冒泡排序

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。

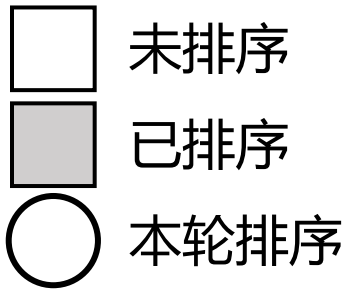
通过比较与交换使得待排序列一个**最值**元素“上浮”到序列**一端**，然后缩小排序范围。

排序步骤：（以升序为例）

- 假设待排序序列为 $a_0, a_1, \cdots, a_{n-1}$。
- 起始时排序范围为 a_0 到 a_{n-1}。
- **从右向左**对相邻两元素进行比较，较大值向右移，较小值向左移。比较完当前排序范围后，键值**最小**的元素被移动到 a_0 位置。
- 下一次排序范围是 a_1 到 a_{n-1}。

冒泡排序示例

待排序数组	68	65	84	65	83	84	82	85	67	84	85	82	69
第一轮	**65**	68	65	84	67	83	84	82	85	69	84	85	82
第二轮	65	**65**	68	67	84	69	83	84	82	85	82	84	85
第三轮	65	65	**67**	68	69	84	82	83	84	82	85	84	85
第四轮	65	65	67	**68**	69	82	84	82	83	84	84	85	85
第五轮	65	65	67	68	**69**	82	82	84	83	84	84	85	85
第六轮	65	65	67	68	69	**82**	82	83	84	84	84	85	85
第七轮	65	65	67	68	69	82	**82**	83	84	84	84	85	85
第八轮	65	65	67	68	69	82	82	**83**	84	84	84	85	85
第九轮	65	65	67	68	69	82	82	83	**84**	84	84	85	85
第十轮	65	65	67	68	69	82	82	83	84	**84**	84	85	85
第十一轮	65	65	67	68	69	82	82	83	84	84	**84**	85	85
第十二轮	65	65	67	68	69	82	82	83	84	84	84	**85**	85



冒泡排序伪代码

算法10-5 冒泡排序 BubbleSort(a,l,r)

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l,a_{l+1},\ldots,a_r$元素顺序，使元素按照非递减顺序排列

```
1  for i ← l to r do //冒泡次数
2  |  for j ← r − 1 downto i do //从右往左冒泡
3  |  |  if a_j > a_{j+1} then
4  |  |  |  Swap(a_j, a_{j+1})  //两个元素交换位置
5  |  |  end
6  |  end
7  end
```

冒泡排序性能分析

假设待排序的序列长度为n，那么冒泡排序共执行n轮，第i轮时最多需要执行$n-i$次比较和交换。因此，冒泡排序最多的比较和交换次数为：

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

所以，冒泡排序的时间复杂度为$\boldsymbol{O(n^2)}$。

冒泡排序仅对数组中的相邻元素进行比较和交换，因此关键字值相同的元素不会改变顺序。所以，冒泡排序是**稳定**的。

但注意，一旦将算法10-5中比较运算$a_j > a_{j+1}$改为$a_j \geq a_{j+1}$，算法就失去了稳定性。

另外，冒泡排序中的交换次数又称为反序数或逆序数，可用于体现数列的错乱程度。

10.4.2 快速排序

快速排序是一种所需**比较次数较少、速度较快**的排序方法，由英国计算机科学家，图灵奖获得者东尼·霍尔于1961年发表。在基于比较的排序算法中，快速排序的**平均性能突出**。

基本思想：通过递归分治方法，基于轴点将待排序序列拆分成两个子序列并分别排序，直到序列有序。

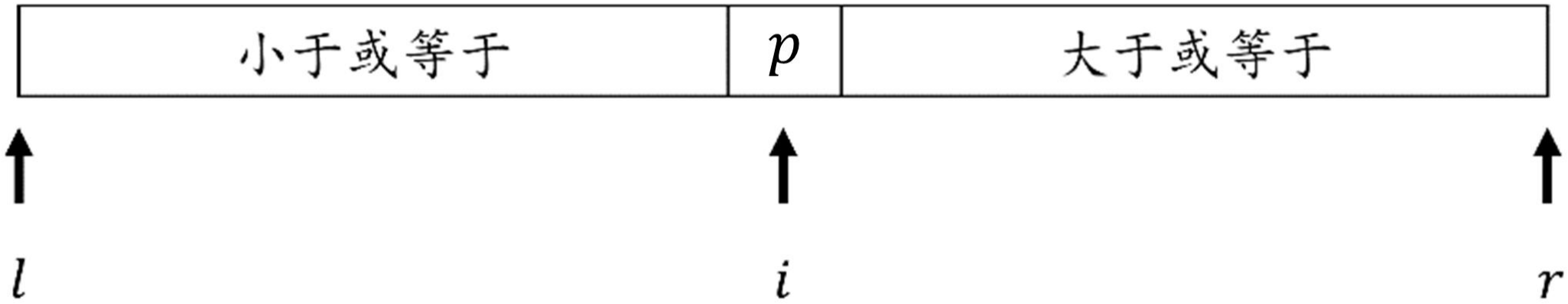
排序步骤：

- 从待排序序列中**选取轴点**。
- 通过交换序列元素，将待排序序列**拆分**为左右两个子序列，左子序列元素**小于等于**轴点，右子序列元素**大于等于**轴点。
- 对两个子序列**递归**进行上述操作，直到子序列**元素个数小于等于1**。

快速排序的序列拆分

序列拆分（Partition）会根据所选**轴点**将序列拆分成两个子序列，进而确定轴点在序列的**最终位置**。

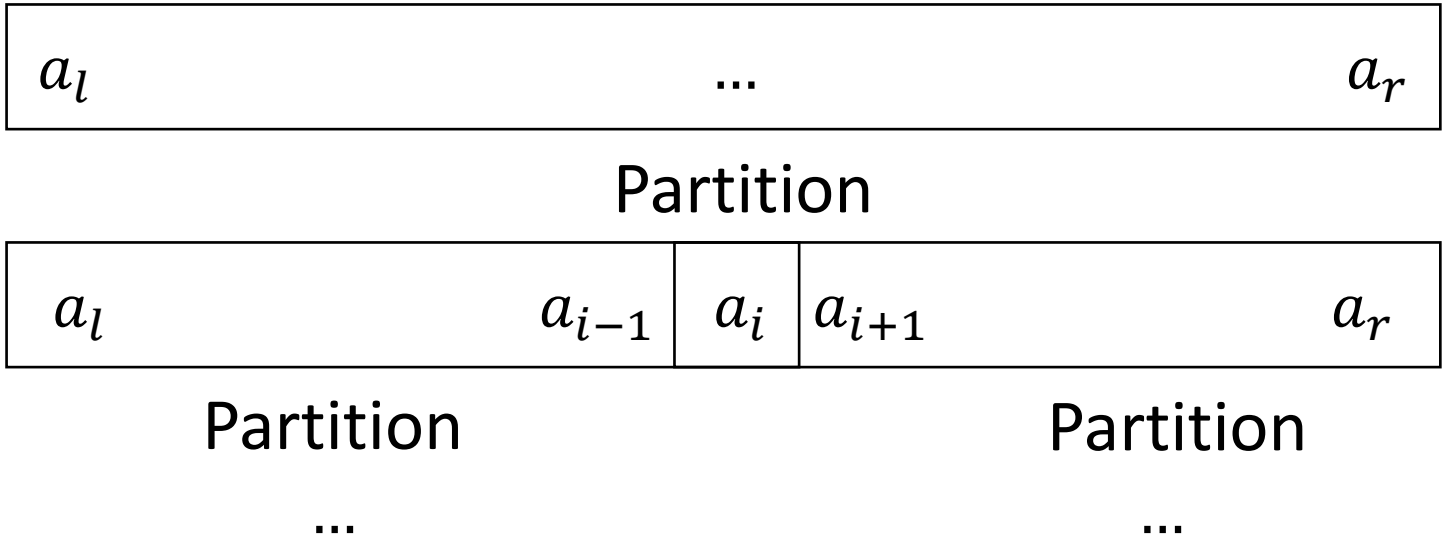
对于序列$a_l, \ldots, a_r$和轴点元素p，我们需要交换序列中的元素，将轴点p置于正确的位置a_i，使得$\boldsymbol{a_l, \ldots, a_{i-1} \le a_i \le a_{i+1}, \ldots, a_r}$，即$\{a_l, \ldots, a_{i-1}\}$均小于等于轴点，$\{a_{i+1}, \ldots, a_r\}$均大于等于轴点，并返回轴点所在下标$i$。



快速排序的序列拆分

序列拆分（Partition）会根据所选**轴点**将序列拆分成两个子序列，进而确定轴点在序列的**最终位置**。

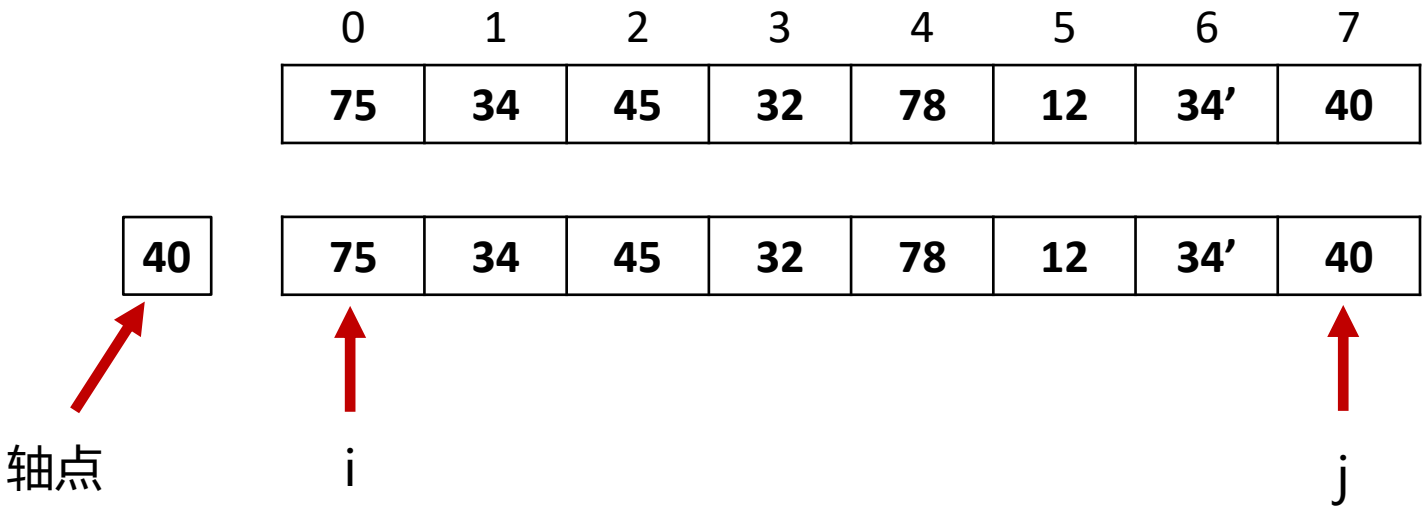
对于序列$a_l, \dots, a_r$和轴点元素p，我们需要交换序列中的元素，将轴点p置于正确的位置a_i，使得$\boldsymbol{a_l, \dots, a_{i-1} \le a_i \le a_{i+1}, \dots, a_r}$，即$\{a_l, \dots, a_{i-1}\}$均小于等于轴点，$\{a_{i+1}, \dots, a_r\}$均大于等于轴点，并返回轴点所在下标$i$。



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

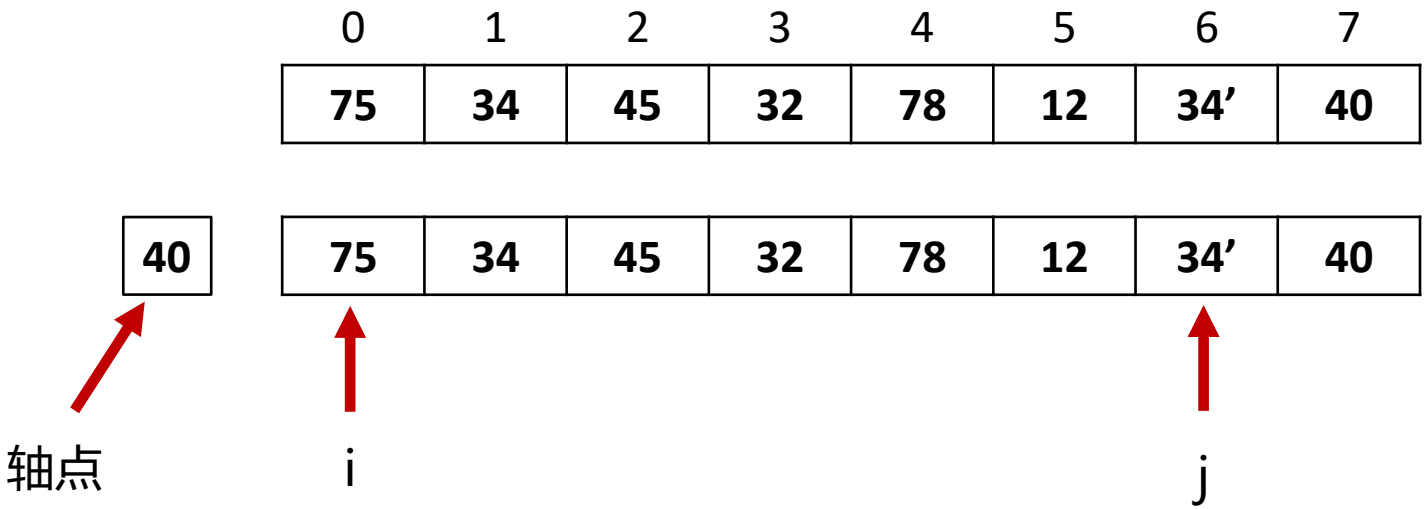
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

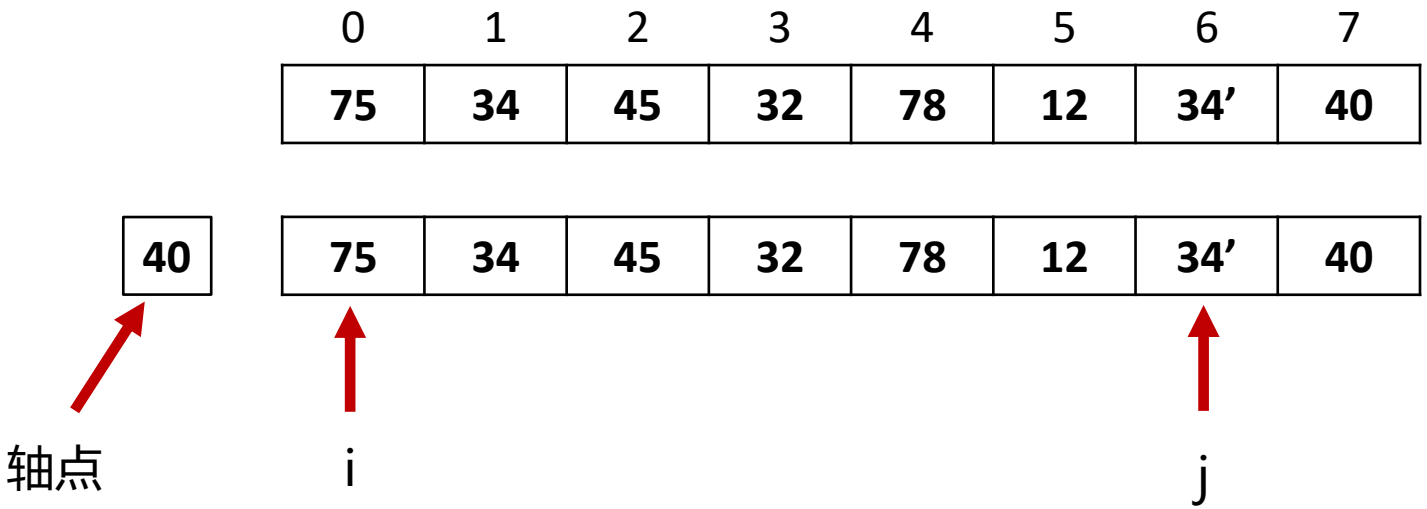
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

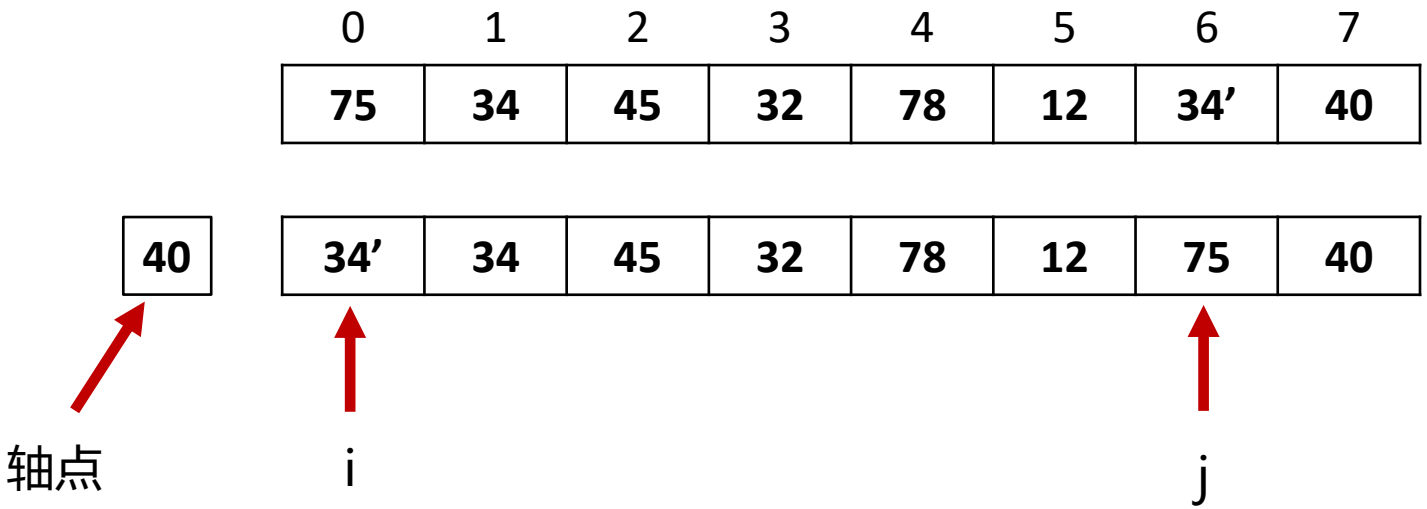
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

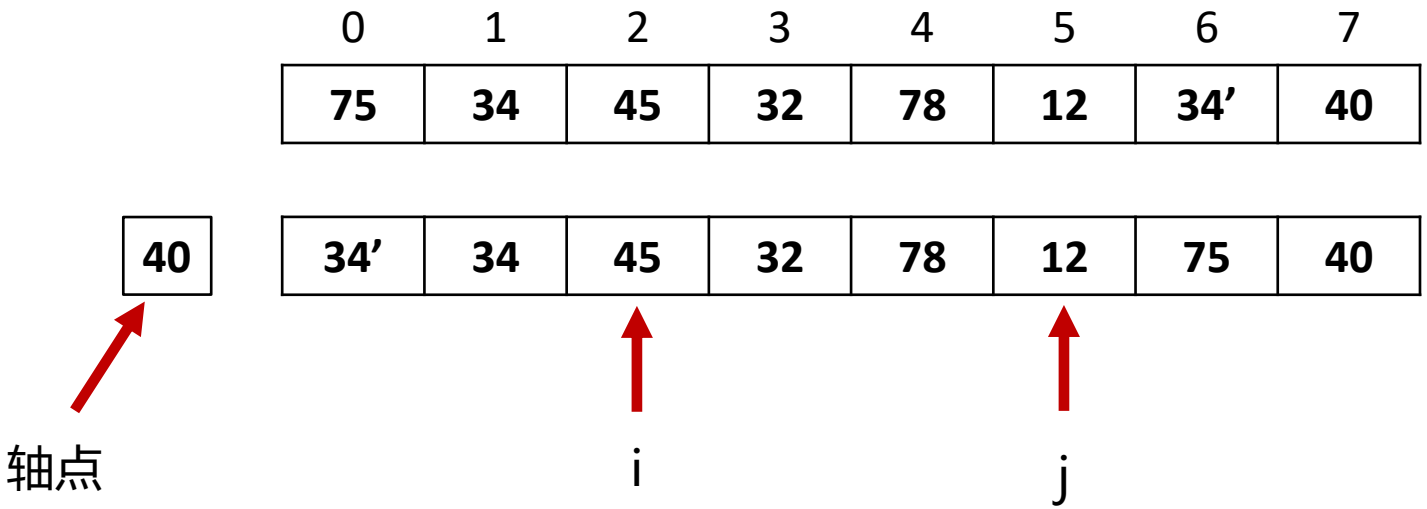
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

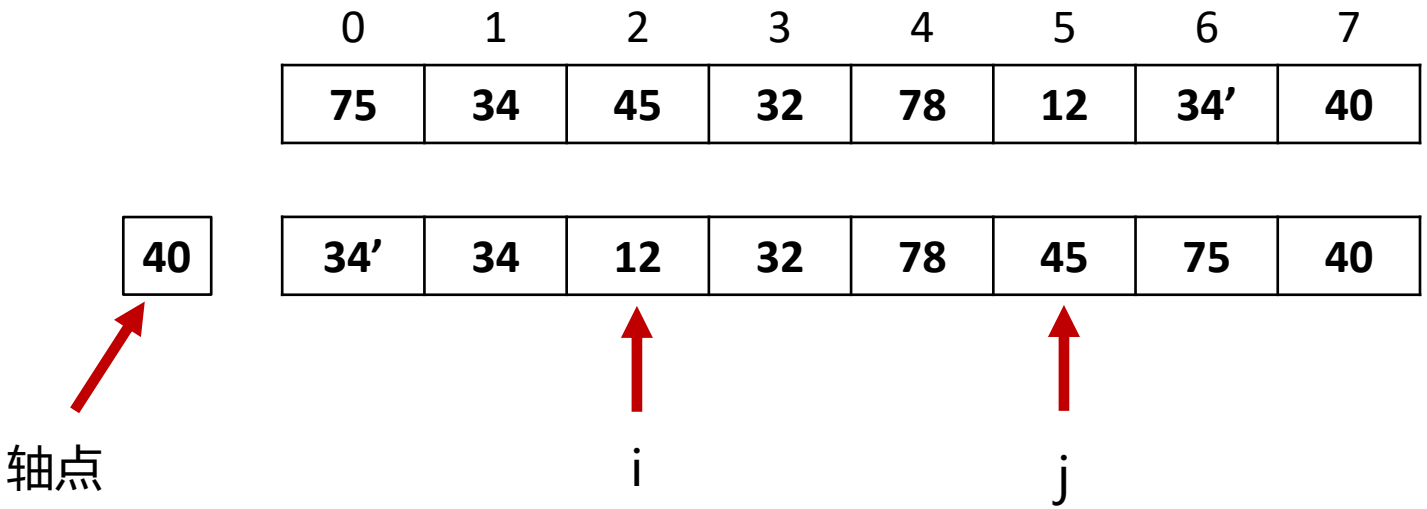
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6
6. goto 3
7. 再一次循环后：交换 a_2 与 a_5



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

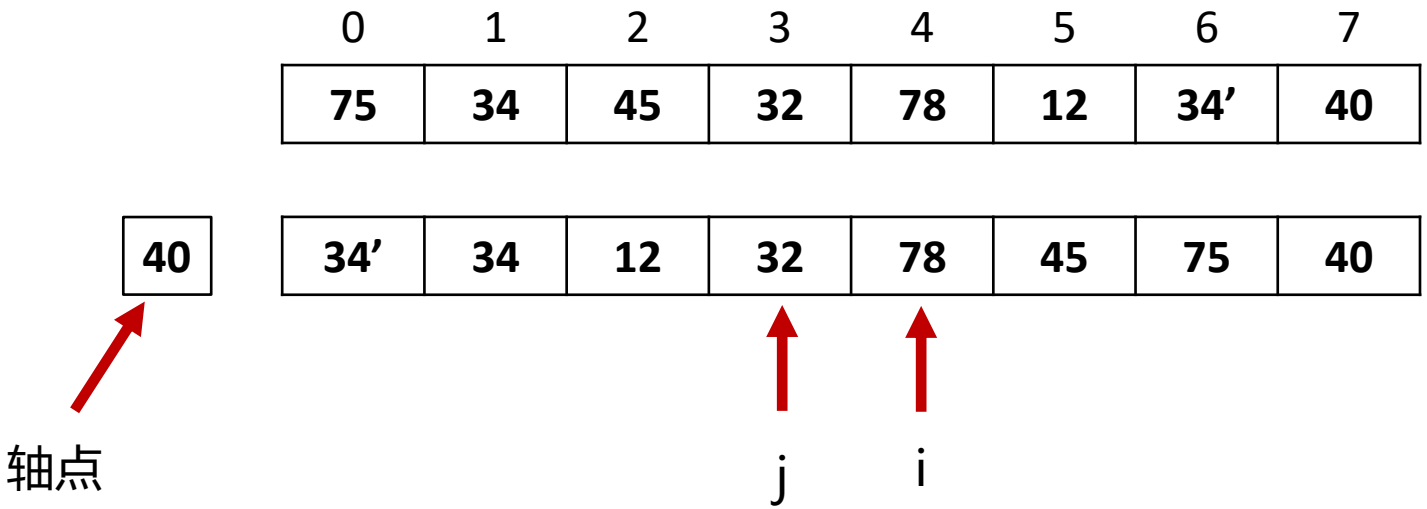
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6
6. goto 3
7. 再一次循环后：交换 a_2 与 a_5



序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

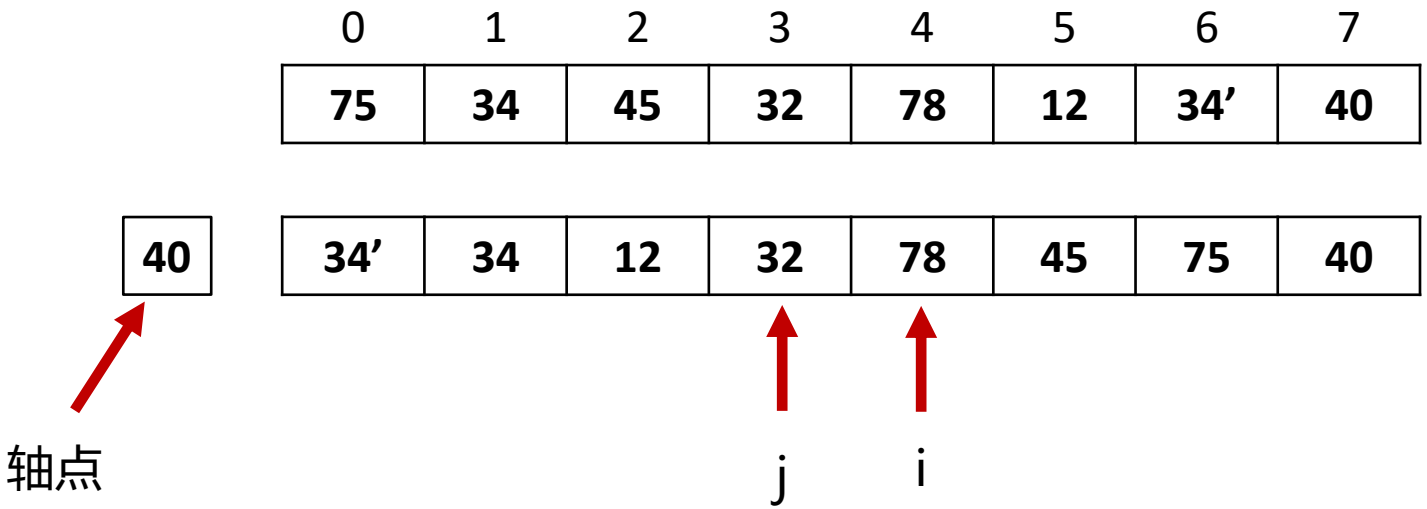
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6
6. goto 3
7. 再一次循环后：交换 a_2 与 a_5
8. 再一次循环后：$i \geq j$循环结束



序列拆分示例

对序列$a=\{75,34,45,32,78,12,34',40\}$进行一次拆分，其中$l=0,r=7$。

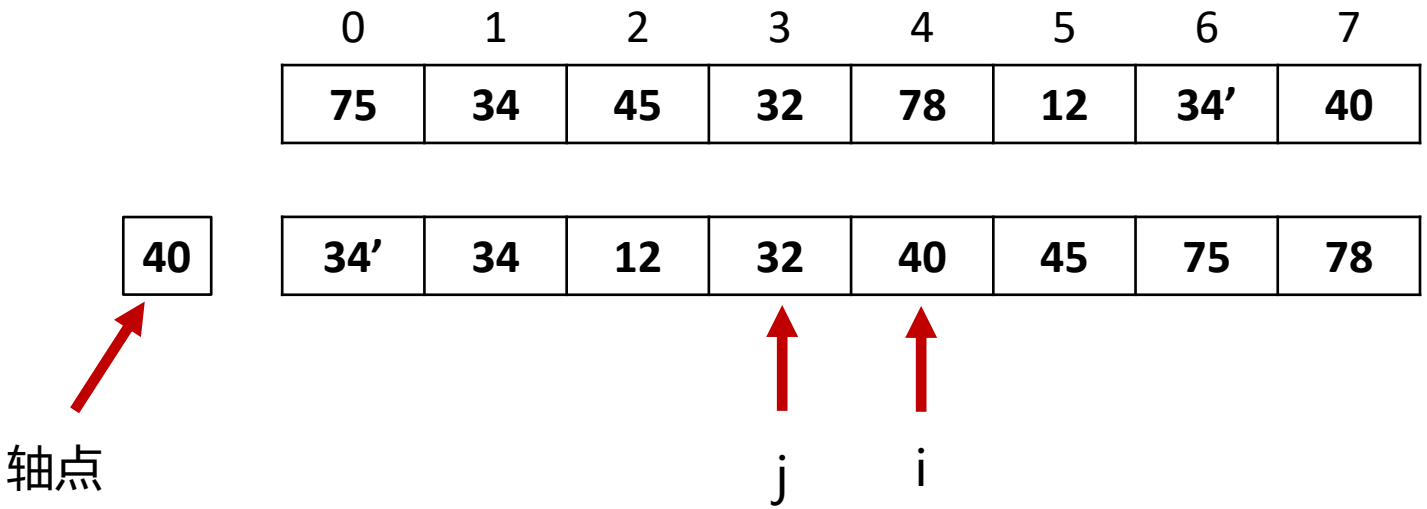
1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p=a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6
6. goto 3
7. 再一次循环后：交换 a_2 与 a_5
8. 再一次循环后：$i \geq j$循环结束
9. 交换a_i与a_r，划分结束



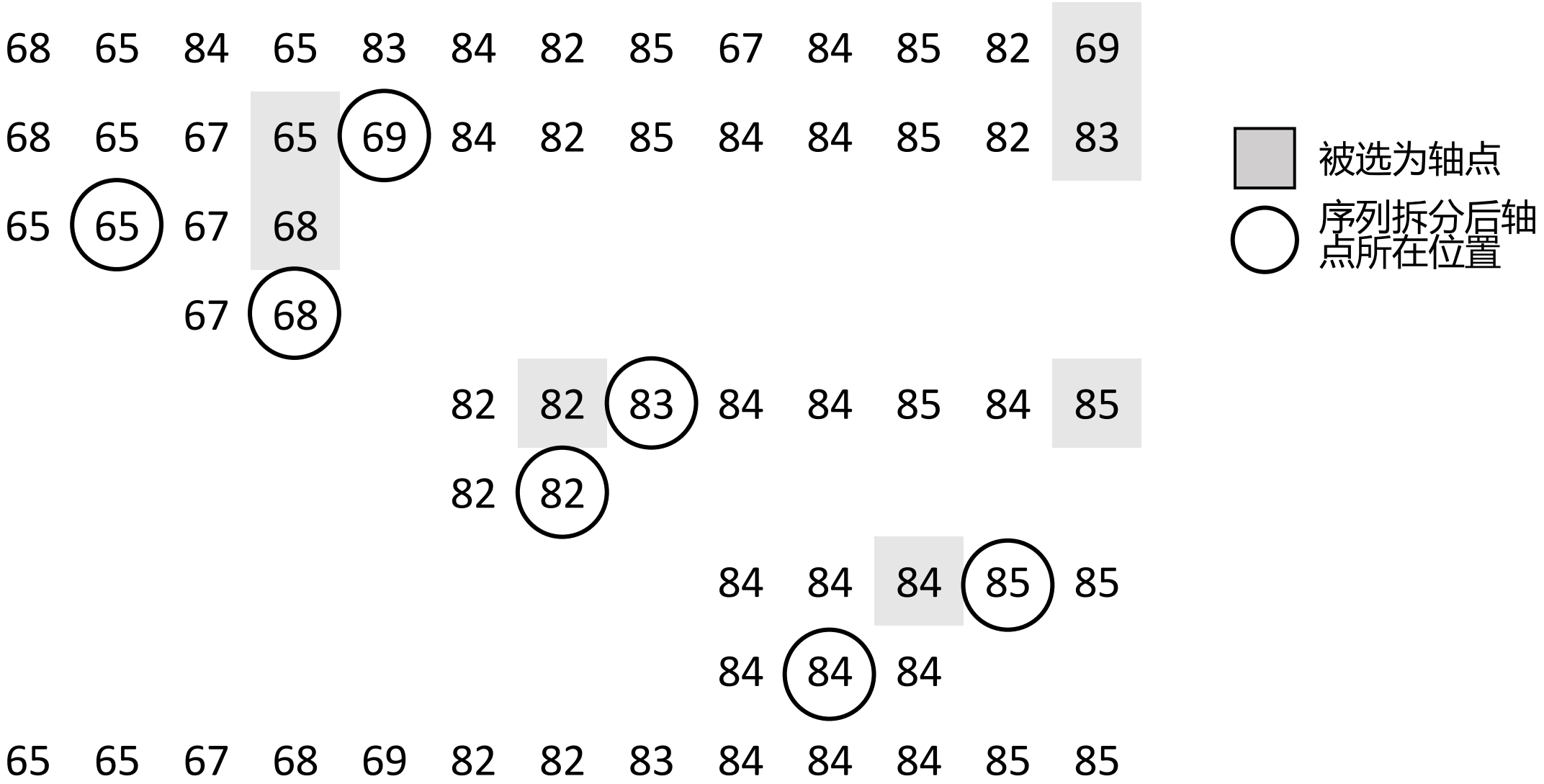
序列拆分示例

对序列$a = \{75, 34, 45, 32, 78, 12, 34', 40\}$进行一次拆分，其中$l = 0, r = 7$。

1. i指向待划分区域首元素，j指向待划分区域尾元素
2. $p = a_j$(将作为轴点的元素暂存)
3. i从前往后移动直到找到一个比轴点大的项
4. j从后往前移动直到找到一个比轴点小的项
5. 交换 a_0 与 a_6
6. goto 3
7. 再一次循环后：交换 a_2 与 a_5
8. 再一次循环后：$i \geq j$循环结束
9. 交换a_i与a_r，划分结束



快速排序示例


68 65 84 65 83 84 82 85 67 84 85 82 69
68 65 67 65 69 84 82 85 84 84 85 82 83
65 65 67 68
67 68
82 82 83 84 84 85 84 85
82 82
84 84 84 85 85
84 84 84
65 65 67 68 69 82 82 83 84 84 84 85 85
被选为轴点
序列拆分后轴点所在位置

序列拆分伪代码

算法10-5 序列拆分 Partition(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：将序列a根据轴点拆分，并输出轴点在序列中的位置

```
1   i ← l
2   j ← r − 1
3   p ← a_r //选择序列最后一个元素作为轴点
4   while true do
5   |  while a_i < p do //找到i以右第一个大于等于轴点的元素
6   |  |  i ← i + 1    //i不会大于r，因为p=a_r
7   |  end
8   |  while a_j > p and j > l do //找到j以前第一个小于等于
                                  轴点的元素
9   |  |  j ← j − 1
10  |  end
11  |  if i ≥ j then //如果i大于等于j，完成拆分，退出循环
12  |  |  break
13  |  end
14  |  Swap(a_i, a_j) //交换a_i和a_j并右移i左移j
15  |  i ← i + 1
16  |  j ← j − 1
17  end
18  Swap(a_i, a_r)
19  //此时{a_l, …, a_{i−1}} ≤ a_i ≤ {a_{i+1}, …, a_r}
20  return i
```

快速排序伪代码

算法10-5 快速排序 QuickSort(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1  if i < r then //超过1个元素才进行排序
2  | i ← Partition(a, l, r)
3  | QuickSort(a, l, i − 1)
4  | QuickSort(a, i + 1, r)
5  end
```

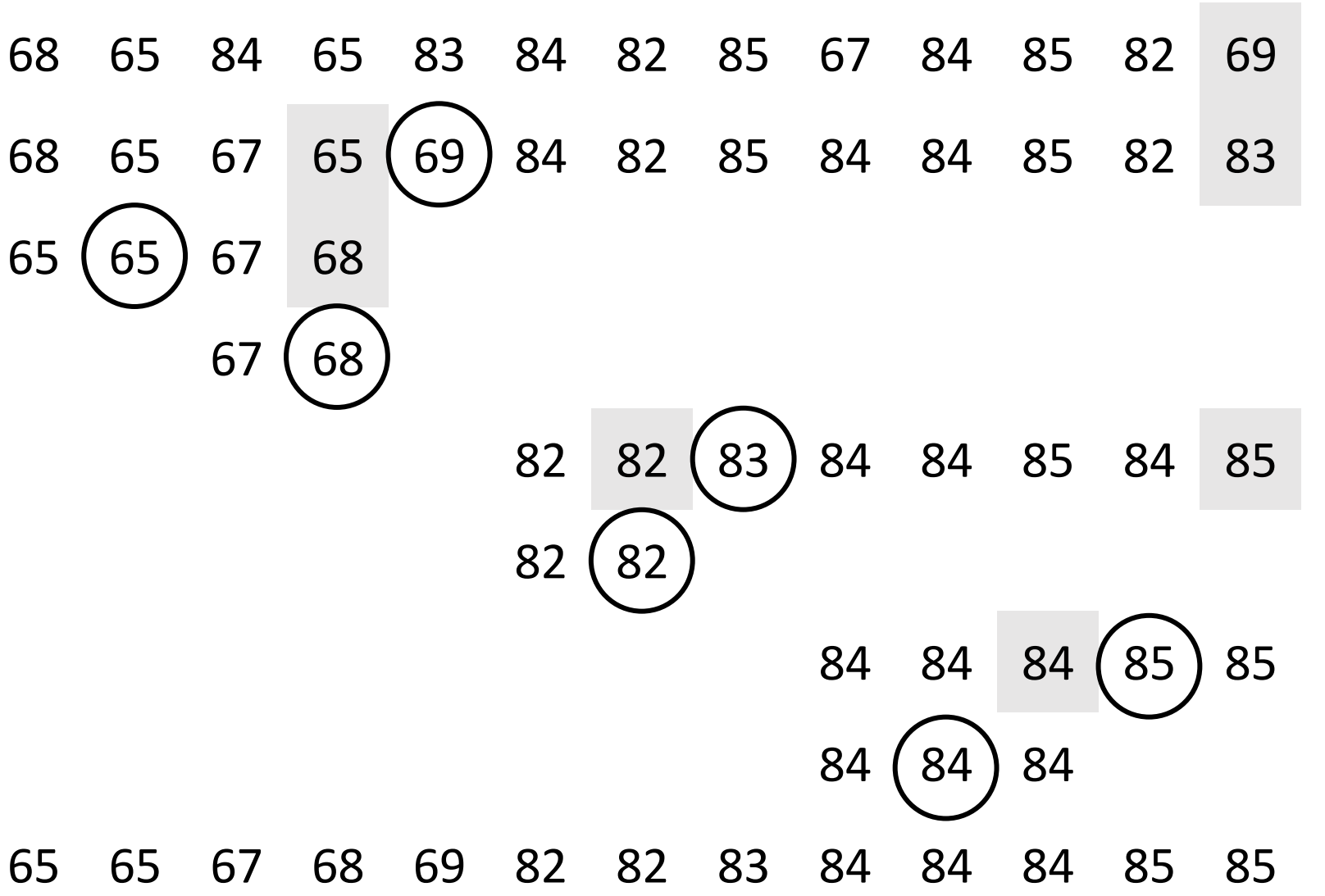
快速排序性能分析

最坏情况分析：

对于已经排序好的序列$a_l, \ldots, a_r$，第一轮选择序列最后一个元素a_r作为轴点，这时通过Partition，所有元素和轴点进行一次比较后，仍需要递归执行快速排序的子序列为 $a_l, \ldots, a_{r-1}$；第二轮选择 a_{r-1} 为轴点，并与其他元素进行比较；……；直到序列长度为1。

上述过程中，第i轮需要排序的元素个数为$\boldsymbol{r-l-i+2}$，共需要$\boldsymbol{n}$轮才能完成排序，需要的比较次数为$\boldsymbol{O(n^2)}$。

快速排序最佳时间复杂度



被选为轴点

序列拆分后轴点所在位置

每次迭代将数组平均分为两份，树高为log n，每层计算量O(n)，因此复杂度为

O(n log n)

快速排序性能分析

时间复杂度：

- 最坏时间复杂度：$O(n^2)$
- 平均时间复杂度：$O(n\log n)$

空间复杂度（递归栈深度）：

- 最坏时间复杂度：$O(n)$
- 平均时间复杂度：$O(\log n)$

快速排序是**不稳定排序**。

快速排序的轴点选择

快速排序具有优秀的平均性能，但是最坏情况下性能会退化成和冒泡排序相当。这是因为快速排序**非常依赖于轴点的选择**，如果轴点选择的不好，会导致子序列划分不均。因此，可以使用下述方法对轴点选择进行改进：

- 选取序列中$a_l, a_{\frac{l+r}{2}}, a_r$三个元素的**中位数**作为轴点
- **随机选取**一个序列内元素作为轴点
- 将上述两种选取策略结合：首先从序列内**随机**选择三个元素，并选择三个元素的**中位数**作为轴点

需要指出的是，上述轴点选择策略都只能降低快速排序退化的概率，而**不能完全避免**快速排序退化。

10.5 归并排序

核心思路：基于**分治**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。

归并次序：

- 自顶向下：将序列拆分直到有序；然后使用归并算法得到排序结果。
- 自底向上：将序列看成多条有序子序列，并将子序列两两合并。

应用场景：

- 求逆序数对：通过归并排序，快速计算序列中逆序数对的数量。

10.5.1 二路归并

二路归并算法将两个有序序列合并为一个新的有序序列。

对两有序序列，分别取出这两个序列中**键值最小**的项，并选出两个数据项中最小的放置到新的序列中；循环上述动作，直到两个序列中的所有数据**都已被放置**到新的序列中。此时，新的序列即为排序结果。

对于长度分别为n和m的有序序列，二路归并算法的主循环最多执行$n+m$次，因此其复杂度为$\boldsymbol{O(n+m)}$。

二路归并伪代码

算法10-8 二路归并

TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入：序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出：将两个有序序列合并后的新有序序列t

```
1   t ← 空序列
2   i ← l_x
3   j ← l_y
4   while i ≤ r_x or j ≤ r_y do
5   |  if j > r_y or (i ≤ r_x and a_i ≤ a_j) then
6   |  |  将a_i添加至t末尾
7   |  |  i ← i + 1
8   |  else
9   |  |  将a_j添加至t末尾
10  |  |  j ← j + 1
11  |  end
12  end
13  return t
```

10.5.2 归并排序

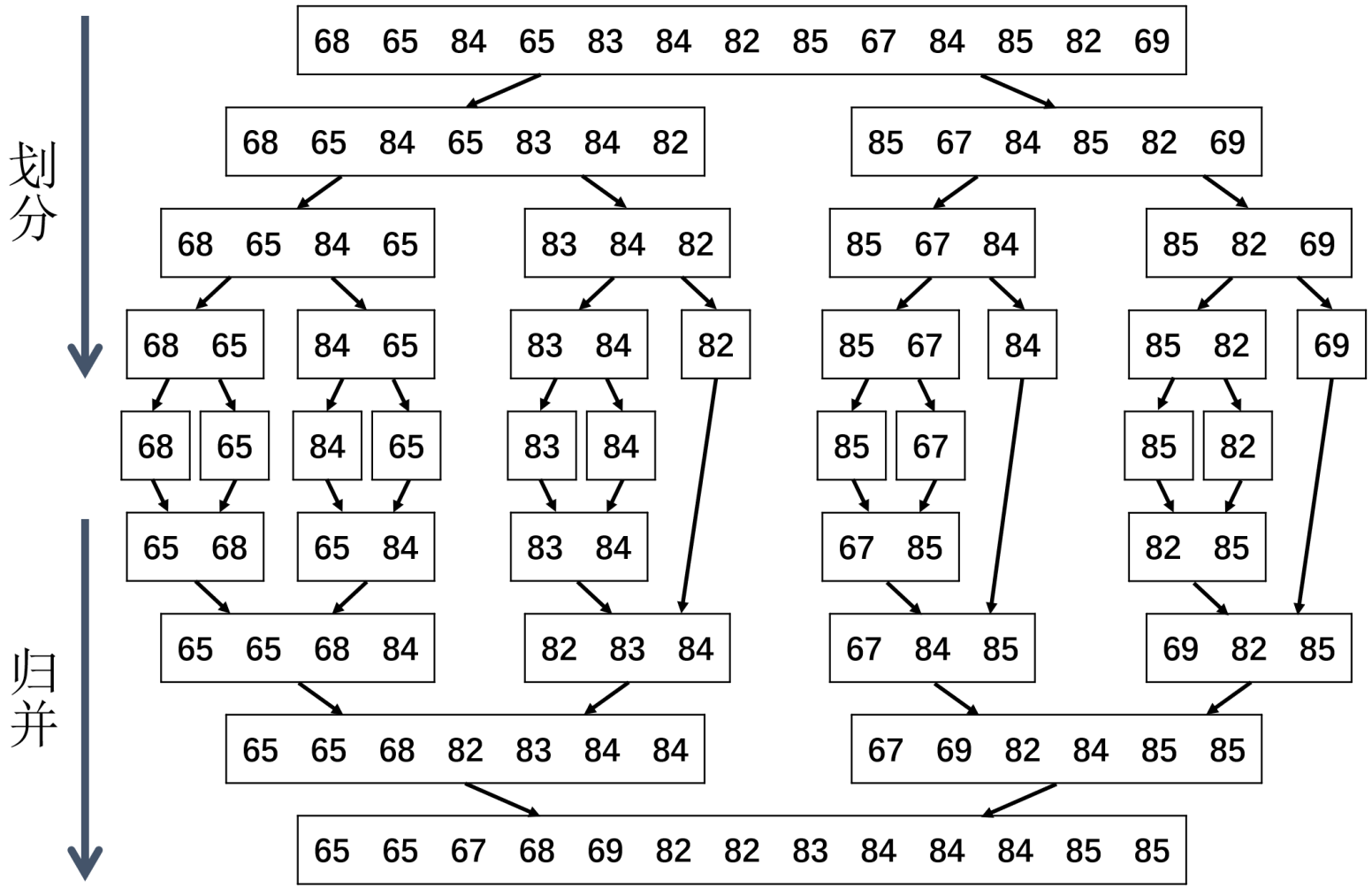
利用上述归并思想和二路归并算法来实现的排序算法称为归并排序。

算法首先将序列拆分，直到被拆分的子序列长度为1，此时子序列显然有序。

然后，使用上述二路归并算法，将有序的短序列依次合并为长序列，最终完成序列的排序。

据归并时对子序列划分方式的不同，归并排序算法又可分为**自顶向下**和**自底向上**两种。

自顶向下归并排序示例


划分
归并
68 65 84 65 83 84 82 85 67 84 85 82 69
68 65 84 65 83 84 82
85 67 84 85 82 69
68 65 84 65
83 84 82
85 67 84
85 82 69
68 65
84 65
83 84
82
85 67
84
85 82
69
68
65
84
65
83
84
85
67
85
82
65 68
65 84
83 84
67 85
82 85
65 65 68 84
82 83 84
67 84 85
69 82 85
65 65 68 82 83 84 84
67 69 82 84 85 85
65 65 67 68 69 82 82 83 84 84 84 85 85

自顶向下归并排序伪代码

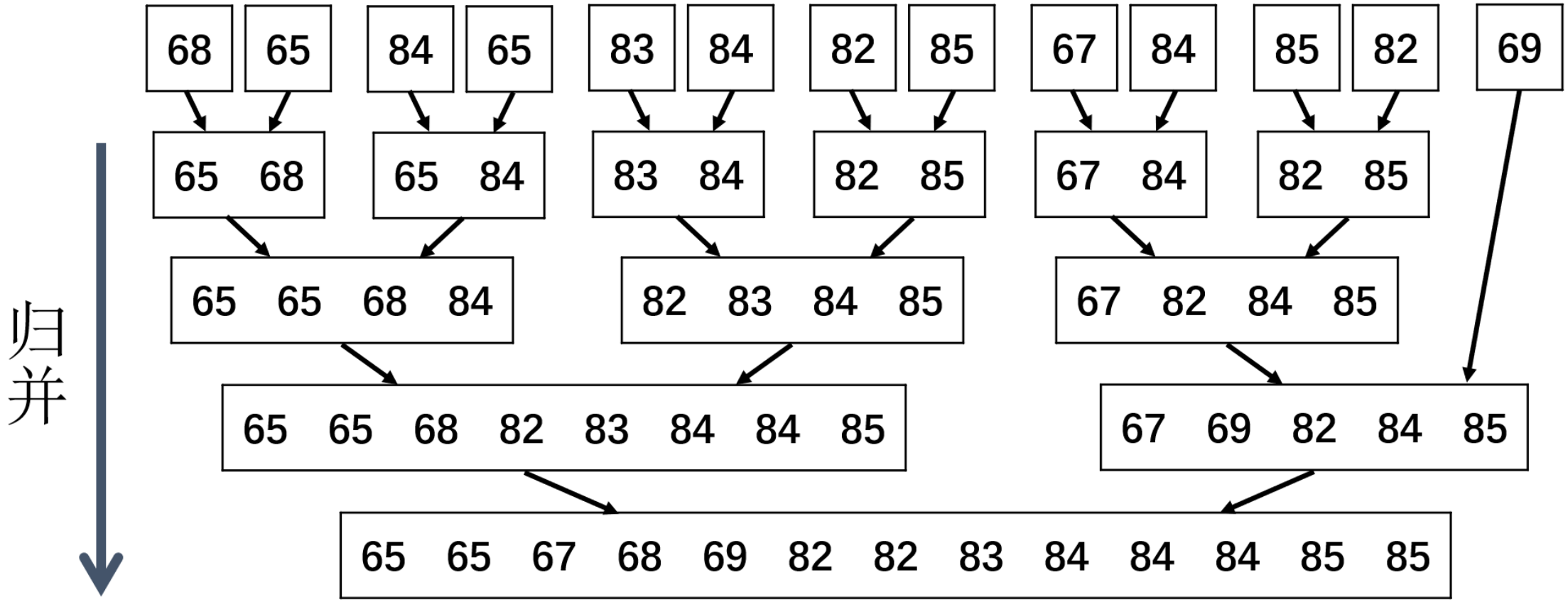
算法10-9 归并排序 MergeSort(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1  if l < r then //序列中有至少两个元素待排
2  | m ← (l + r)/2
3  | MergeSort(a, l, m)
4  | MergeSort(a, m + 1, r)
5  | ⟨a_l, ..., a_r⟩ ← TwoWayMerge(a, l, m, m + 1, r)
6  end
```

自底向上归并排序示例


68 65 84 65 83 84 82 85 67 84 85 82 69
65 68 65 84 83 84 82 85 67 84 82 85
65 65 68 84 82 83 84 85 67 82 84 85
65 65 68 82 83 84 84 85 67 69 82 84 85
65 65 67 68 69 82 82 83 84 84 84 85 85
归并

自底向上归并排序伪代码

算法10-9 自底向上归并排序 MergeSortBottomUp(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

1. *sorted_len* ← 1 //当前有序子列长度
2. n ← r-l+1 //待排元素个数，即序列长度
3. **while** *sorted_len* < n **do** //当前有序子列长度小于序列长度，则相邻两子序列归并
4. | l_x ← l //左子序列从最左端开始
5. | **while** $l_x \le r$-*sorted_len* **do**
6. | | r_x ← l_x+*sorted_len*-1 //左子序列的右端点
7. | | l_y ← r_x+1 //右子序列的左端点
8. | | r_y ← Min(l_y+*sorted_len*-1, r) //右子序列的右端点
9. | | $\langle a_{l_x}, \ldots, a_{r_y} \rangle$ ← TwoWayMerge(a, l_x, r_x, l_y, r_y) //归并
10. | | l_x ← r_y+1 //下一对子序列的左子序列的左端点
11. | **end**
12. | *sorted_len* ← *sorted_len* × 2 //有序子列长度加倍
13. **end**

归并排序性能分析

由于二路归并算法的复杂度为两个序列长度之和$O(n+m)$，因此我们分析每个元素至多会被二路归并算法调用几次。

自顶向下：

递归的深度每多一层，其待排序序列的长度就**减半**；同时，对于任意深度相同的递归调用，它们所覆盖的元素不相交。因此，递归的最大深度为$\lceil \log(n) \rceil$，且每层递归最多覆盖n个元素，其时间复杂度为$O(n \log n)$。

自底向上：

其外循环的最大循环次数为$\lceil \log(n) \rceil$，内循环将每个元素进行二路归并最多一次，因此复杂度也为$O(n \log n)$。

归并排序的改进

从二路归并的伪代码可以看到，需要使用一个临时数组来存储归并结果，并在归并排序完成后将结果拷贝回原数组，共进行$2n+2m$次拷贝。在序列元素占用内存较大时，拷贝可能花费大量时间。实际上，可以通过互换当前数组和临时数组的技巧来**减少一半**的拷贝时间。

以自底向上的归并排序为例，当外循环进行**奇数次**时，a作为有序子序列，t作为存放合并结果的临时序列；当外循环进行**偶数次**时，t作为有序子序列，a作为存放合并结果的临时序列。

改进后二路归并的伪代码

算法10-11 改进二路归并 TwoWayMergeImproved(a, l_x, r_x, l_y, r_y)

输入：序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出：将两个有序序列合并后的新有序序列t

```
1    i ← l //左子序列当前待比较的元素位置
2    j ← m + 1 //右子序列当前待比较的元素位置
3    k ← l //结果序列当前待放入的元素位置
4    while i ≤ m or j ≤ r do
5    |  if j > r or (i ≤ m and a_i ≤ a_j) then
6    |  |  t_k ← a_i
7    |  |  i ← i + 1
8    |  |  k ← k + 1
9    |  else
10   |  |  t_k ← a_j
11   |  |  j ← j + 1
12   |  |  k ← k + 1
13   |  end
14   end
```

改进后自底向上归并排序的伪代码

算法10-12 改进自底向上归并排序 MergeSortBottomUp(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1.  sorted_len ← 1 //当前有序子列长度
2.  n ← r-l+1 //待排元素个数，即序列长度
3.  count ← 0
4.  while sorted_len < n do //当前有序子列长度小于序列长度，则相邻两子序列归并
5.  | count ← count + 1
6.  | l_x ← l //左子序列从最左端开始
7.  | while l_x ≤ r-sorted_len do
8.  | | r_x ← l_x+sorted_len-1 //左子序列的右端点
9.  | | r_y ← Min(r_x+sorted_len, r) //右子序列的右端点
10. | | if count%2=1 then
11. | | | TwoWayMergeImproved(a, t, l_x, r_x, r_y) //a并入t
12. | | else
13. | | | TwoWayMergeImproved(t, a, l_x, r_x, r_y) //t并入a
14. | | end
15. | | l_x ← r_y+1 //下一对子序列的左子序列的左端点
16. | end
17. | sorted_len ← sorted_len × 2 //有序子列长度加倍
18. end
```

求逆序对数量

归并排序的一个经典应用是求逆序对数量。

在一个序列中，两个元素a_i和a_j逆序指它们满足$i < j$和$a_i > a_j$，称这两个元素为一个**逆序对**。求一个序列的逆序对数量则是找到序列中有多少组不同的i和j，满足a_i和a_j是逆序对。

显然，当序列有序时，序列的逆序对数量为零。一种最简单的方法是枚举所有可能的i和j，如果逆序则答案加一。这种算法的时间复杂度为$\boldsymbol{O(n^2)}$。

基于归并排序算法思想，可以在$\boldsymbol{O(n \log n)}$的时间里找到序列的逆序对数量。算法的核心思想在于：统计两个子序列进行二路归时序列逆序对数量**减少了多少**。

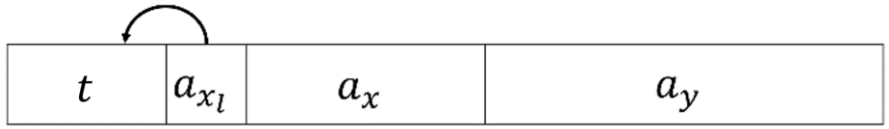
求逆序对数量

考虑二路归并时，输入有序子序列x, y的区间分别为l_x, r_x和l_y, r_y。

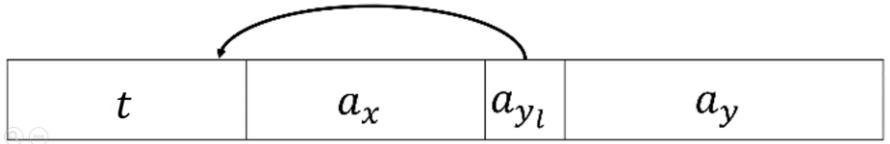
由于在归并排序中待归并的子序列满足**首尾相连**，规定$l_x \le r_x < l_y \le r_y$，就有$r_x + 1 = l_y$。

为了便于理解，可以将辅助序列t看作拼接于待排序子序列前，且每次将元素插入t末尾的操作看成将元素从原序列首移动至序列末尾，如图所示。

初始序列



归并子序列x的首元素



归并子序列y的首元素

求逆序对数量

初始时，辅助序列t中不包含任何元素。

若将x的a_{l_x}移至t末尾，相当于序列t和子序列x的分界符l_x右移，序列元素顺序**没有发生改变**。

若将y的a_{l_y}移至t末尾，相当于将该元素使用插入排序插至l_x处，此时序列中元素间两两顺序关系发生改变的只有**a_{l_y}和子序列x剩余元素**之间。

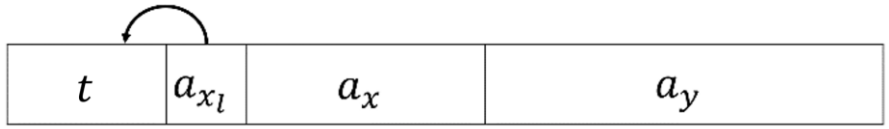
由于二路归并每次选择的元素是子序列剩余元素中**最小的**，逆序对数量减少了x目前**剩余的元素数量**。

最后，二路归并这两个子序列时，如果一个逆序对的两个元素未被两个子序列完全包含，则二路归并后它们**仍然为逆序对**。

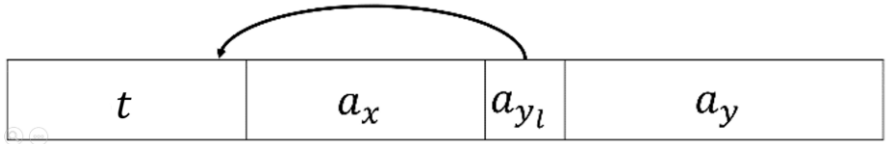
因此，通过二路归并，将两个子序列合并为有序序列后，求出了逆序对**减少的数量**，同时**没有影响**其它未被两个子序列完全包含的逆序对数量。

算法时间复杂度和归并排序相同，为$O(n\log n)$。

初始序列



归并子序列x的首元素



归并子序列y的首元素

二路归并求逆序对减量的伪代码

算法10-13 二路归并求逆序对减量

TwoWayInversionCount(a, l, m, r)

输入：序列a，相邻两个有序子序列范围l, m和$m+1, r$

输出：将两个有序序列合并，并返回减少的逆序对数量

```
1   t ←空序列
2   i ← l
3   j ← m + 1
4   count← 0
5   while i ≤ m or j ≤ r do
6   |  if j > r or (i ≤ m and a_i ≤ a_j) then
7   |  |  将a_i添加至t末尾
8   |  |  i ← i + 1
9   |  else
10  |  |  将a_j添加至t末尾
11  |  |  j ← j + 1
12  |  |  count ← count + (m − i + 1)
13  |  end
14  end
15  return count
```

归并排序兼求逆序对数量的伪代码

算法10-14 归并排序兼求逆序对数量 $\text{InversionCount}(a, l, r)$

输入： 序列a，左端点下标l，右端点下标r

输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列，同时返回序列中逆序对数量

```
1  count ←0
2  if l < r then //序列中至少有2个元素时才执行
3  | m ← (l + r)/2
4  | count ← count + InversionCount(a, l, m)
5  | count ← count + InversionCount(a, m + 1, r)
6  | count ← count + TwoWayInversionCount(a, l, m , r)
7  end
8  return count
```

10.6 基于比较排序的复杂度分析

核心问题：是否存在时间复杂度优于 $O(n\log n)$ 的基于比较排序的算法?

- 基于比较的排序的复杂度下界
- 基于比较多排序的平均复杂度
- 最少比较排序

10.6.1 基于比较的排序的复杂度下界

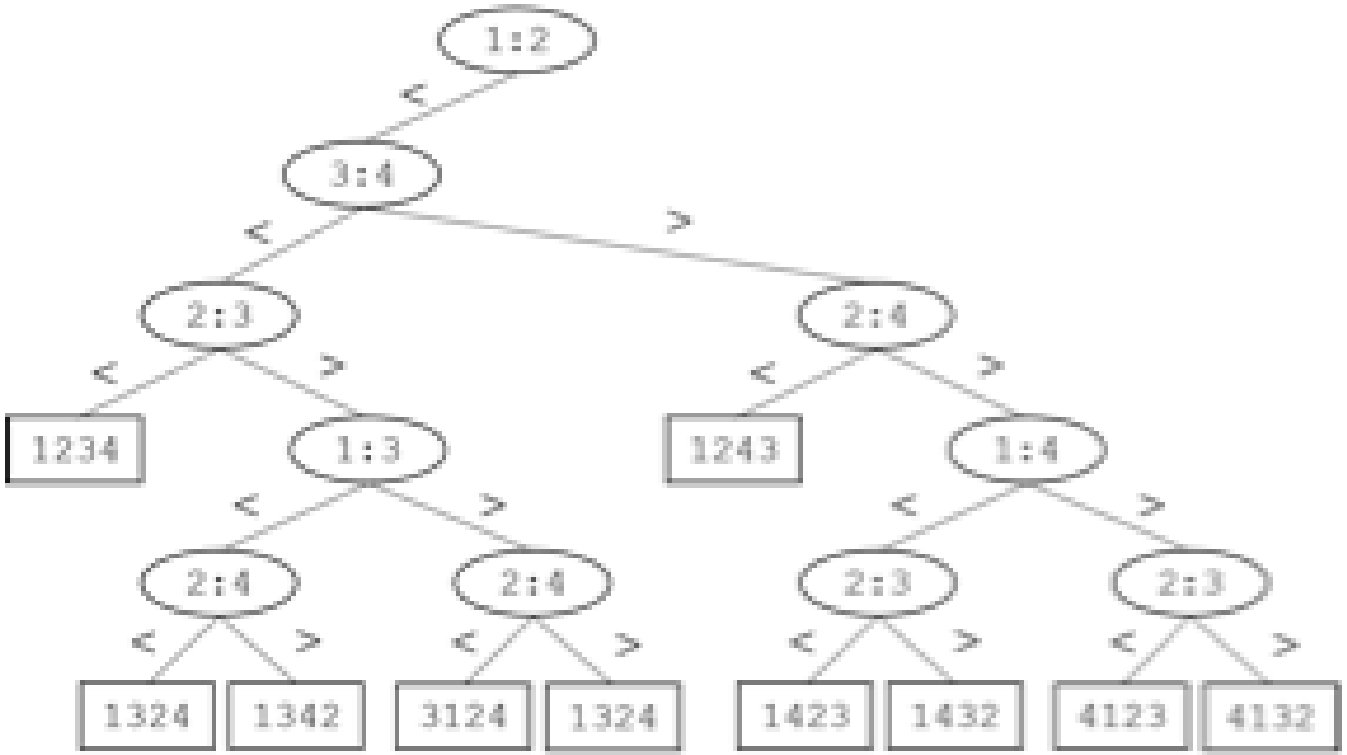
核心问题：每次操作可以询问两个元素 a_i 是否大于 a_j，对于任意的输入数组，在最坏情况下，至少需要进行多少次操作才能确定输入数组中所有元素的次序。

理论分析：对于每次询问操作，都可以将剩下可能的次序分成“满足条件”和“不满足条件”两类，在最坏情况下，我们每次最多只能排除一半的可能次序。对于 $n!$ 种可能次序，至少需要进行 $O(\log(n!))$ 次操作。

$$\log(n!) = \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \frac{1}{12n} + \frac{1}{288n^2} + \cdots\right)\right) \sim O(n \log n)$$

10.6.1 基于比较的排序的复杂度下界

理论分析（二叉决策树）：（以四个元素为例）非叶节点 $i:j$ 表示比较 a_i 和 a_j，如果小于则走向左子树，否则走向右子树。叶子结点包含了所有的可能次序。



10.6.1 基于比较的排序的复杂度下界

理论分析（二叉决策树）： 设所有非叶节点的深度均小于 d，则叶子结点的数量最多为 2^d（满二叉树时取到）。

$$n! \leq 2^d \Rightarrow$$

$$d \geq \lceil \log(n!) \rceil \sim O(n \log n)$$

10.6.3 最少比较排序

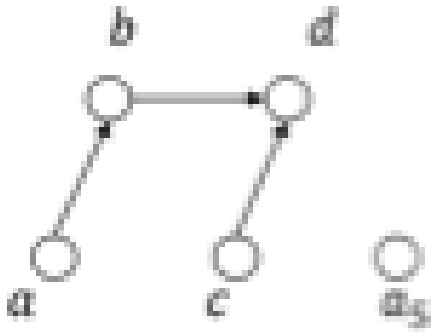
核心问题：我们通过理论分析得到了比较排序比较次数的下界，**但是否能找到满足下界的排序算法仍是未知之数**。对于长度 n 的序列，找到一个在最坏情况下比较次数最少的排序算法。记理论下界为 $C(n)$，已知的最好比较排序的比较次数为 $S(n)$[1]。

	$C(n)$	$S(n)$		$C(n)$	$S(n)$		$C(n)$	$S(n)$
$n=3$	3	3	$n=8$	16	16	$n=13$	33	34
$n=4$	5	5	$n=9$	19	19	$n=14$	37	38
$n=5$	7	7	$n=10$	22	22	$n=15$	41	42
$n=6$	10	10	$n=11$	26	26	$n=16$	45	46
$n=7$	13	13	$n=12$	29	30	$n=17$	49	50

[1] Stober F, Weiß A. Lower Bounds for Sorting 16, 17, and 18 Elements[C]//2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX). Society for Industrial and Applied Mathematics, 2023: 201-213.

10.6.3 最少比较排序

示例（$n=5$）：7 次比较。先比较 a_1、a_2 和 a_3、a_4，再将两者的较大者进行比较，得到如下图所示次序图，其中 $a \to b$ 表示 $a < b$。再通过最多两次比较将 a_5 插入到 a、b、d 中的适当位置。继续通过最多两次比较将 c 插入到 a、b、d、a_5 中的适当位置。一共进行最多七次比较完成排序。



思考：当 $n=6$ 时，如何设计一个最小比较排序算法[1]。

[1] Ford L R , Johnson S M .A Tournament Problem[J].American Mathematical Monthly, 1959, 66(5):387-389.DOI:10.2307/2308750.

10.7 基于分配的排序

定义：不基于“比较-移动”的排序方式，常见的基于分配的排序有

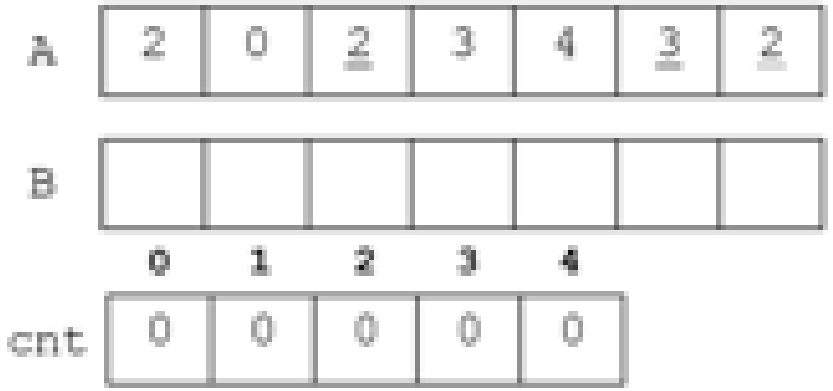
- 计数排序（Counting Sort）
- 桶排序（Bucket Sort）
- 基数排序（Radix Sort）

10.7.1 计数排序

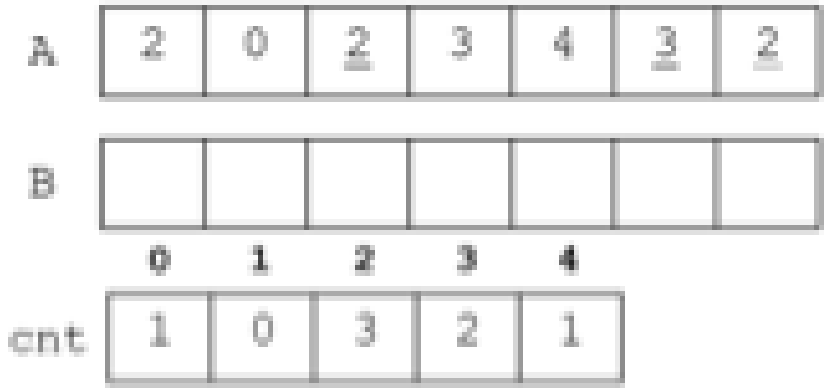
假设：假设 n 个输入元素的每一个都是 0 到 k 的一个整数。

核心思路：对于每一个输入元素 x，确定小于 x 的元素个数，这样就可以将 x 放在它在输出数组的位置上。当有多个 x 相同时，我们统计相同 x 的数量，再依次存放即可。

示例：待排序数组 (2, 0, 2, 3, 4, 3, 2)。



初始状态



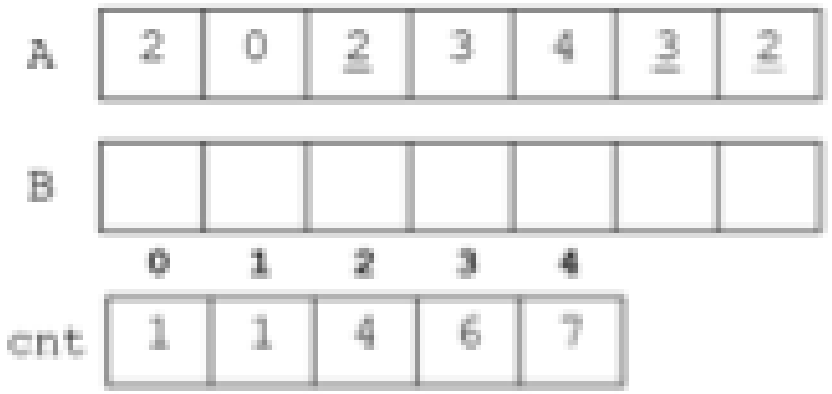
统计每个值出现的次数

10.7.1 计数排序

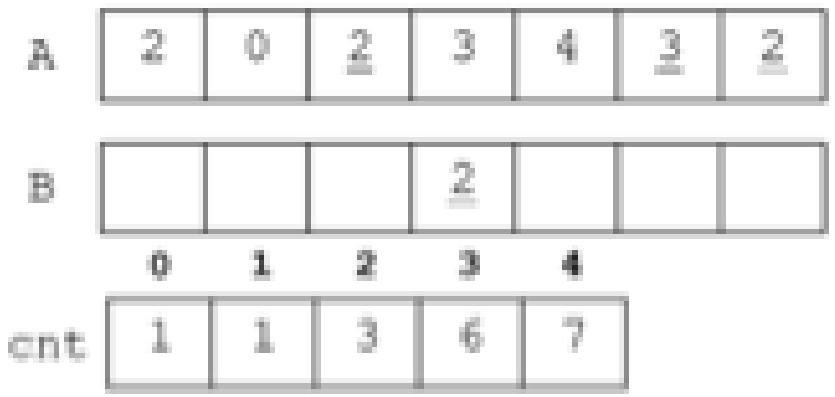
假设：假设 n 个输入元素的每一个都是 0 到 k 的一个整数。

核心思路：对于每一个输入元素 x，确定小于 x 的元素个数，这样就可以将 x 放在它在输出数组的位置上。当有多个 x 相同时，我们统计相同 x 的数量，再依次存放即可。

示例：待排序数组 (2, 0, 2, 3, 4, 3, 2)。



统计 cnt 前缀和



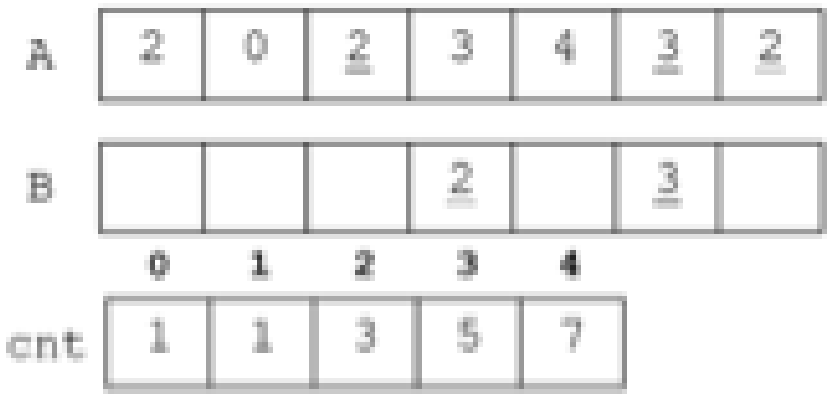
从后向前依次将 A 的每个元素 x 放入 B 中，具体位置由 cnt[x] 确定

10.7.1 计数排序

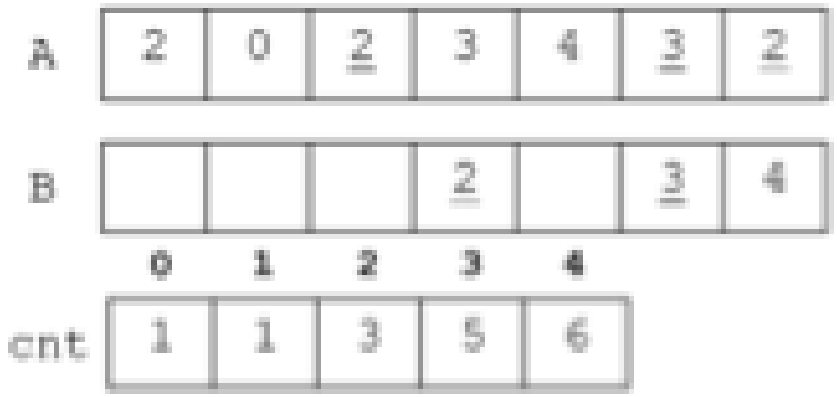
假设：假设 n 个输入元素的每一个都是 0 到 k 的一个整数。

核心思路：对于每一个输入元素 x，确定小于 x 的元素个数，这样就可以将 x 放在它在输出数组的位置上。当有多个 x 相同时，我们统计相同 x 的数量，再依次存放即可。

示例：待排序数组 (2, 0, 2, 3, 4, 3, 2)。



插入 3



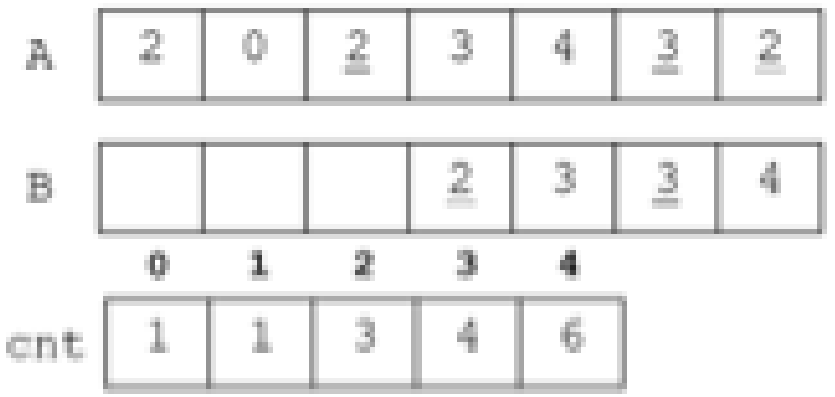
插入 4

10.7.1 计数排序

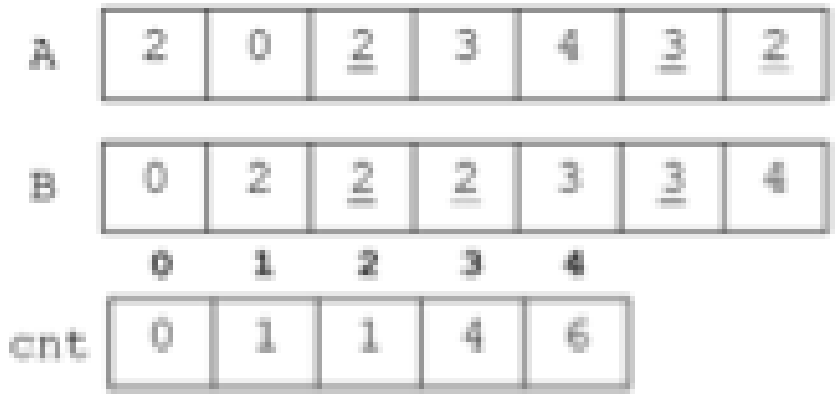
假设：假设 n 个输入元素的每一个都是 0 到 k 的一个整数。

核心思路：对于每一个输入元素 x，确定小于 x 的元素个数，这样就可以将 x 放在它在输出数组的位置上。当有多个 x 相同时，我们统计相同 x 的数量，再依次存放即可。

示例：待排序数组 (2, 0, 2, 3, 4, 3, 2)。



插入 3



插入剩余的元素

计数排序伪代码

算法 10-15 计数排序 CountingSort(a, l, r, k)

输入： 序列 a，左端点下标 l，右端点下标 r，元素最大值 k

输出： 调整 $a_l, a_{l+1}, \ldots, a_r$ 元素顺序，使元素按照非递减顺序排列

```
1.  b ← new ElemSet [r-l+1] //临时存放有序序列的数组
2.  cnt ← new int [k+1] () //计数数组，初始全零
3.  for i←l to r do
4.  |  cnt[a_i] ← cnt[a_i] + 1
5.  end
6.  for i←1 to k do
7.  |  cnt[i] ← cnt[i-1] + cnt[i]
8.  end
9.  for i← r downto l do
10. |  p ← cnt[a_i]-1 //a_i 应该在b中的位置
11. |  b_p ← a_i //将a_i放入
12. |  cnt[a_i] ← cnt[a_i] - 1
13. end
14. for i←l to r do //将有序的b放回a中，b从0开始，a从l开始
15. |  a_i ← b_{i-l+1}
16. end
```

计数排序性能分析

时间复杂度：在上述伪代码中，第 2-8 行统计小于等于每个元素的数的数量，时间复杂度为 $O(n+k)$；第 9-13 行根据 cnt 将元素放至正确位置，时间复杂度为 $O(n)$；第 14-16 行将完成排序的元素放回原序列，时间复杂度为 $O(n)$。总体时间复杂度为 $O(n+k)$。当 k 为 n 的常数 c 倍，即 $k=cn$ 时，计数排序的时间复杂度为 $O(n)$。

空间复杂度：排序过程中需要两个额外数组 b 和 cnt。故空间复杂度为 $O(n+k)$。

计数排序是稳定的。

“特殊的”计数排序——桶排序

桶排序：

1. 通过一个单调映射函数 f 将 n 个元素分配到 k 个桶中
2. 然后对每个桶中的元素进行排序（一般使用插入排序）
3. 最后从这 k 个桶中依次取出就得到有序的序列。

与计数排序的关系：计数排序可以看成映射函数为 $f(x)=x$，桶个数为元素最大值的桶排序。此时，由于同一桶内的元素大小均相等，所以不需要再对桶内元素排序，而改为计数即可。

桶排序性能分析

时间复杂度：桶排序的效率主要取决于两个因素：映射函数和桶的数量。如果映射函数不能把元素较均匀地分配到各个桶，或者桶的数量很少，就会存在大量桶内排序的代价。假设映射函数把元素均匀地分配到 k 个桶，此时桶排序的平均时间复杂度为

$$O\left(n+\frac{n^2}{k}+k\right)$$

注意，复杂度和桶内排序方法有关！可以自由选择桶内排序方法。

空间复杂度：桶排序的空间复杂度为 $O(n+k)$。

桶排序是稳定的。

10.7.2 基数排序

核心思路：将待排序元素转换成基于基数的元组表示，再对这些元组进行排序。对于十进制整数，以 10 为基数，则每个数的元组表示为由各个数位依次组成的列表，即

$$a_i = v_i^1 10^{d-1} + v_i^2 10^{d-2} + \cdots + v_i^d 10^0 = (v_i^1, \ldots, v_i^{d-1}, v_i^d) = v_i$$

其中 $0 \le v_i^j \le 9$。v_i^1 为元组关键字的最高位，v_i^d 为元组关键字的最低位。使用**计数排序**来对元组的每一位进行排序。

基数排序分为两种：

- 最高位优先基数排序 MSD(Most Significant Digit First)
- 最低位优先基数排序 LSD(Least Significant Digit First)

基数排序中的计数排序伪代码

算法 10-16 基数排序中使用的计数排序CountingSort2(a, l,r,$radix$, k)

输入：序列 a，左端点下标 l，右端点下标 r，基数 $radix$，计数位 k

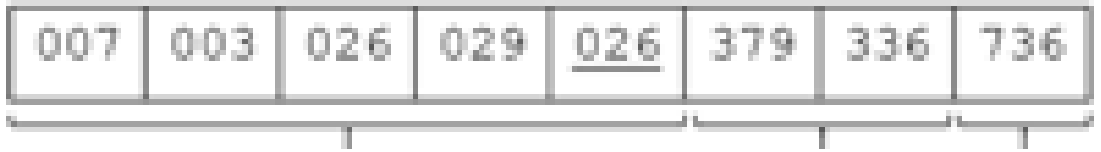
输出：调整 $a_l, a_{l+1}, \ldots, a_r$ 元素顺序，使元素按照计数位 k 非递减顺序排列，并返回统计小于每个基数元素个数的 cnt 数组

```
1.  b ← new ElemSet [r-l+1] //临时存放有序序列的数组
2.  c ← new DigitSet [r-l+1] //存储元素第k位的数组
3.  cnt ← new int [radix] () //计数数组，初始全零
4.  for i←l to r do
5.  |  c_i ← GetDigit( a_i, radix, k) //得到a_i在基数radix下元组表示的第k位
6.  |  cnt[c_i] ← cnt[c_i] + 1
7.  end
8.  for i←1 to radix-1 do
9.  |  cnt[i] ← cnt[i-1] + cnt[i]
10. end
11. for i← r downto l do
12. |  p ← cnt[c_i] //a_i 应该在b中的位置
13. |  b_p ← a_i //将a_i放入
14. |  cnt[c_i] ← cnt[c_i] - 1
15. end
16. for i←l to r do //将有序的b放回a中
17. |  a_i ← b_{i-l+1}
18. end
19. return cnt
```

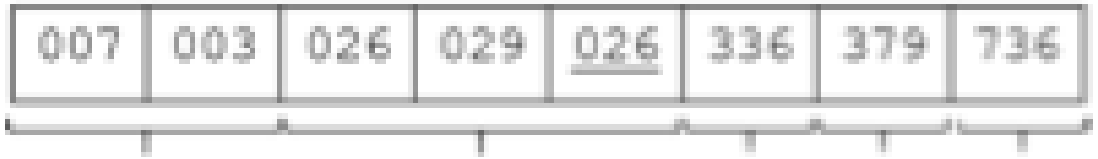
最高位优先基数排序

核心思路：首先按关键字最高位 v_i^1 使用计数排序，可得到若干个最高位值都相同的序列；接着对每个序列分别按关键字 v_i^2 使用计数排序，再将其分成若干个子序列，此时每个子序列的最高位和次高位的值都相同。以此类推，第 p 轮将对 v_i^p 使用计数排序，直到每个子序列中都仅含一个元素，或者所有位都经过排序。

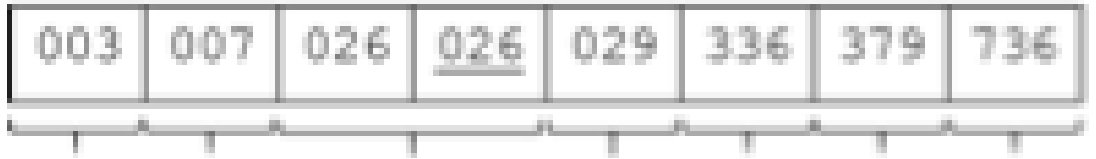
初始状态



以百位划分子序列



以十位划分子序列



以个位划分子序列

最高位优先基数排序伪代码

算法 10-17 MSD 基数排序 MSDRadixSort$(a, l, r, radix, k, d)$

输入： 序列 a，左端点下标 l，右端点下标 r，基数 $radix$，计数位 k，元组长度 d

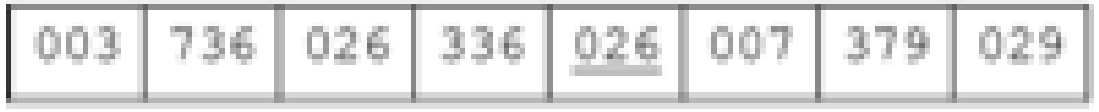
输出： 调整 $a_l, a_{l+1}, \ldots, a_r$ 元素顺序，使元素按照计数位 k 非递减顺序排列

```
1  if l + 1 ≥ r or k > d then //子序列长度不足1，或计数位超过元组长度
2  |  return
3  end
4  cnt ← CountingSort2(a, l, r, radix, k)
5  for i ← 0 to radix − 2 do
6  |  MSDRadixSort(a, l + cnt_i, l + cnt_{i+1} − 1, radix, k + 1, d)
7  end
```

最低位优先基数排序

核心思路：首先按关键字最低位 v_i^d 的值的大小将元组序列分成若干个子序列，再按 v_i^d 的值，从小到大依次将各个子序列收集起来，产生一个新序列；再对新的元组序列按 v_i^{d-1} 的值的大小分成若干个子序列，再按 v_i^{d-1} 值从小到大依次将各个子序列收集起来，又产生了一个新的序列；以此类推，再按 $v_i^{d-2}, \ldots, v_i^1$ 的值依次重复上述过程。

初始状态



按个位进行排序

按十位进行排序

按百位进行排序

最低位优先基数排序伪代码

算法 10-18 LSD 基数排序 $\mathrm{LSDRadixSort}(a, l, r, radix, d)$

输入：序列 a，左端点下标 l，右端点下标 r，基数 $radix$，元组长度 d

输出：调整 $a_l, a_{l+1}, \ldots, a_r$ 元素顺序，使元素按照计数位 k 非递减顺序排列

```
1  for i ← d downto 1 do
2  |  CountingSort2(a, l, r, radix, i)
3  end
```

利用计数排序的稳定性，低位排好序后，高位排序不会改变低位的相对次序

基数排序性能分析

时间复杂度：取决于算法CountingSort2的调用次数。LSD基数排序调用了d次，因此时间复杂度为 $O\big(d(n+radix)\big)$。MSD基数排序调用该函数次数和划分出的子序列数量直接相关，最坏情况下划分的子序列个数远远多余d，其时间复杂度比LSD高。

空间复杂度：两种基数排序的空间复杂度均为 $O(n+radix)$ 。

10.8 索引排序

在之前提到的排序算法中，不论是基于比较的排序还是基于分配的排序，都需要拷贝或移动序列中的元素。当元素移动和拷贝的代价很大时，需要尽可能减少元素移动和拷贝的次数。

核心思路：创建一个**索引序列idx**，在排序时使用原序列元素进行比较，使用索引序列进行元素交换。最后，算法会给出**排序后的索引序列**。

基于插入排序的索引排序的伪代码

算法10-19：基于插入排序的索引排序 IndexedInsertionSort(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：返回一个索引序列idx，$idx_i = p$指a_p是原序列中第i大的元素

```
1.  for i ← l+1 to r do
2.  | t ← idx_i
3.  | for j←i-1 downto l do
4.  | | if a_idxj>a_t then
5.  | | | idx_j+1 ← idx_j
6.  | | else
7.  | | | idx_j+1 ← t
8.  | | | break
9.  | | end
10. | end
11. | if a_idxl>a_t then
12. | | idx_l ← t
13. | end
14. end
15. return idx
```

元素顺序调整

在得到排序的索引序列以后，需要基于索引序列调整原序列中元素顺序使其有序。

一种简单的方法是新建一个临时序列tmp，令$tmp_i = a_{idx_i}$，然后将tmp拷贝回a。但是这种方法会使用较多的存储空间。实际上，按照下述方法，可以仅使用$\boldsymbol{O(1)}$的空间和$\boldsymbol{O(n)}$的时间，在拥有排序的索引序列后调整原序列的元素顺序使其有序。

- 检查第i个位置的元素位置是否正确
- 如果位置不正确，将a_i**暂存**，并将应放的元素$\boldsymbol{a_{idx_i}}$移至该位置，并记该元素当前所在位置$\boldsymbol{idx_i}$**为**$\boldsymbol{j}$。
- 继续检查位置j是否正确，重复上述操作直到**应放元素为**$\boldsymbol{a_i}$为止。
- 此时所有在循环中访问到的元素已归位，继续检查第$\boldsymbol{i+1}$个位置的元素位置是否正确，直到所有元素归位。

元素顺序调整的伪代码

算法10-19：基于插入排序的索引排序 IndexedInsertionSort(a, l, r)

输入：序列a，左端点下标l，右端点下标r

输出：返回一个索引序列idx，$idx_i = p$指a_p是原序列中第i大的元素

```
1.  for i ← l+1 to r do
2.  | t ← idx_i
3.  | for j←i-1 downto l do
4.  | | if a_idxj>a_t then
5.  | | | idx_j+1 ← idx_j
6.  | | else
7.  | | | idx_j+1 ← t
8.  | | | break
9.  | | end
10. | end
11. | if a_idxl>a_t then
12. | | idx_l ← t
13. | end
14. end
15. return idx
```

10.9 拓展延伸

前面几节介绍了经典排序算法的原理和特点。本节从库函数实现的角度，介绍两个拓展算法：

- **内省排序**：C++语言中最常用的排序函数实现时采用的排序算法，它是快速排序的一种拓展，是以快速排序为主干，混合堆排序和插入排序的一种混合排序算法；

- **蒂姆排序**：Java和Python语言中最常用的排序函数实现时采用的排序算法，它是归并排序的一种拓展，主要思想是对序列中的有序（顺序或逆序）子序列进行利用以提高效率，同时混合了插入排序作为优化。

10.9.1 内省排序

在所有基于比较的排序中，**快速排序**是平均运行速度最快的排序方法。然而，快速排序有其劣势，在某些情况下其表现不如其他排序算法。最坏时间复杂度会退化至$O(n^2)$，这在一些情况下是不可接受的。例如在网络服务中，攻击者可以构造特殊的序列，使服务器需要花费普通序列更多的时间用于排序，从而使攻击者容易对服务器进行拒绝服务（DoS）攻击。为了提升排序算法的效率和稳定性，在快速排序的基础上，David Musser在1997年提出了内省排序算法。

内省排序算法是一种**混合排序算法**，按照如下步骤避免快速排序退化，提升快速排序的效率：

- 算法算法会首先尝试使用**快速排序**对序列进行排序。
- 当递归层数超过预先设定的深度阈值D后转为**堆排序**。
- 如果剩余元素个数少于阈值kThrLen，则会采用**插入排序**。

在目前的主流实现中，D取$1.5\log n \sim 2\log n$，kThrLen取6~32之间。

内省排序的伪代码

算法10-21：索引排序 IntroSort(a, l, r)

输入：序列a，左端点下标l，右端点下标r，深度阈值d

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1   m← kThrLen //设定短子序列的阈值
2   if r − l < m then
3   |  InsertSort(a, l, r)
4   else if d = 0 then
5   |  HeapSort(a, l, r)
6   else
7   |  i ← Partition(a, l, r)
8   |  IntroSort(a, l, i − 1, d − 1)
9   |  IntroSort(a, i + 1, r，  d − 1)
10  end
```

10.9.2 蒂姆排序

蒂姆排序：蒂姆排序是一种混合、稳定的自适应排序算法，其结合了归并排序和插入排序的思想，旨在对真实情况下的数据有更好的排序效率。

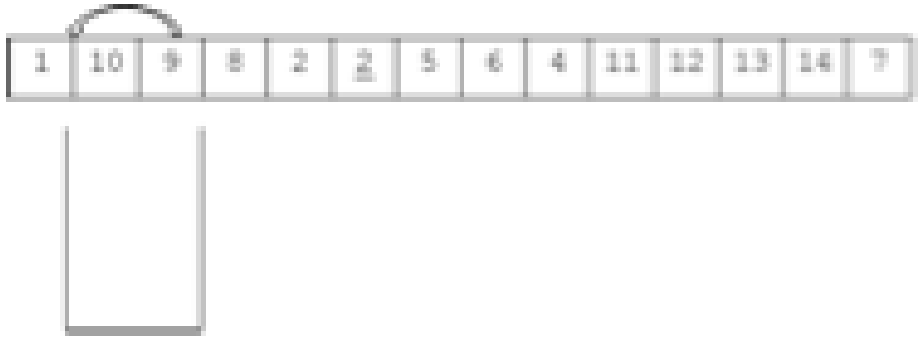
蒂姆排序由 Tim Peters 在 2002 年提出，并自 Python2.3 版本以来，一直作为其标准排序算法，并且在 Java SE 7、Swift 和 Rust 等语言中，使用它对非原始类型的数组进行排序。蒂姆排序的基本思想是：先找到输入数据中的有序连续子序列（run），然后通过某种方式归并这些子序列，以得到有序的数组。

蒂姆排序算法步骤

1. 根据输入数组长度确定最小有序子序列长度 minRun。
2. 划分成长度不小于 minRun 的非递减或严格递减的连续子序列，即 run。不足 minRun 的使用插入排序补足。如果得到的 run 为严格递减的子序列，则将其进行翻转反序（即首尾两两交换）。
3. 对 run 进行合并。用栈来依次存储 run，要求栈中任意三个连续的 run，S_x、S_{x+1}、S_{x+2} 的长度满足限制
   ① $|S_{x+2}| > |S_{x+1}| + |S_x|$
   ② $|S_{x+1}| > |S_x|$
4. 从上到下依次归并所有的 run。

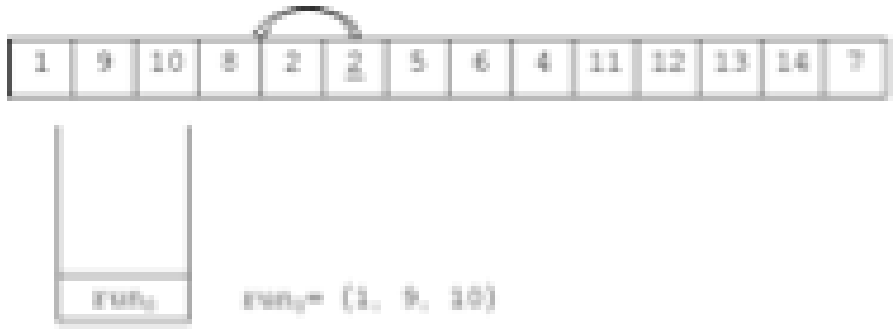
蒂姆排序执行过程示例

初始数组和初始栈



插入排序生成第一个 run

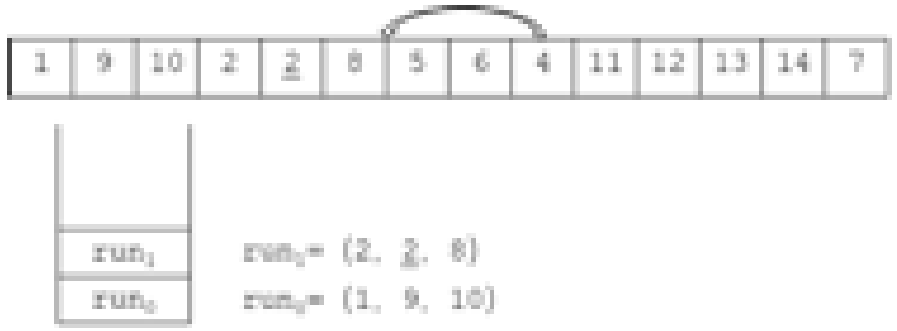
run_0=(1, 9, 10)



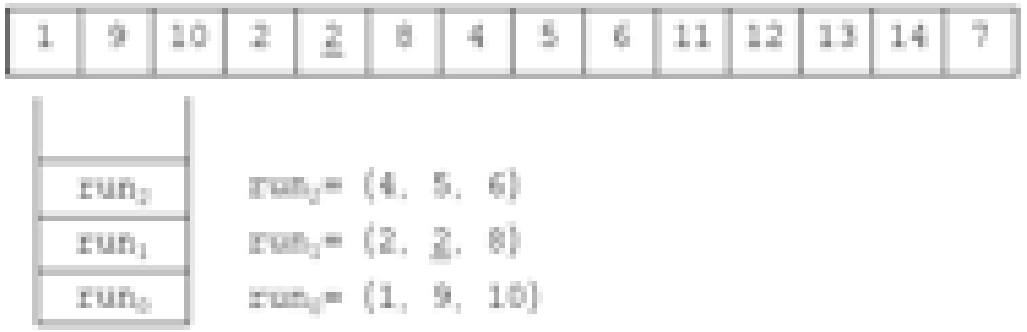
插入排序生成第二个 run

蒂姆排序执行过程示例

原地翻转得到 $run_1=(2, \underline{2}, 8)$



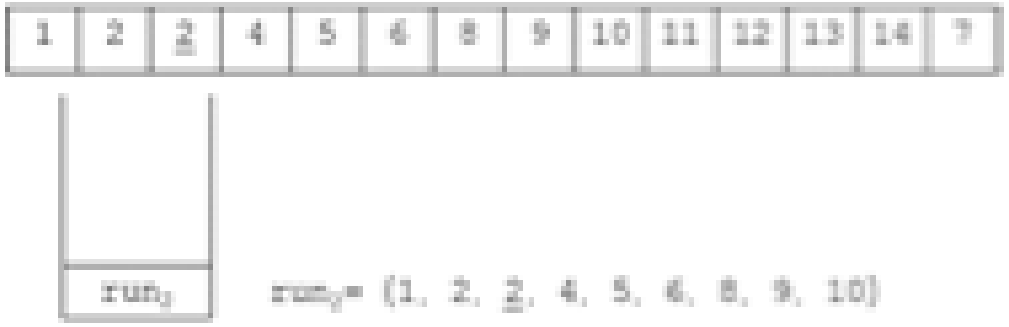
插入排序生成第三个 run



$run_2=(4, 5, 6)$

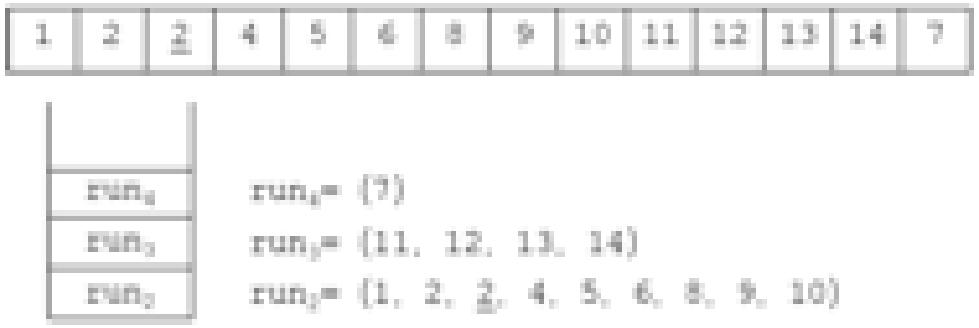
归并 run_2 和 run_1

蒂姆排序执行过程示例



归并 run_2 和 run_0

run_3=(11, 12, 13, 14)



run_4=(7)

由栈顶到栈底依次归并每个 run

蒂姆排序伪代码

算法 10-22 蒂姆排序 TimSort(a, l, r)

输入： 序列 a，左端点下标 l，右端点下标 r

输出： 调整 $a_l, a_{l+1}, \ldots, a_r$ 元素顺序，使元素按照非递减顺序排列

```
⟨run_1, ..., run_k⟩ ←待排序数组a的连续子序列分解
InitStack(S)//将栈内run按栈顶到栈底的顺序从S_1开始编号
for i ← 1 to k do
    Push(S, run_i) //将run_i加入栈S顶部
    while true do
        if |S| ≥ 3 且 |S_1| > |S_3| then //不满足限制
            Merge(S_2, S_3)
        else if |S| ≥ 2 且 |S_1| ≥ |S_2| then //不满足限制
            Merge(S_1, S_2)
        else if |S| ≥ 3 且 |S_1| + |S_2| ≥ |S_3| then //不满足限制
            Merge(S_1, S_2)
        else if |S| ≥ 4 且 |S_2| + |S_3| ≥ |S_4| then //不满足限制
            Merge(S_1, S_2)
        else
            break
        end
    end
end
while |S| > 1 do
    Merge(S_1, S_2)
end
for i ← l to r do
    a_i ← s_{1,i−l+1}
end
```

蒂姆排序性能分析

时间复杂度：将数组分解成 run 可以在线性时间内完成。上述伪代码的主循环中最多需要 $O(n|S|_{max})$ 次比较，其中 $|S|_{max}$ 为栈的最大高度，且满足 $|S|_{max} \leq \log n$。最后的归并使用了启发式的从小长度到大长度的归并，时间复杂度为 $O(n \log n)$，故总体的时间复杂度为 $O(n \log n)$ 。

空间复杂度：算法的空间复杂度为 $O(n)$。

10.10 应用场景：考试录取中的成绩排序

在考试录取工作中，考试和录取按省份分别开展，各省考试人数从**几万到一百多万**不等，这样规模的录取工作需要计算机来处理，其中最核心的步骤就是将考生按考分进行排序，**排序的效率**会直接决定录取工作的处理时长。

在高考这个实际的例子中，由于考生众多，很容易出现**总分同分**的现象。为了决定在总分同分时的录取顺序，还需要制定更详细的比较规则。以浙江省为例，在录取时会依次按文化总分、语文数学总分、语文或数学单科成绩、外语成绩、选考科目单科成绩由高到低，以及志愿号由小到大决定录取顺序。

从排序的角度来讲，文化总分是排序的**主关键字**（第一关键字），语文数学总分是**次关键字**（第二关键字），语文单科成绩是**第三关键字**，依次类推。这些比较项目全部相同的考生为同位次，当然出现这种情况的可能性就非常小了。

10.11 小结

本章我们介绍了若干经典的排序方法，包括：插入排序、冒泡排序、选择排序、归并排序、快速排序、堆排序、希尔排序、计数排序、基数排序、桶排序，还扩展介绍了C++、Java和Python的库函数中实现的内省排序和蒂姆排序。

排序方法	平均时间复杂度	最坏时间复杂度	最坏辅助空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
归并排序	$O(nlogn)$	$O(nlogn)$	$O(n)$	稳定
快速排序	$O(nlogn)$	$O(^{n^2})$	$O(n)$	不稳定
堆排序	$O(nlogn)$	$O(nlogn)$	$O(1)$	不稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
基数排序（LSD）	$O\big(d(n+radix)\big)$	$O\big(d(n+radix)\big)$	$O(n+radix)$	稳定
桶排序	$O(n+k+\frac{n^2}{k})$	$O\big(n^2+k\big)$	$O(n+k)$	稳定
内省排序	$O(nlogn)$	$O(nlogn)$	$O(logn)$	不稳定
蒂姆排序	$O(nlogn)$	$O(nlogn)$	$O(n)$	稳定

谢谢观看