

## 第二部分：排序和顺序统计量

INTRODUCTION TO

# ALGORITHMS

THIRD EDITION

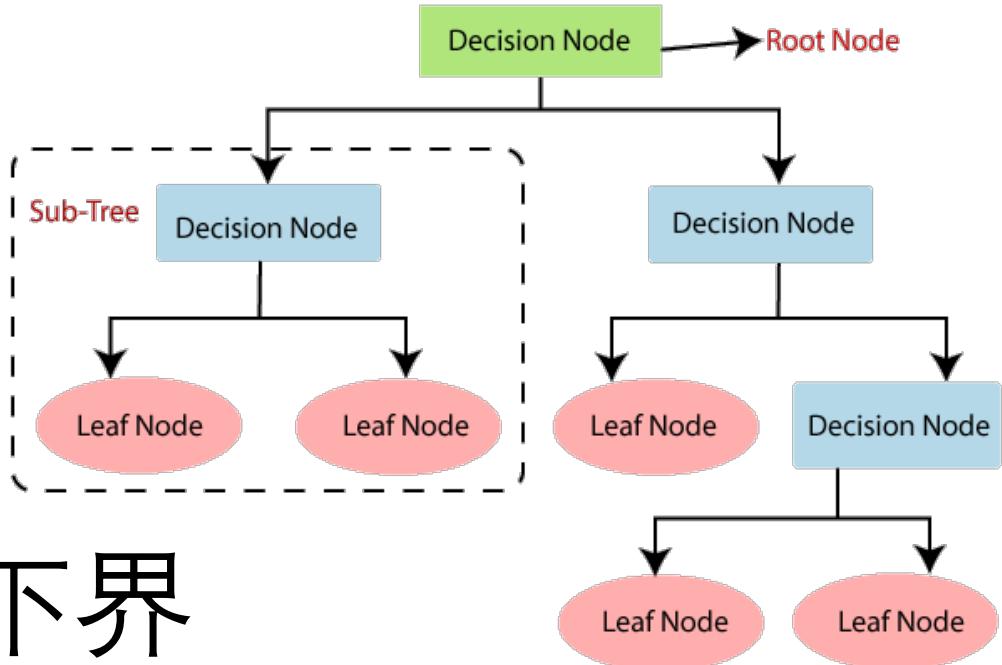
# 线性时间排序

《算法导论》—— 第7讲

*jiacaicui@163.com*

# 内容提要

- 排序算法的下界
  - 基于比较的排序算法最坏情况运行时间的下界为  $\Omega(n \log n)$
  - 决策树模型
- 线性时间排序
  - 不基于比较，但是要增加限制条件
  - 计数排序
  - 基数排序
  - 桶排序



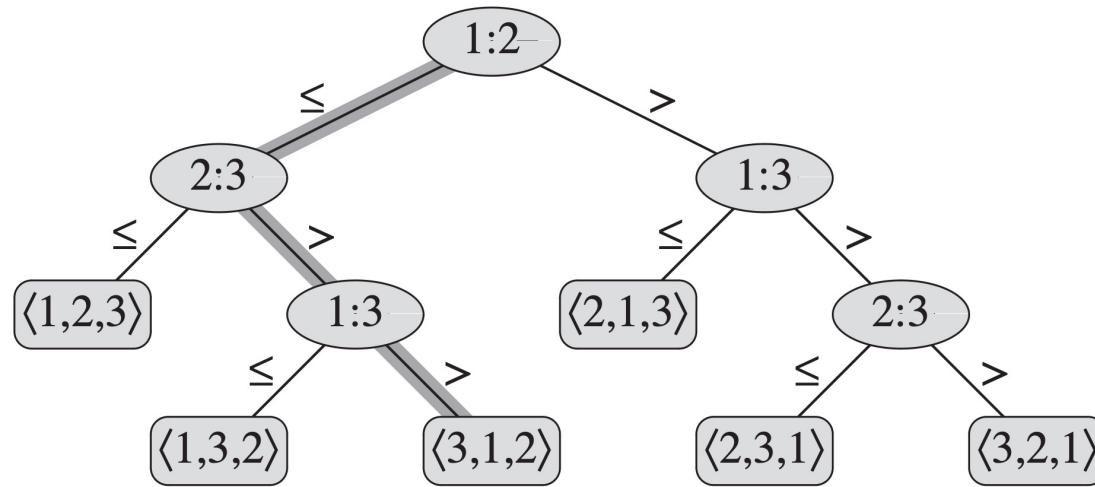
# 排序算法的下界

基于决策树模型证明

# 比较排序

- 在一个比较排序算法中，我们只使用元素间的比较来获得输入序列  $\langle a_1, a_2, \dots, a_n \rangle$  中元素间的次序信息。
  - $a_i < a_j$ 、 $a_i \leq a_j$ 、 $a_i = a_j$ 、 $a_i \geq a_j$ 、 $a_i > a_j$
- 不失一般性，后续分析中，不妨假设所有的输入元素都是互异的。
  - 于是  $a_i = a_j$  的比较就没有意义了，因此我们可以假设不需要这种比较。
  - 注意到此时， $a_i < a_j$ 、 $a_i \leq a_j$ 、 $a_i > a_j$  和  $a_i \geq a_j$  是等价的，因此我们可以假设所有的比较采用的都是  $a_i \leq a_j$  的形式。

# 决策树模型



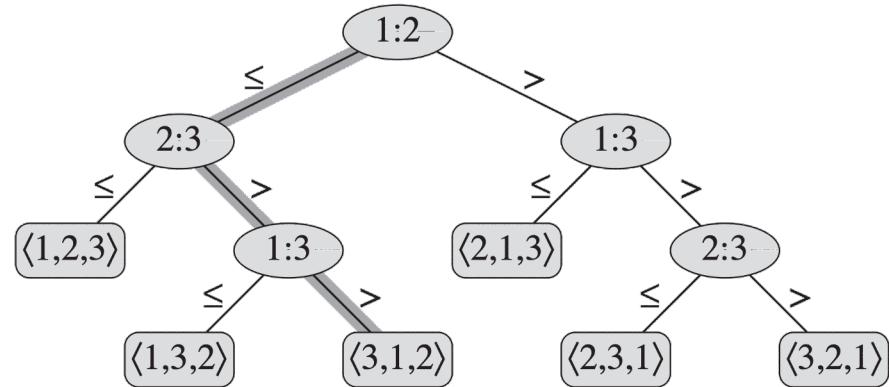
- 内部结点标记为  $i:j$ ，表示  $a_i$  与  $a_j$  之间的一次  $\leq$  比较；
- 左子树表示确定  $a_i \leq a_j$  后的比较，右子树表示  $a_i < a_j$  后的比较。
- 叶结点标记为  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ ，是  $\langle 1, 2, \dots, n \rangle$  的一个排列，表示得到的顺序  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ 。
- 对于一个具体输入实例的排序过程对应于决策树中的一条路径，例如该决策树对于输入  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$  的排序过程对应于上图阴影路径。

# 决策树模型

INSERTION-SORT( $A$ )

```

1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```



- 决策树是对于基于比较排序的抽象，我们只关心比较操作，忽略诸如控制、数据移动等其他操作。
  - 例如上面右图是插入排序在输入序列长度为 3 的情况下的决策树。
  - 3 个元素共有  $3! = 6$  种排列，因此决策树至少有 6 个结点（可能重复）。
- 一个比较排序算法对于一个输入实例排序需要进行的比较的次数，就是其决策树中从对应的一条从根结点到叶结点的路径长度。

# 最坏情况的下界

定理8.1：在最坏情况下，任何比较排序算法都需要做  $\Omega(n \log n)$  次比较。

证明：一个正确的比较排序算法的决策树至少有  $n!$  个叶子结点。记一个高度为  $h$  的二叉树最多有  $2^h$  个叶子结点，其中  $h$  就是决策树中最长路径的长度，也就是最坏情况下的比较次数。于是我们有

$$n! \leq 2^h \Rightarrow h \geq \log n! = \Theta(n \log n)$$

其中， $\log n! = \Theta(n \log n)$  详见练习2-2问题2。

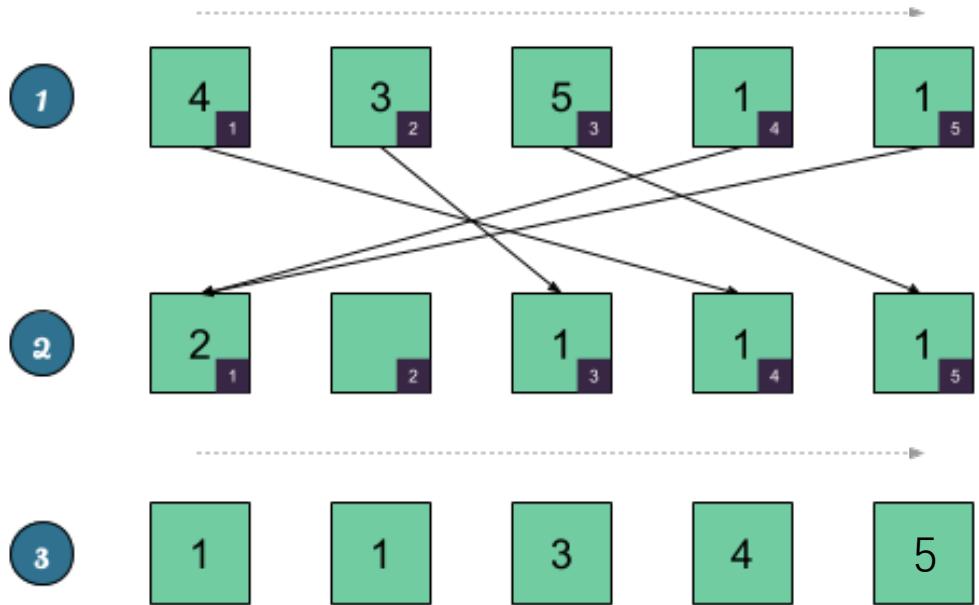
于是  $h = \Omega(n \log n)$ ，即任何比较排序算法最坏情况下都需要做  $\Omega(n \log n)$  次比较。

推论8.2：堆排序和归并排序都是渐近最优的比较排序算法。

证明：堆排序和归并排序的运行时间上界为  $O(n \log n)$ ，这与最坏情况的下界是一致的。

# 计数排序

对于一定范围内的数据排序



# 计数排序

- 前提条件：假设  $n$  个输入元素中的每一个都是在 0 到  $k$  区间内的一个整数。
- 运行时间：当  $k = O(n)$  时，计数排序的运行时间为  $O(n)$ 。
- 基本思想：对每一个输入元素  $x$ ，确定小于  $x$  的元素个数。
  - 利用这一信息，可以直接把  $x$  放到它在输出数组中的位置上。
  - 例如，如果有 17 个元素小于  $x$ ，则  $x$  应该在第 18 个输出位置上。
  - 当有几个元素相同时，往后顺延即可。

# 计数排序伪代码

---

COUNTING-SORT( $A, B, k$ )

```

1 let  $C[0..k]$  be a new array
2 for  $i = 0$  to  $k$ 
3    $C[i] = 0$ 
4 for  $j = 1$  to  $A.length$ 
5    $C[A[j]] = C[A[j]] + 1$ 
6 //  $C[i]$  now contains the number of elements equal to  $i$ .
7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$ 
9 //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11    $B[C[A[j]]] = A[j]$ 
12    $C[A[j]] = C[A[j]] - 1$ 
```

- $A[1..n]$  为输入数组， $A.length = n$ ， $B[1..n]$  为输出数。
- $C[0..k]$  提供临时存储空间（空间换时间）。

# 计数排序过程

COUNTING-SORT( $A, B, k$ )

```

1 let  $C[0..k]$  be a new array
2 for  $i = 0$  to  $k$ 
3    $C[i] = 0$ 
4 for  $j = 1$  to  $A.length$ 
5    $C[A[j]] = C[A[j]] + 1$ 
```

6 //  $C[i]$  now contains the number of elements equal to  $i$ .

1	2	3	4	5	6	7	8	
$A$	2	5	3	0	2	3	0	3
$C$	0	1	2	3	4	5		
$C$	2	0	2	3	0	1		

```

7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$ 
```

9 //  $C[i]$  now contains the number of elements less than or equal to  $i$ .

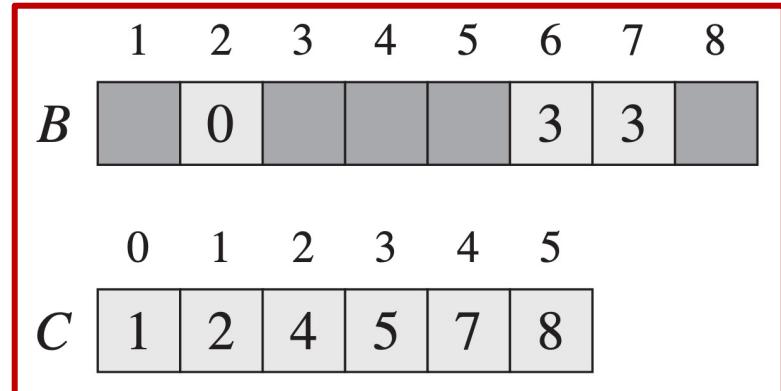
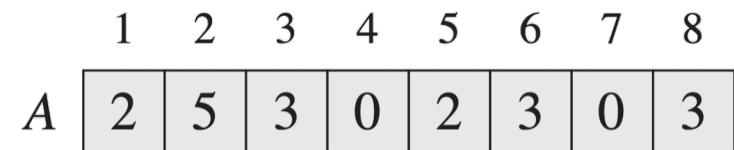
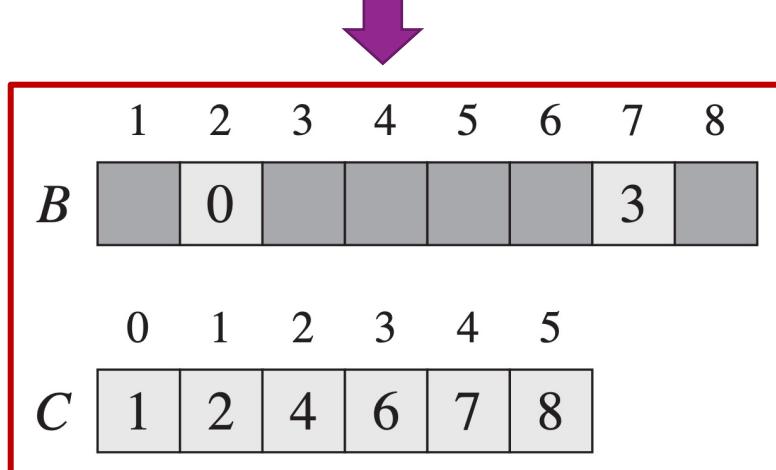
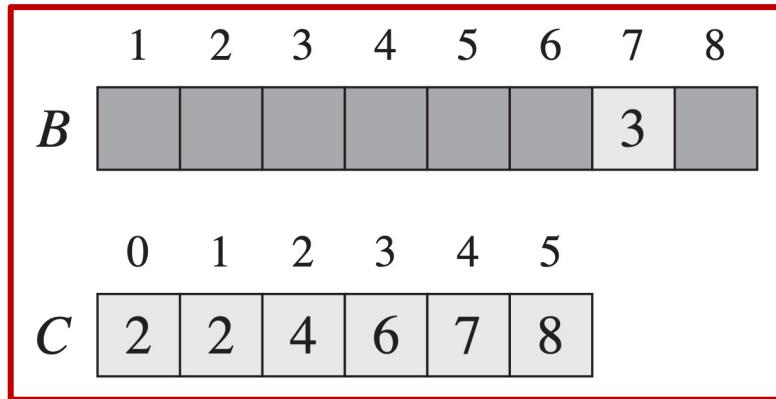
0	1	2	3	4	5	
$C$	2	2	4	7	7	8

# 计数排序过程

```

10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1

```



# 计数排序的运行时间

COUNTING-SORT( $A, B, k$ )

```
1 let  $C[0..k]$  be a new array
2 for  $i = 0$  to  $k$ 
3    $C[i] = 0$ 
4 for  $j = 1$  to  $A.length$ 
5    $C[A[j]] = C[A[j]] + 1$ 
6 //  $C[i]$  now contains the number of elements equal to  $i$ .
7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$ 
9 //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11    $B[C[A[j]]] = A[j]$ 
12    $C[A[j]] = C[A[j]] - 1$ 
```

- 不难看出，运行时间为  $\Theta(n + k)$ ，一般我们在  $k = O(n)$  时采用计数排序，此时运行时间为  $\Theta(n)$ 。

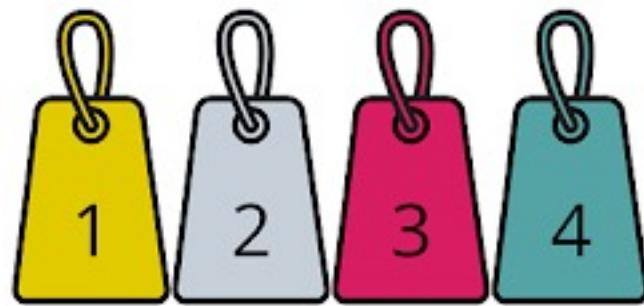
# 计数排序的稳定性

```
10  for j = A.length downto 1  
11      B[C[A[j]]] = A[j]  
12      C[A[j]] = C[A[j]] - 1
```

- 计数排序是**稳定的**：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的相对次序相同。
  - 通常，这种稳定性只有当进行排序的数据还附带卫星数据时才比较重要。
  - 选择排序、堆排序、快速排序、希尔排序是不稳定的。
  - 计数排序稳定性证明详见练习7-1问题5。
- 计数排序经常会被用作**基数排序**算法的一个子过程。
  - 为了使基数排序正确运行，计数排序必须是稳定的。

# 基数排序

卡片排序机的算法



# 基数排序 (radix sort)

RADIX-SORT( $A, d$ )

- 1 **for**  $i = 1$  **to**  $d$
- 2       use a stable sort to sort array  $A$  on digit  $i$

- 假设  $n$  个  $d$  位的元素存放在数组  $A$  中，其中第 1 位是最低位，第  $d$  位是最高位。
- 核心思路：从低位到高位一列一列地“稳定”排序。

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

# 基数排序的正确性和运行时间

RADIX-SORT( $A, d$ )

- 1 **for**  $i = 1$  **to**  $d$
- 2       use a stable sort to sort array  $A$  on digit  $i$

- 引理8.3：给定  $n$  个  $d$  位数，其中每一个数位有  $k$  个可能的取值。如果 RADIX-SORT 使用的**稳定排序**方法耗时  $\Theta(n + k)$ ，那么它就可以在  $\Theta(d(n + k))$  时间内将这些数排好序。
  - 证明：运行时间显然，正确性证明详见练习7-2问题1。
- 当  $d = O(1)$  且  $k = O(n)$  时，基数排序具有线性的时间代价。
  - 基数排序可以很方便地对于**多关键字域的记录**进行排序，例如**日期**，**时间**等等。
  - 在更一般的情况下，我们可以灵活地决定如何将每个关键字分解成若干位。

# 更灵活的关键字分解

- 引理8.4：给定  $n$  个  $b$  位数和任何正整数  $r \leq b$ ，如果 RADIX-SORT 使用的稳定排序算法对数据区间是 0 到  $k$  的输入进行排序耗时  $\Theta(n + k)$ ，那么它就可以在  $\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$  时间内将这些数排好序。

- 证明：对于一个值  $r \leq b$ ，每个关键字可以看做是  $d = \left\lceil \frac{b}{r} \right\rceil$  个  $r$  位数。每个数都是在 0 到  $2^r - 1$  之间的一个整数，这样就可以采用计数排序，其中  $k = 2^r - 1$ 。

- 例如我们可以将一个 32 位的数看成是 4 个 8 位的数，于是有  $b = 32$ ，  
 $r = 8$ ， $k = 2^r - 1 = 255$  和  $d = \frac{b}{r} = 4$ 。

- 每一轮排序花费的时间为  $\Theta(n + k) = \Theta(n + 2^r)$ ，总时间代价为

$$\Theta(d(n + 2^r)) = \Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$$

# r 的选择

- 引理8.4：给定  $n$  个  $b$  位数和任何正整数  $r \leq b$ ，如果 RADIX-SORT 使用的稳定排序算法对数据区间是 0 到  $k$  的输入进行排序耗时  $\Theta(n + k)$ ，那么它就可以在  $\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$  时间内将这些数排好序。
- 对于给定的  $n$  和  $b$ ，我们希望选择合适的  $r(r \leq b)$ ，使得  $\left(\frac{b}{r}\right)(n + 2^r)$  最小。
  - 如果  $b < \lfloor \log n \rfloor$ ，则对于任何满足  $r \leq b$  的  $r$ ，都有  $n + 2^r = \Theta(n)$ ，显然，选择  $r = b$  得到的时间代价为  $\left(\frac{b}{b}\right)(n + 2^b) = \Theta(n)$ ，这一结果是渐近意义上最优的。
  - 如果  $b \geq \lfloor \log n \rfloor$ ，选择  $r = \lfloor \log n \rfloor$  得到的运行时间为  $\Theta\left(\frac{bn}{\log n}\right)$ 。

# 基数排序 .vs 比较排序

- 基数排序是否比基于比较的排序更好呢？
  - 如果  $b = O(\log n)$ , 且我们选择  $r \approx \log n$  时, 基数排序的运行时间为  $\Theta(n)$ , 看上去渐近优于基于比较的排序的  $\Theta(n \log n)$ , 但隐藏在  $\Theta$  符号背后的常数因子是不同的。
  - 哪一个排序算法更适合依赖于具体实现和底层硬件的特性 (例如, 快速排序通常可以比基数排序更有效地使用硬件缓存), 以及输入数据的特征。
  - 此外, 利用计数排序作为中间稳定排序的基数排序不是原址排序, 而很多  $\Theta(n \log n)$  时间的比较排序是原址排序。

# 桶排序

粗处划分，细处排序



# 桶排序 (bucket sort)

- 桶排序假设输入数据服从某个区间上的均匀分布，平均情况下它的时间代价为  $O(n)$ 。
  - 不失一般性，假设输入数据均匀、独立地分布在  $[0, 1)$  区间上。
  - 核心思路：
    - 将  $[0, 1)$  区间划分为  $n$  个相同大小的子区间（称为桶），然后将  $n$  个输入分别放到各个桶中。
    - 因为输入数据均匀、独立地分布在  $[0, 1)$  区间上，所以一般不会出现很多数落在同一个桶中的情况。
    - 为了得到输出结果，我们先对每个桶中的数进行排序，然后遍历每个桶，按照次序把各个桶中的元素列出来即可。

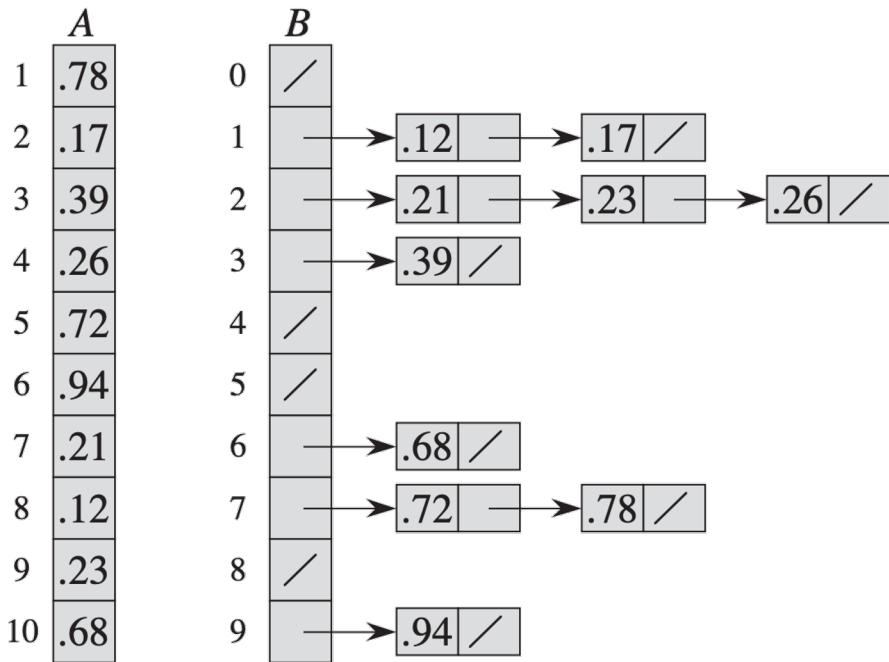
# 桶排序的伪代码与过程

粗处：按照区间划分成桶

**BUCKET-SORT( $A$ )**

```

1 let  $B[0..n - 1]$  be a new array
2  $n = A.length$ 
3 for  $i = 0$  to  $n - 1$ 
4     make  $B[i]$  an empty list
5 for  $i = 1$  to  $n$ 
6     insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$ 
7 for  $i = 0$  to  $n - 1$ 
8     sort list  $B[i]$  with insertion sort
9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order
    
```



(a)

(b)

} 细处：区间内部插入排序

# 桶排序的正确性

- 考虑任意两个元素  $A[i]$  和  $A[j]$ ，说明它们最终被正确排序了。
  - 不失一般性，不妨假设  $A[i] \leq A[j]$ 。
  - 由于  $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$ ，元素  $A[i]$ ，
    - 要么与  $A[j]$  被放入了同一个桶中，则算法第 7 到 8 行的 **for** 循环会将他们按适当的顺序排列。
    - 要么被放入一个下标比  $A[j]$  所在的桶的下标更小的桶中，则算法第 9 行会将他们按照适当的顺序排列。
  - 因此，桶排序是正确的。

# 桶排序的运行时间

BUCKET-SORT( $A$ )

```

1 let  $B[0..n - 1]$  be a new array
2  $n = A.length$ 
3 for  $i = 0$  to  $n - 1$ 
4     make  $B[i]$  an empty list
5 for  $i = 1$  to  $n$ 
6     insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$ 
7 for  $i = 0$  to  $n - 1$ 
8     sort list  $B[i]$  with insertion sort
9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order

```

- 将数据放入桶中和从桶中取出的代价为  $\Theta(n)$ ；
- 假设  $n_i$  是表示桶  $B[i]$  中元素个数的随机变量，对桶  $B[i]$  插入排序的代价为  $O(n_i^2)$ 。
- 总运行时间：

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

# 桶排序的运行时间

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- 最坏情况：输入极不均匀，所有的元素都落在了一个桶中， $T(n) = O(n^2)$ 。
- 平均情况：

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

- 其中，如果输入服从均匀分布， $E[n_i^2] = 2 - \frac{1}{n}$ （下一页PPT证明），于是

$$E[T(n)] = \Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) = \Theta(n)$$

- 因此，输入均匀服从均匀分布时，桶排序可以在线性时间内完成。
  - 其实只要所有桶期望大小的平方和不超过  $\Theta(n)$  即可。

# 平均情况下桶大小平方的期望

定义  $A[j]$  落入桶  $i$  为事件  $H_{ij}$  ,  $X_{ij} = I\{H_{ij}\}$  , 则,

$$n_i = \sum_{j=1}^n X_{ij}, \quad E[X_{ij}] = \Pr\{H_{ij}\} = \frac{1}{n}$$

$$E[X_{ij}^2] = 1^2 \cdot \Pr\{H_{ij}\} + 0^2 \cdot (1 - \Pr\{H_{ij}\}) = \frac{1}{n}$$

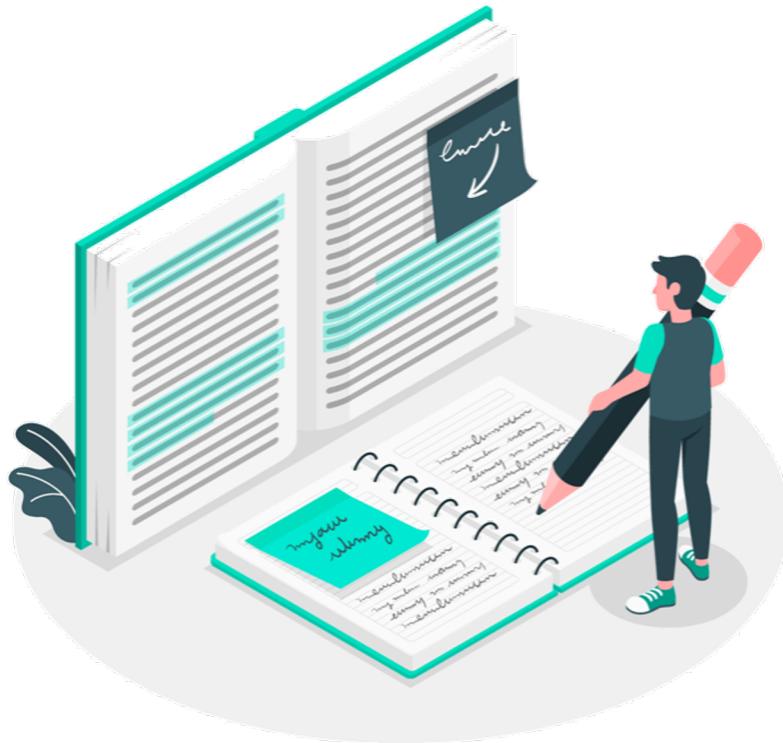
其中由于输入元素是均匀独立的, 我们有任意  $X_{ij}$  与  $X_{ik}$  相互独立, 则

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

于是

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] = \sum_{j=1}^n E[X_{ij}^2] + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^n E[X_{ij}X_{ik}] = n \cdot \frac{1}{n} + 2C_n^2 \cdot \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} = 2 - \frac{1}{n} \end{aligned}$$

# 本讲小结



# 内容提要

- 排序算法的下界
  - 基于比较的排序算法最坏情况运行时间的下界为  $\Omega(n \log n)$
  - 决策树模型
- 线性时间排序
  - 不基于比较，但是要增加限制条件
  - 计数排序
  - 基数排序
  - 桶排序

The End!

