海量资源：666java.com

排序算法下界& 线性时间排序

海量资源：666java.com

上次讲到

- 随机算法
- BogoSort & **QuickSort**

海量资源：666java.com

目前为止我们学了

- 分治算法
  - Karatsuba
  - MergeSort
  - SELECT
  - QuickSort
- 渐进分析
  - 如何证明 大-O, 大-Omega, 大-Theta
- 递归式: 递归树/表, 主定理, 代入法
  - 分治算法本质上是递归的(解决规模较小的子问题), 分析它们的运行时需要推导/求解递归式
- 证明算法正确性(通过归纳法)
- 随机算法

海量资源：666java.com

今天的内容

- 排序算法时间复杂度的下界
  - 我们使用什么样的**计算模型**?
- 线性时间的排序

海量资源：666java.com

排序算法下界

我们学习了 O(n log n) 的排序... 还能更快吗?

海量资源：666java.com

O(n log n) 的排序

- 归并排序
  - 最坏情况 $\Theta(n \log n)$
- 快速排序
  - 期待运行时间/平均情况: $\Theta(n \log n)$

海量资源：666java.com

O(n log n) 的排序

- 归并排序
  - 最坏情况 $\Theta(n \log n)$
- 快速排序
  - 期待运行时间/平均情

问题来了...

还能更快吗？

海量资源：666java.com

… 意大利面排序算法?

输入: 一串实数

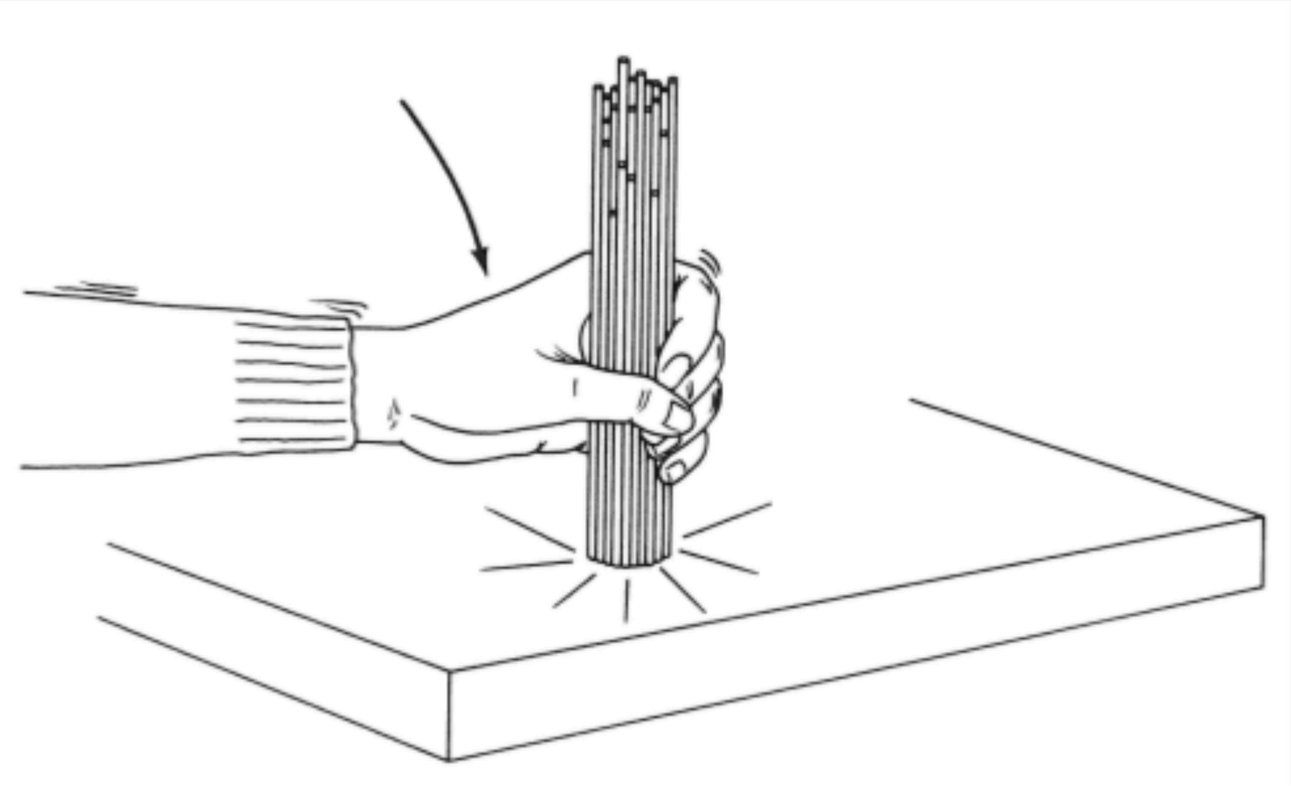
海量资源：666java.com

… 意大利面排序算法?

输入: 一串实数

算法:

- 对于每个输入的数，将一根意面掰至其大小的长度 **O(n)**
- 单手握住这些意面，将一端用桌面对齐 **O(1)**
- 另一只手慢慢从高处向下移动 - 触碰到最长的意面时将其抽出并记录长度, 重复这个过程直到握住的意面全部被抽出 **O(n)**



这个“算法”涉及到一些有趣的问题

海量资源：666java.com

什么是计算模型?

输入: 数组中的元素

输出: 一个排序的数组

允许的操作: 比较

海量资源：666java.com

什么是计算模型?

输入: 数组中的元素

输出: 一个排序的数组

允许的操作: 比较

输入: 一些实数

输出: 排序的实数

允许的操作: 掰断意面, 使用桌面对齐, 移动另一只手

海量资源：666java.com

什么是计算模型?

输入: 数组中的元素

输出: 一个排序的数组

允许的操作: 比较

输入: 一些实数

输出: 排序的实数

允许的操作: 掰断意面, 使用桌面对齐, 移动另一只手

显然我们更关心计算机能实现哪些操作，第一种模型更合适

海量资源：666java.com

基于比较的排序

- **目标是将一个序列的元素进行排序**
- **不能直接访问元素的值: 只能通过 *比较* 的操作来比较任意两个元素的大小关系**
- 例: 插入排序, 归并排序, 快速排序

海量资源：666java.com

基于比较的排序

- **目标是将一个序列的元素进行排序**
- **不能直接访问元素的值: 只能通过 *比较* 的操作来比较任意两个元素的大小关系**
- 例: 插入排序, 归并排序, 快速排序

“基于比较的排序” 对输入唯一的要求是

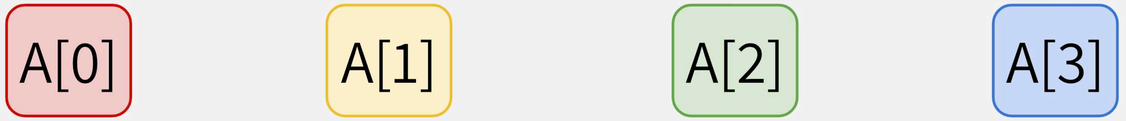
输入元素均属于同一个有序集合(如整数集)

海量资源：666java.com

基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?



海量资源：666java.com

基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

A[0] A[1] A[2] A[3]

某个基于比较的排序

A[1] 大于 A[3]?

是!



海量资源：666java.com

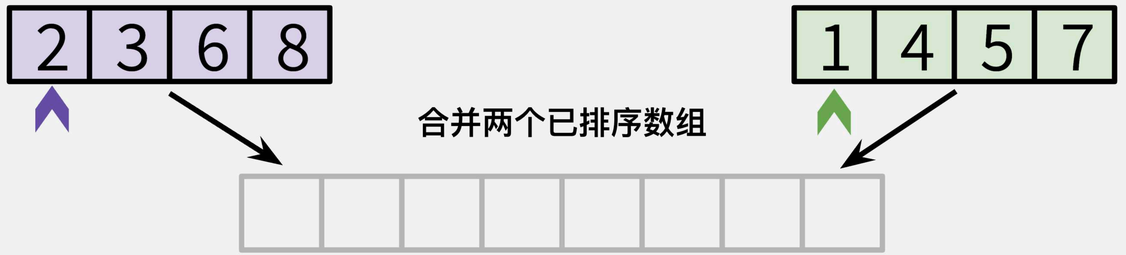
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



海量资源：666java.com

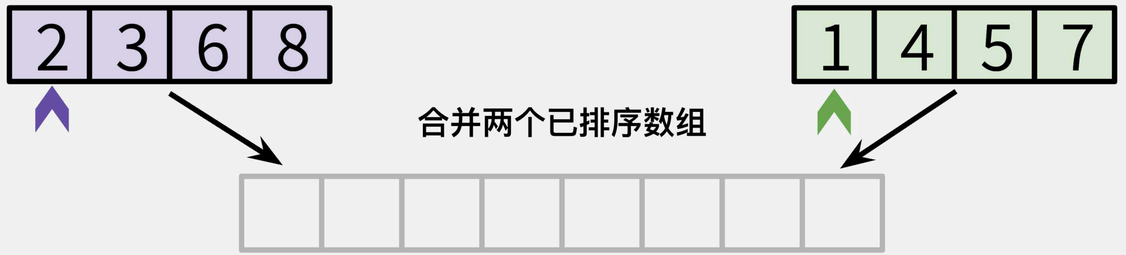
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:





归并排序



海量资源：666java.com

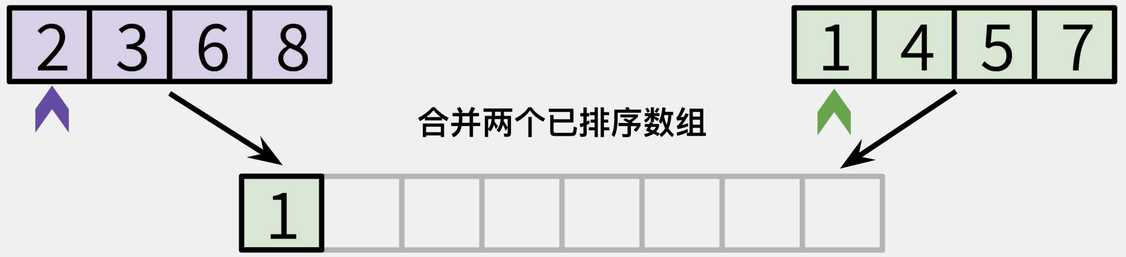
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



归并排序





海量资源：666java.com

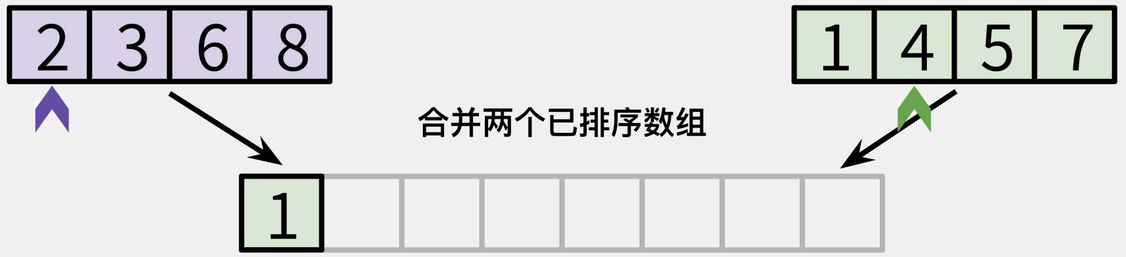
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:





归并排序



海量资源：666java.com

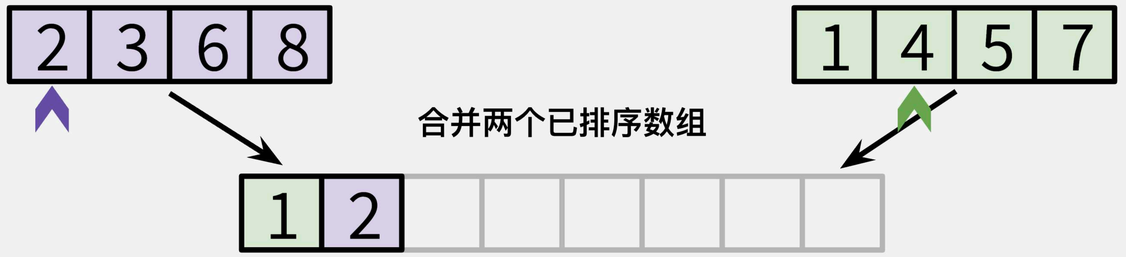
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



归并排序





海量资源：666java.com

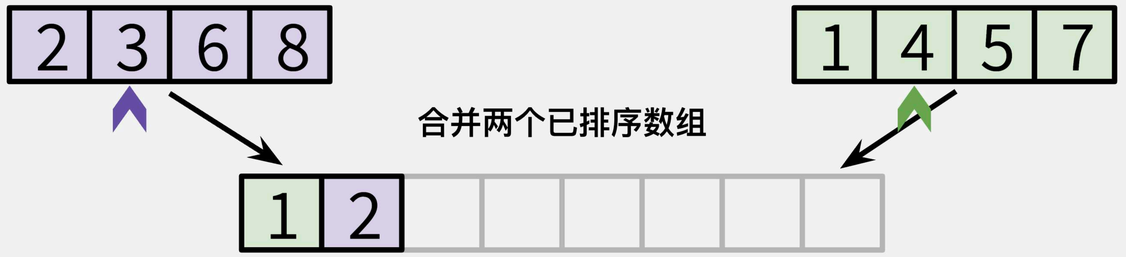
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:





归并排序



海量资源：666java.com

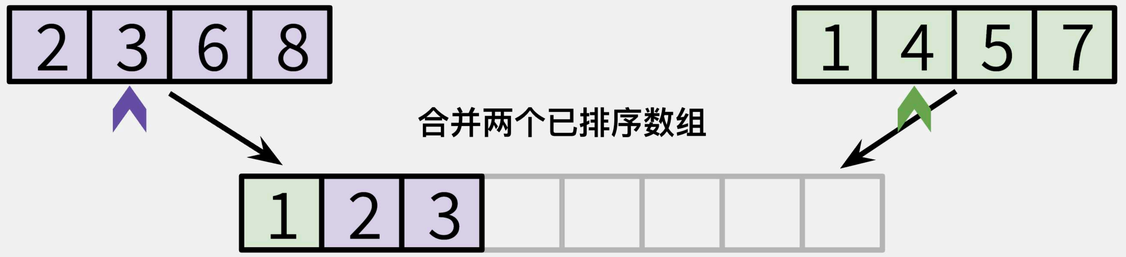
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

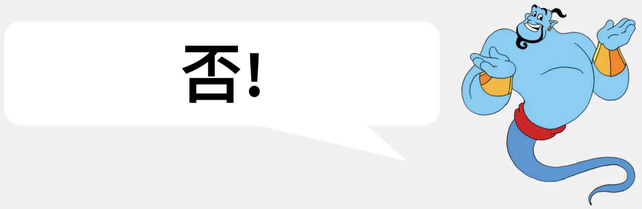
(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



归并排序





海量资源：666java.com

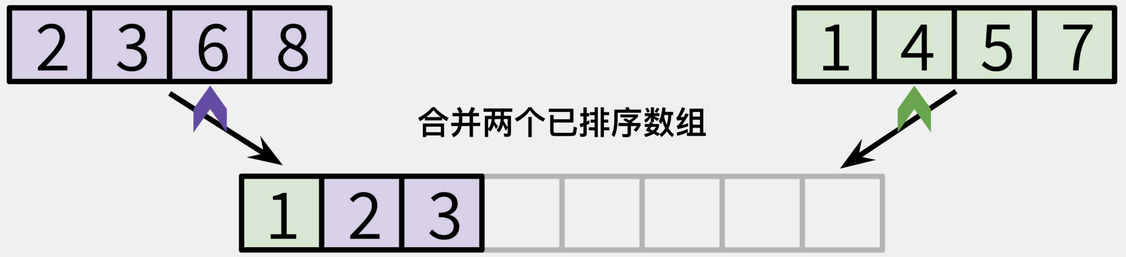
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



6 大于 4 ?

归并排序



海量资源：666java.com

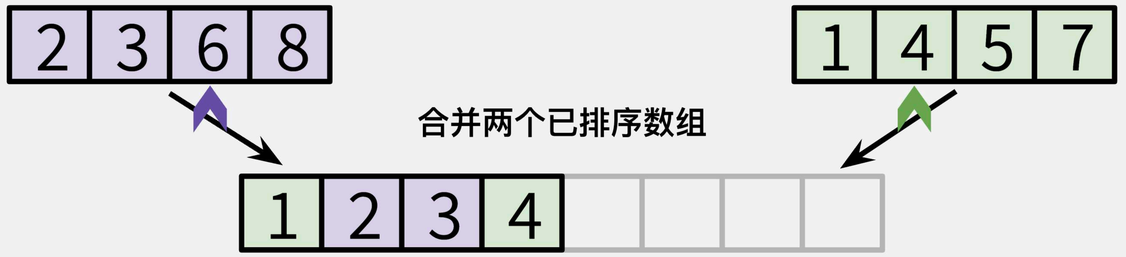
基于比较的排序

即, 唯一允许的操作为:

给定两个下标 i 和 j, A[i] 是否大于 A[j]?

(我们不关心具体的元素大小，只关心上面这个问题的答案)

以归并排序为例:



归并排序

6 大于 4 ?

是!



海量资源：666java.com

基于比较的排序

定理:

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

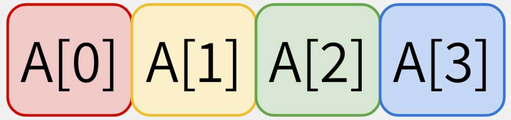
海量资源：666java.com

基于比较的排序

定理:

在最坏情况下，任何比较排序算法都需要做 Ω(n log n) 次比较

想象这是某种算法的输入



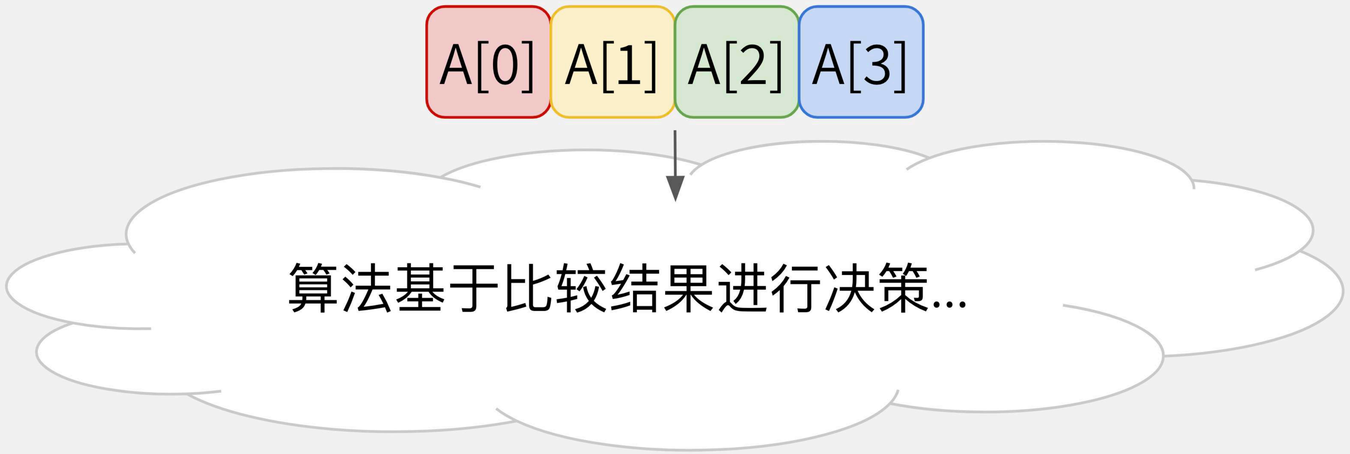
海量资源：666java.com

基于比较的排序

定理:

在最坏情况下，任何比较排序算法都需要做 Ω(n log n) 次比较

想象这是某种算法的输入

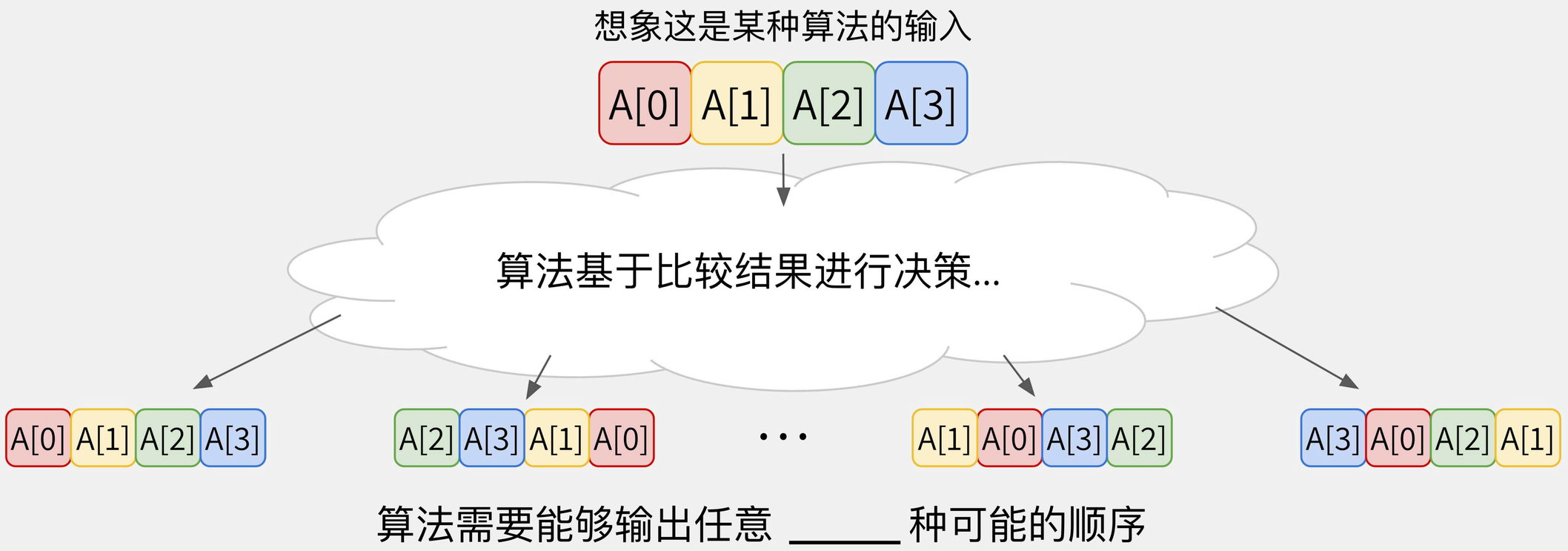


海量资源：666java.com

基于比较的排序

定理:

在最坏情况下，任何比较排序算法都需要做 Ω(n log n) 次比较

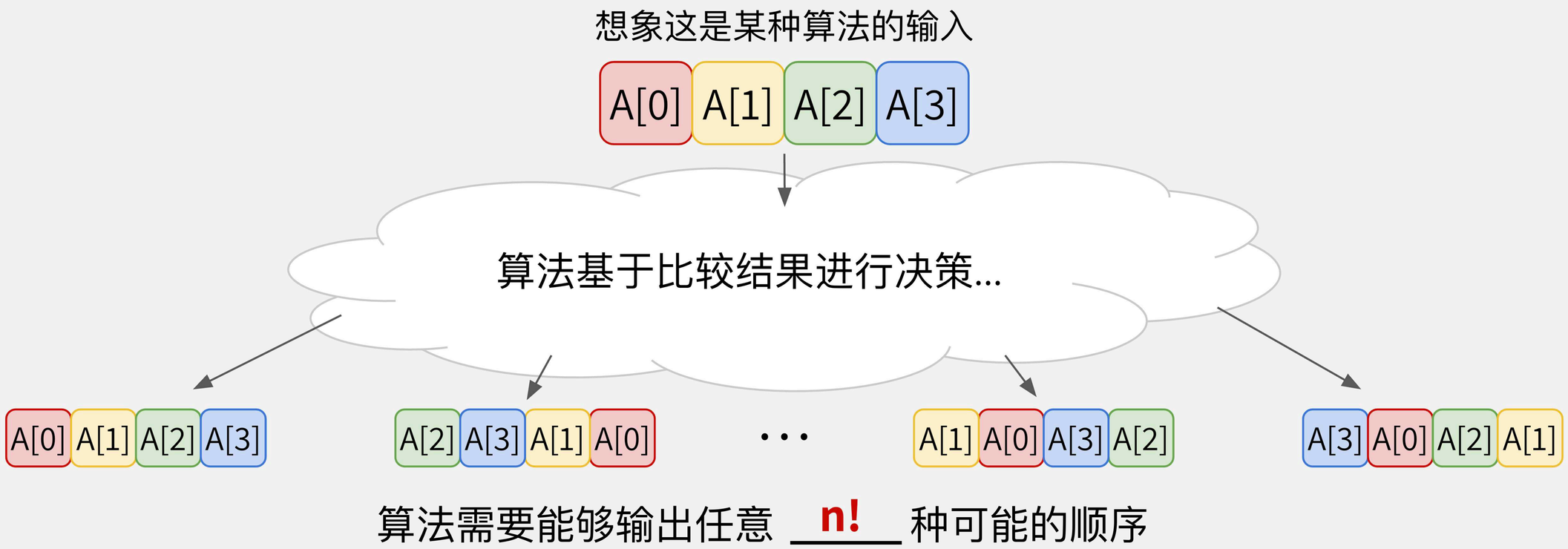


海量资源：666java.com

基于比较的排序

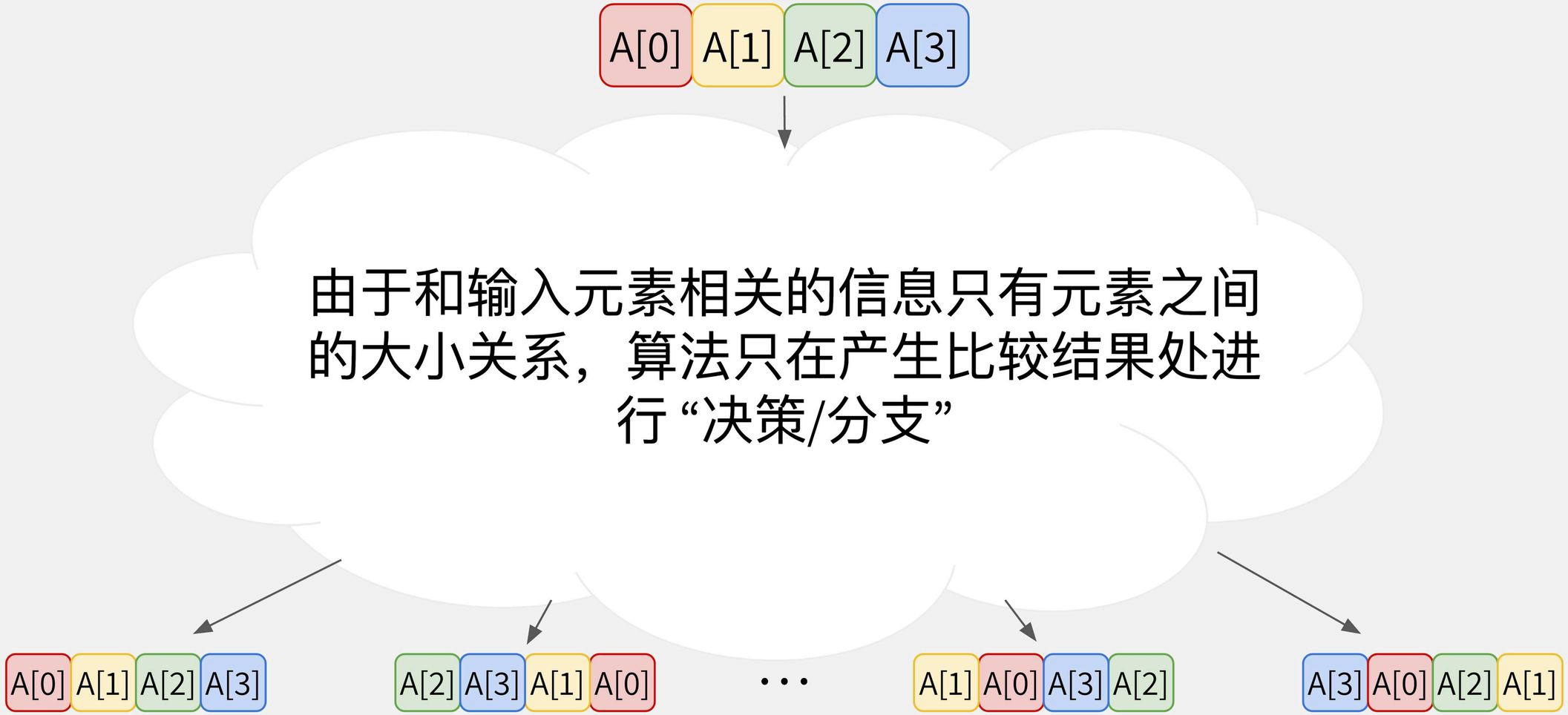
定理:

在最坏情况下，任何比较排序算法都需要做 Ω(n log n) 次比较



海量资源：666java.com

基于比较的排序

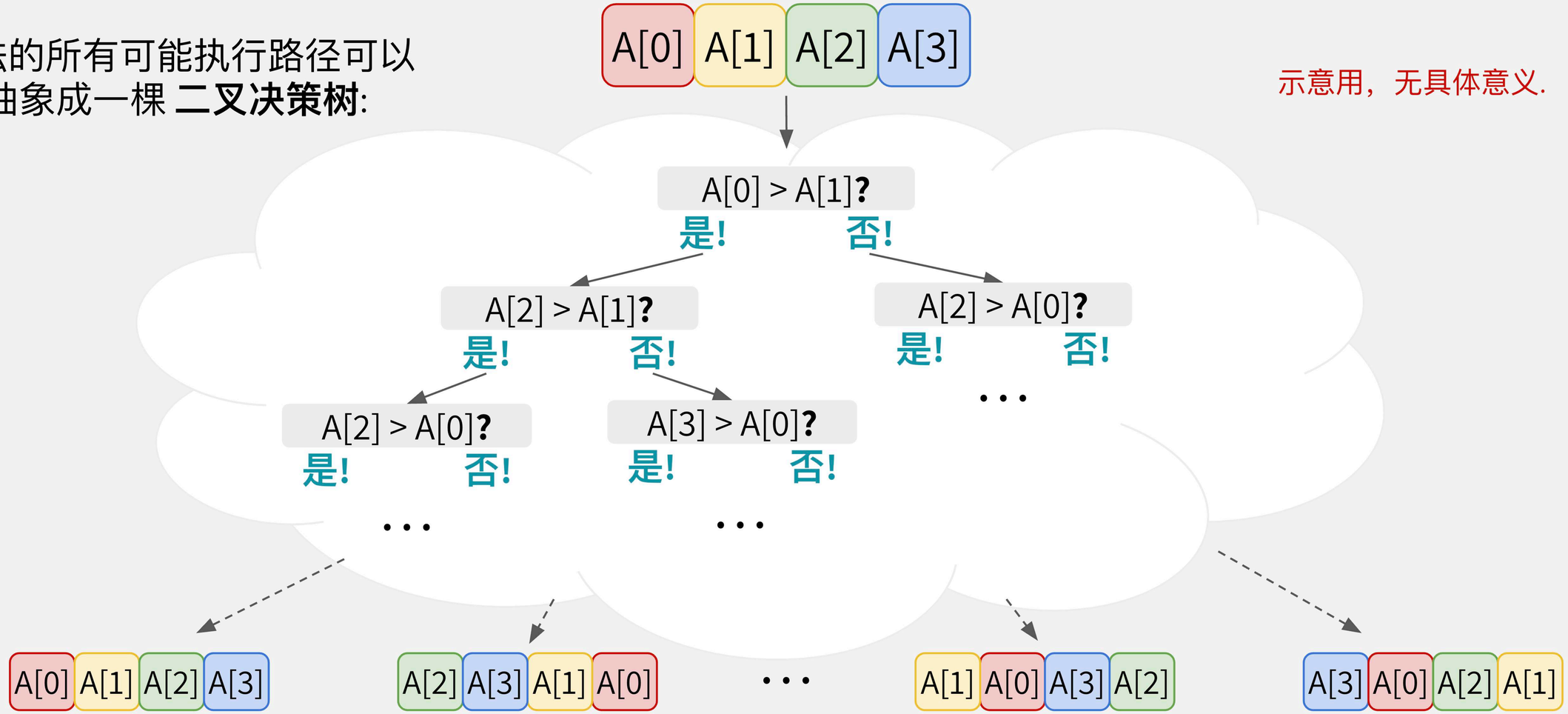


算法需要能够输出任意 **n!** 种可能的顺序

海量资源：666java.com

基于比较的排序

算法的所有可能执行路径可以抽象成一棵 **二叉决策树**:



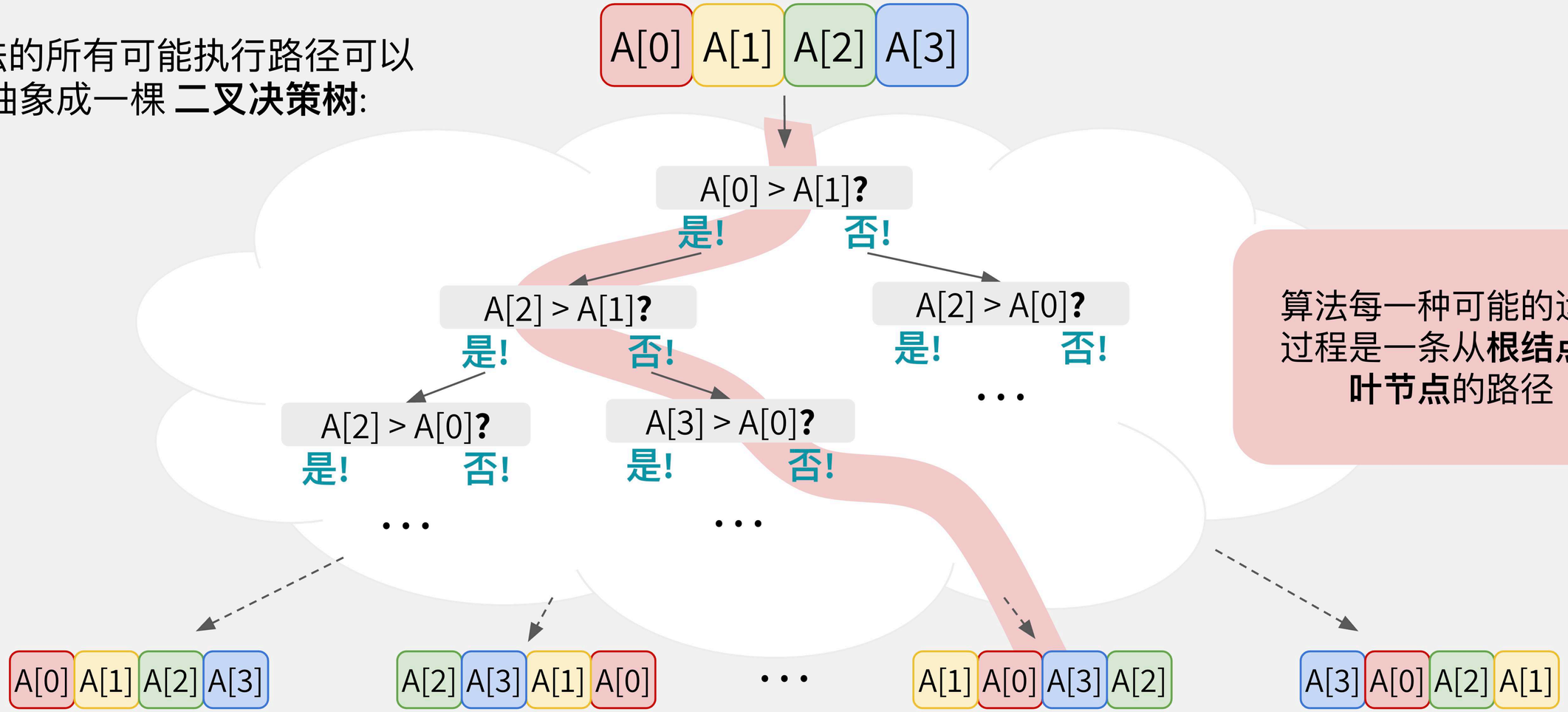
示意用，无具体意义.

算法需要能够输出任意 **n!** 种可能的顺序

海量资源：666java.com

基于比较的排序

算法的所有可能执行路径可以抽象成一棵 **二叉决策树**:



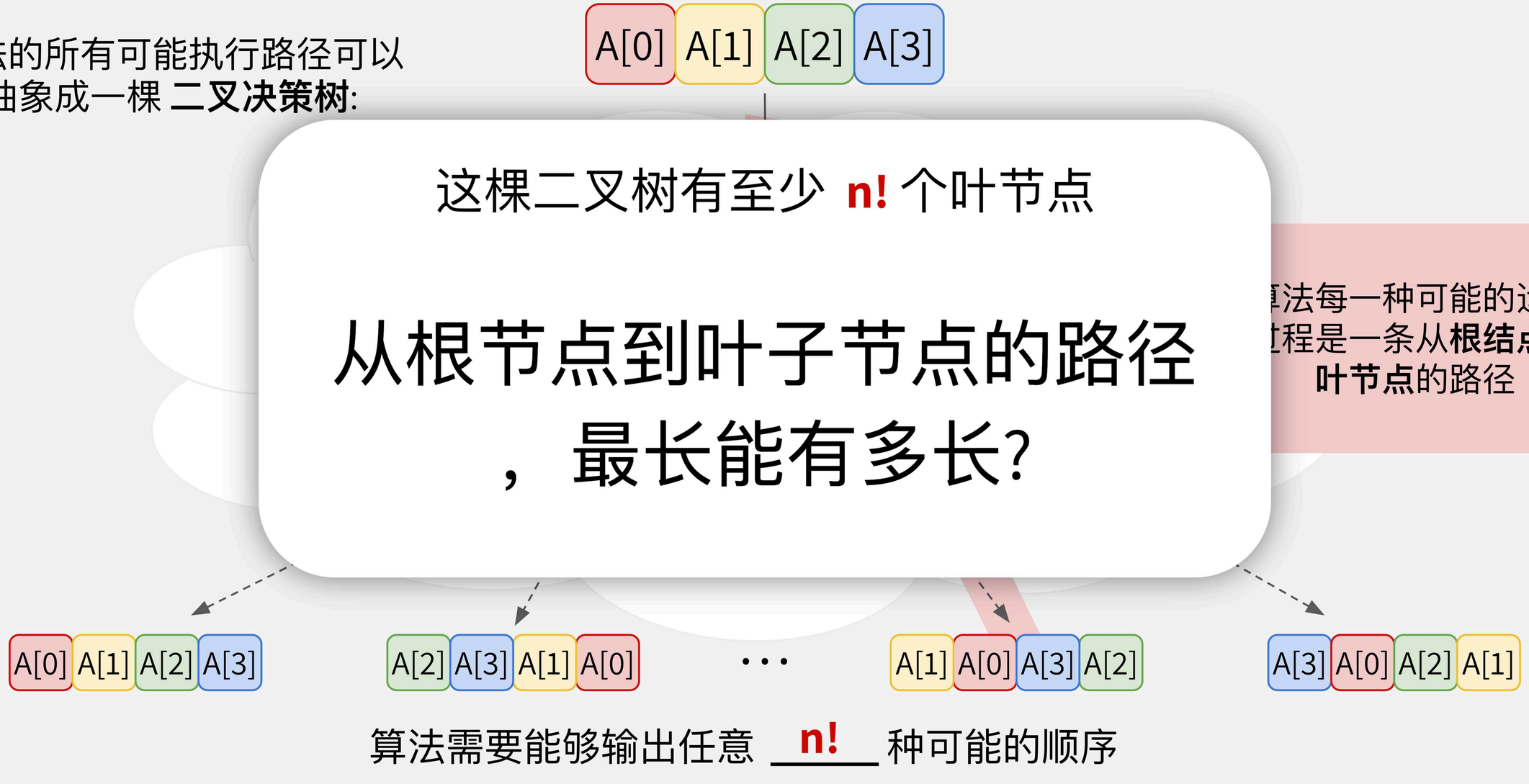
算法每一种可能的运行过程是一条从**根结点**到**叶节点**的路径

算法需要能够输出任意 **n!** 种可能的顺序

海量资源：666java.com

基于比较的排序

算法的所有可能执行路径可以抽象成一棵 **二叉决策树**:

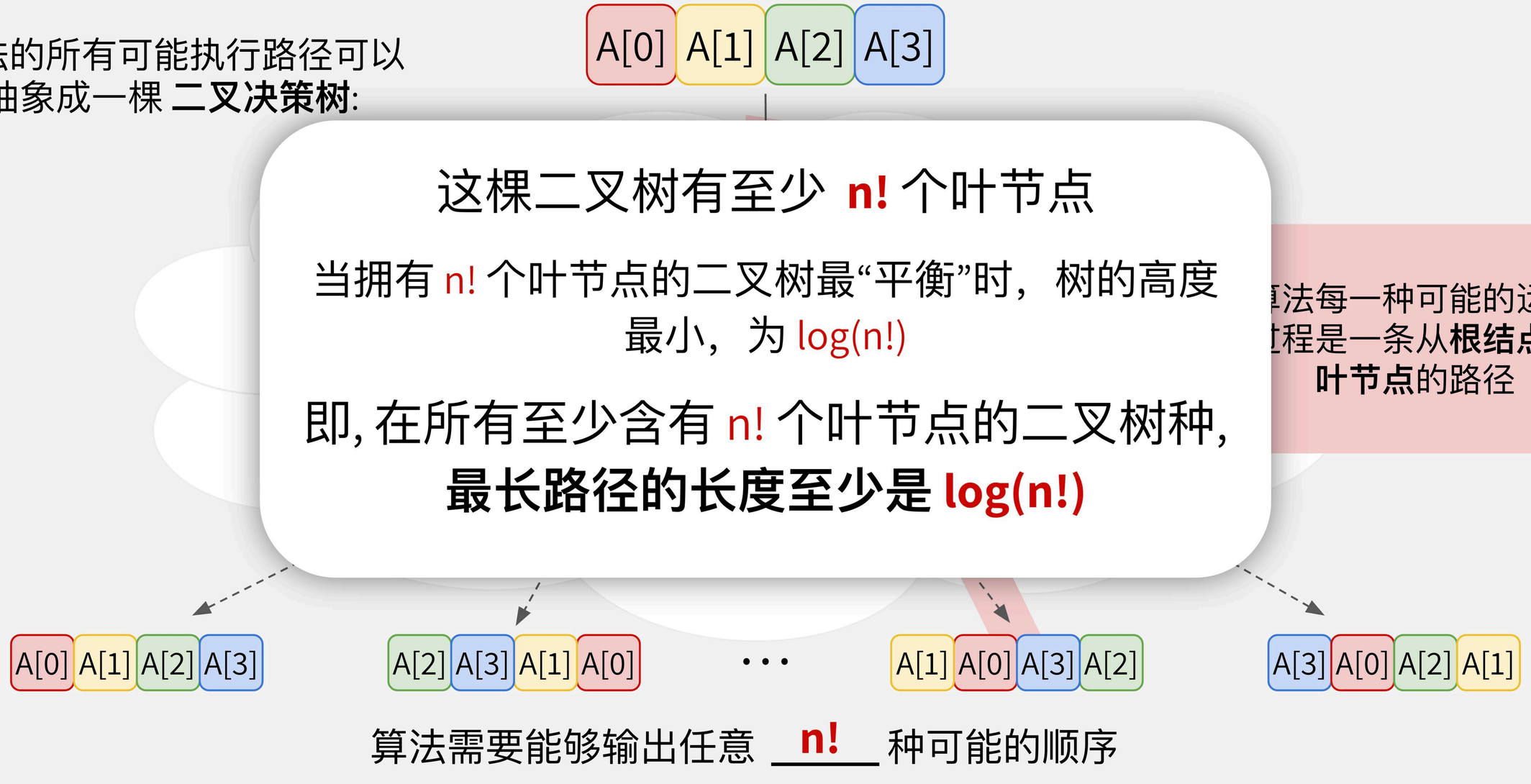


算法需要能够输出任意 **n!** 种可能的顺序

海量资源：666java.com

基于比较的排序

算法的所有可能执行路径可以抽象成一棵 **二叉决策树**:



算法需要能够输出任意 **n!** 种可能的顺序

海量资源：666java.com

基于比较的排序

最长路径的长度至少是 log(n!)

也就是说, 一个比较排序算法至少需要执行比较操作 log(n!) 次.

最坏情况的运行时间至少是 log(n!) = Ω(n log n).

海量资源：666java.com

基于比较的排序

最长路径的长度至少是 log(n!)

也就是说, 一个比较排序算法至少需要执行比较操作 log(n!) 次.

最坏情况的运行时间至少是 log(n!) = Ω(n log n).

$$
\begin{aligned}
\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\
&\geq \log\left(\frac{n}{2} + 1\right) + \log\left(\frac{n}{2} + 2\right) + \cdots + \log(n-1) + \log n \\
&\geq \left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) \\
&= \frac{1}{2} n \left(\log n - \log 2\right) \\
&= \Omega(n \log n)
\end{aligned}
$$

海量资源：666java.com

总结

定理:

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

- 任何比较算法都可以抽象成一棵含 n! 个叶节点的决策树
- 最坏情况的运行时间至少是决策树种最长路径的长度
- 包含 n! 个叶节点的二叉树的最长路径长度至少是 $\log(n!) = \Omega(n \log n)$
- 所以, 任何比较排序的最坏情况运行时间至少是 $\Omega(n \log n)$

海量资源：666java.com

总结

定理:

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

该边界也适用于随机化比较算法的期望运行时间! 证明略.

所以 归并排序 的速度是最块的!

海量资源：666java.com

总结

在最坏情况下，任 Ω(n log n) 次比较

该边 明略.

所以 块的!

问题来了...

还能更快吗?

**不使用意面排序的计算模型?*

海量资源：666java.com

线性时间排序

非比较排序算法

海量资源：666java.com

一个新的计算模型

元素本身的值是有意义的

海量资源：666java.com

一个新的计算模型

元素本身的值是有意义的

比较排序的计算模型中:

我们不关心元素本身值的大小所代表的意义
(我们只关心比较的结果)

海量资源：666java.com

一个新的计算模型

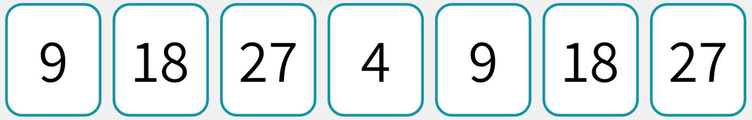
元素本身的值是有意义的

比较排序的计算模型中:

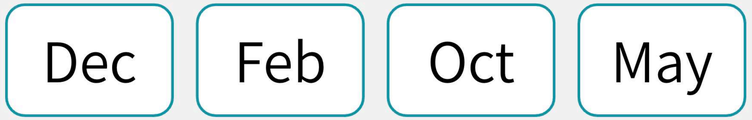
我们不关心元素本身值的大小所代表的意义
(我们只关心比较的结果)



新模型中:



一些不是很大的正整数



月份

海量资源：666java.com

计数排序 COUNTING SORT

假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:

30	50	20	30	10	60	50	20

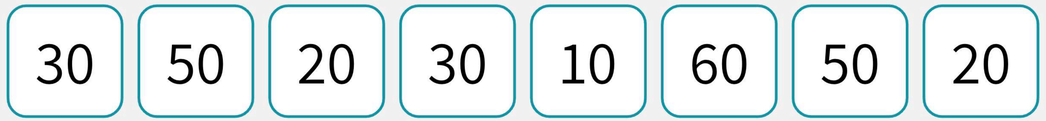
海量资源：666java.com

计数排序 COUNTING SORT

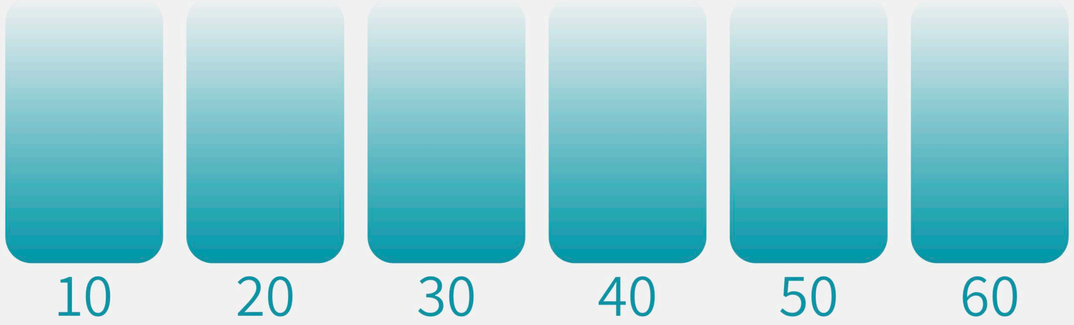
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



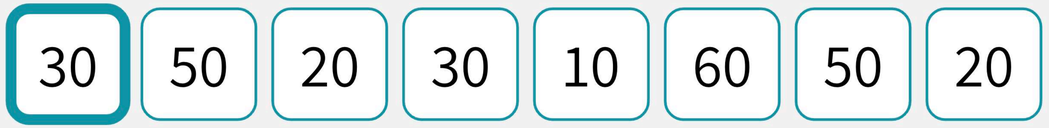
海量资源：666java.com

计数排序 COUNTING SORT

假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



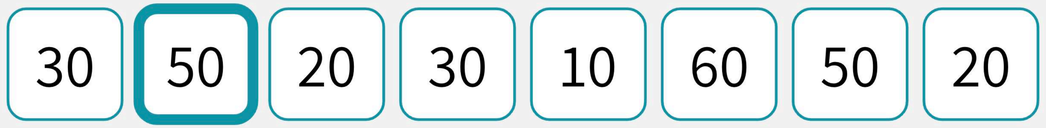
海量资源：666java.com

计数排序 COUNTING SORT

假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



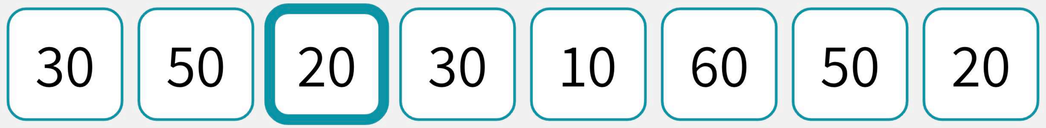
海量资源：666java.com

计数排序 COUNTING SORT

假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



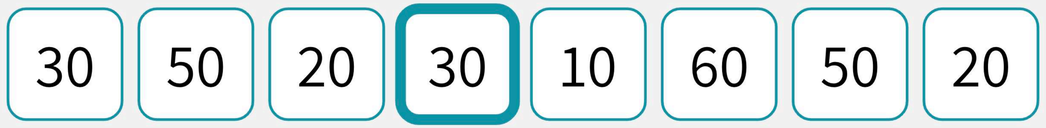
海量资源：666java.com

计数排序 COUNTING SORT

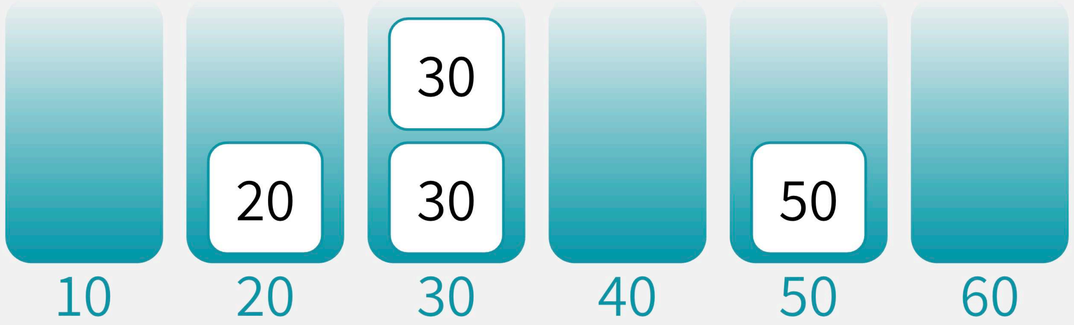
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



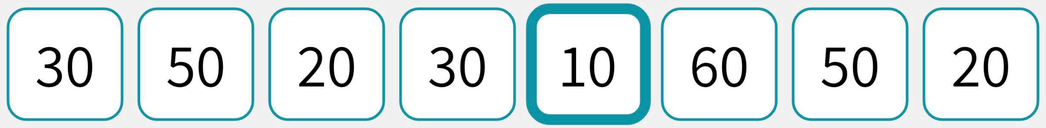
海量资源：666java.com

计数排序 COUNTING SORT

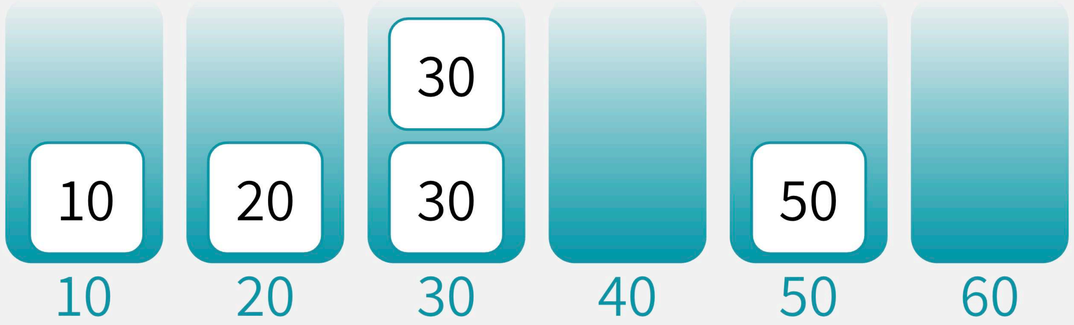
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



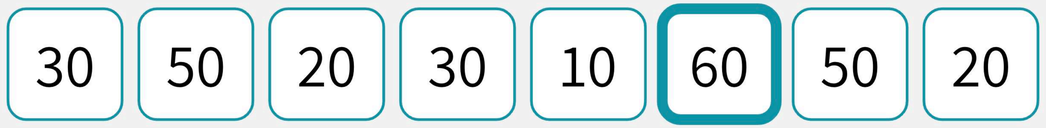
海量资源：666java.com

计数排序 COUNTING SORT

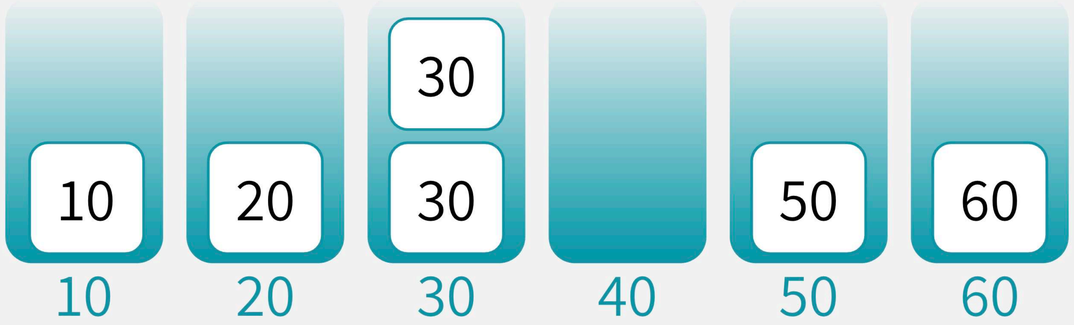
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



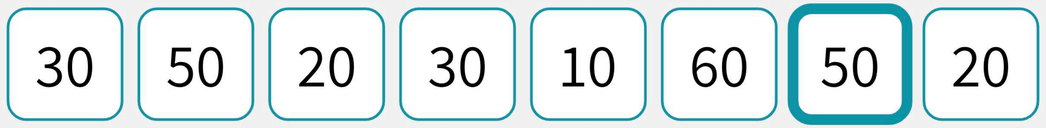
海量资源：666java.com

计数排序 COUNTING SORT

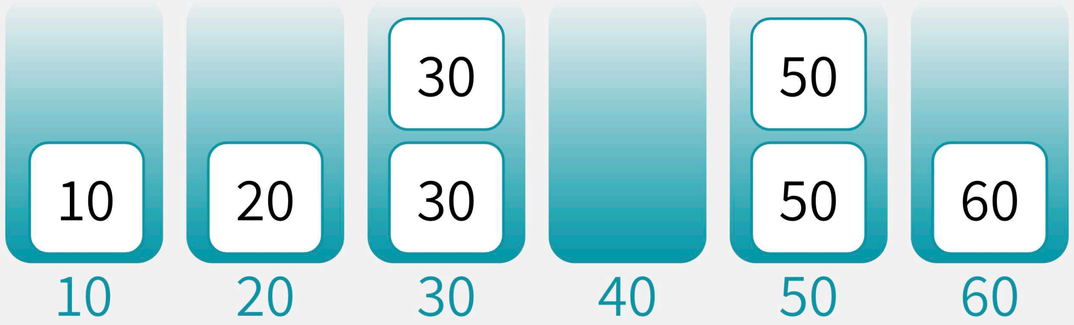
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



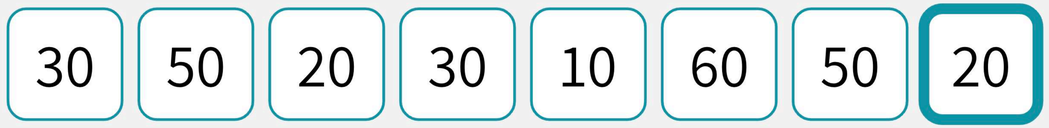
海量资源：666java.com

计数排序 COUNTING SORT

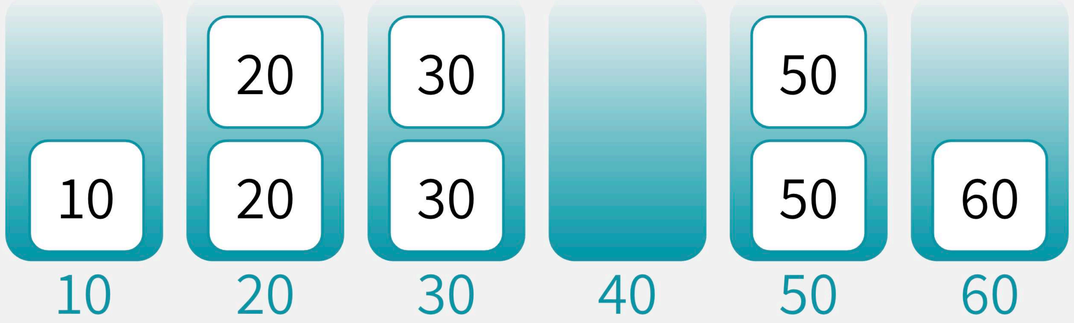
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



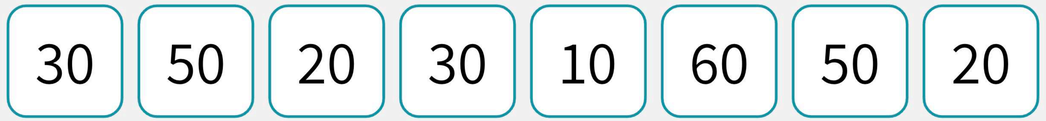
海量资源：666java.com

计数排序 COUNTING SORT

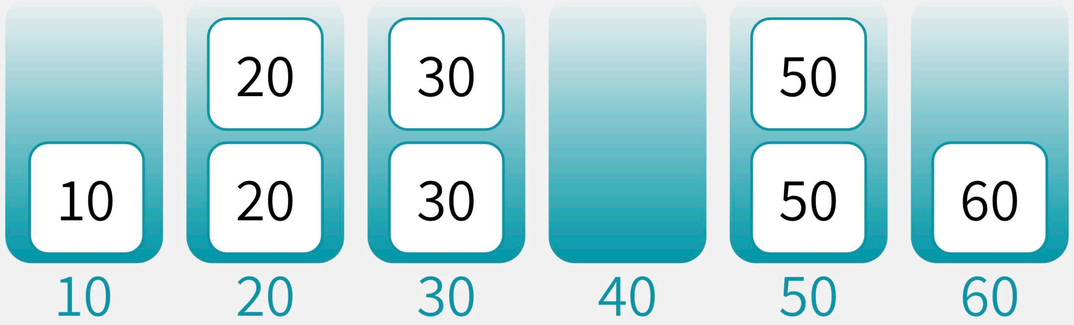
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:

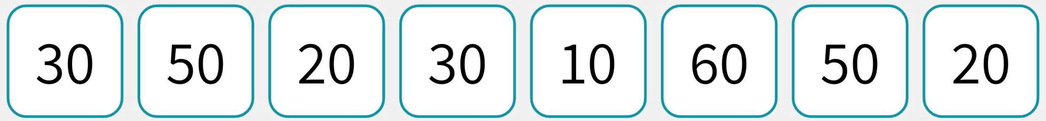
海量资源：666java.com

计数排序 COUNTING SORT

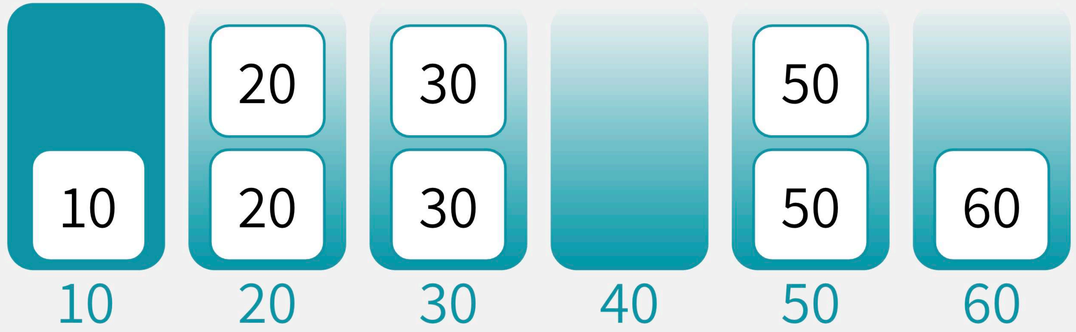
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:



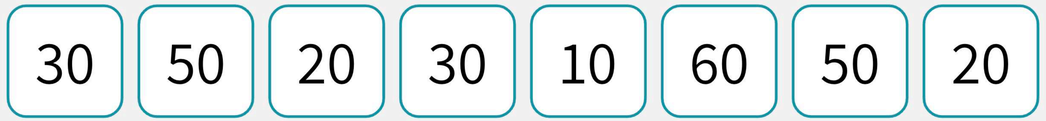
海量资源：666java.com

计数排序 COUNTING SORT

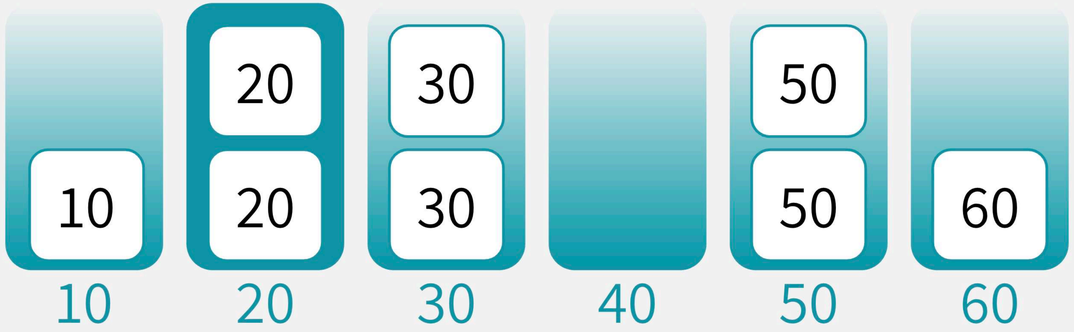
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:



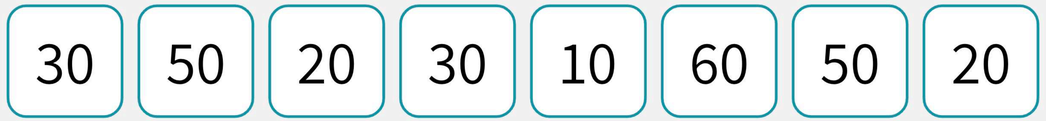
海量资源：666java.com

计数排序 COUNTING SORT

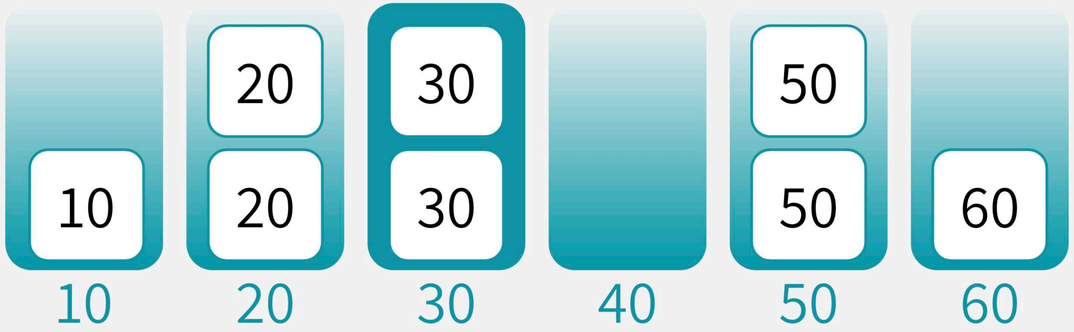
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:



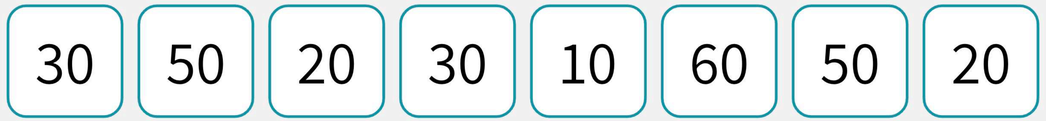
海量资源：666java.com

计数排序 COUNTING SORT

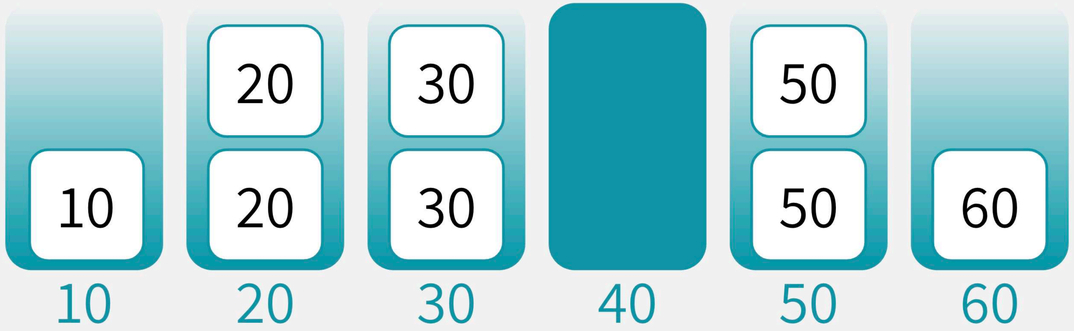
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:



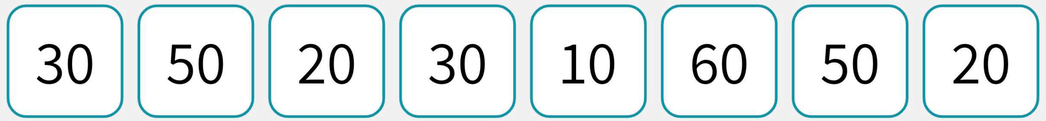
海量资源：666java.com

计数排序 COUNTING SORT

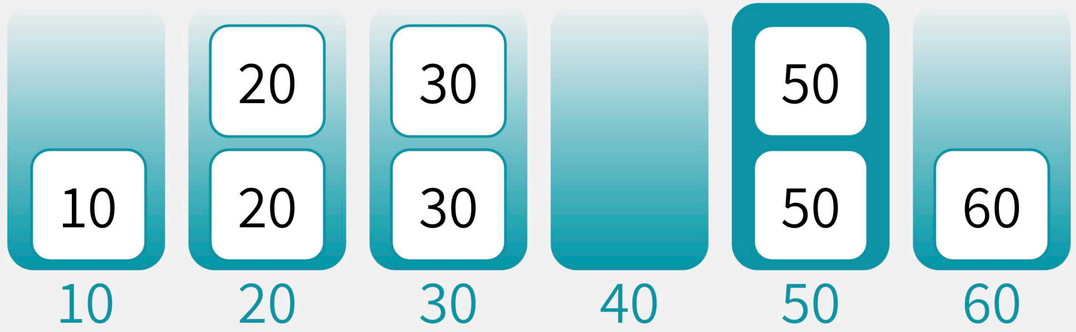
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



桶:



输出:



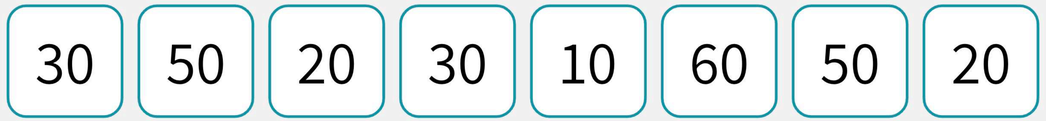
海量资源：666java.com

计数排序 COUNTING SORT

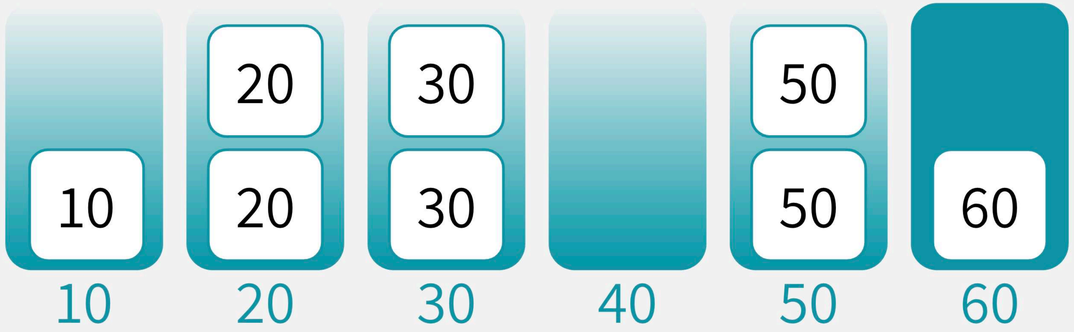
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

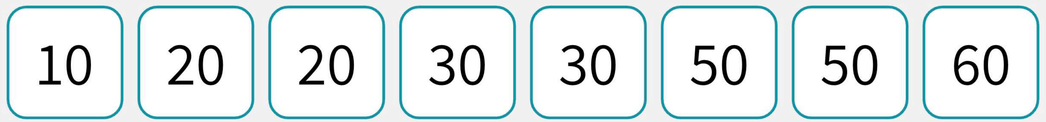
输入:



桶:



输出:



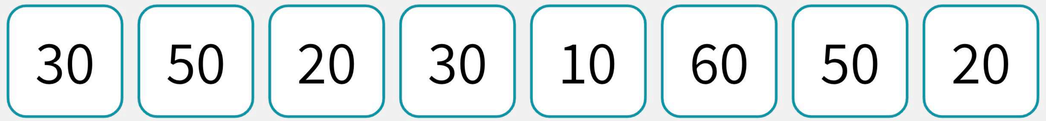
海量资源：666java.com

计数排序 COUNTING SORT

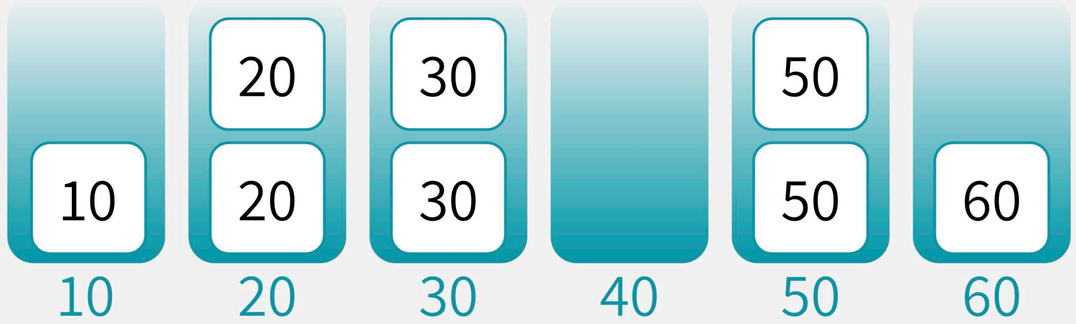
假设输入数组只可能包含 k 种可能的值 (预先知道这 k 种值)

例: 输入元素是 {10, 20, 30, 40, 50, 60} 中的值

输入:



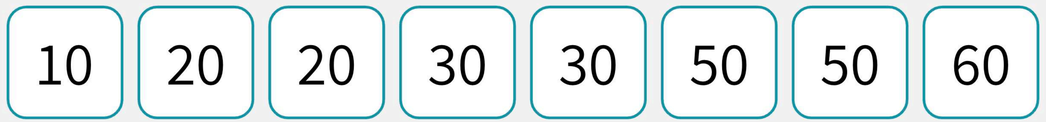
桶:



排序用时:

O(n)

输出:



海量资源：666java.com

计数排序

假设:

可以将元素按其值放入相对应的桶中.

提前知道可能出现的值.

输入的可能值数量有限.

如果可能出现的输入的值太多,
我们需要准备许多桶...
这会消耗大量的空间.

海量资源：666java.com

基数排序

适用于排序大小不超过 M 的整数

(或排序字符串)

海量资源：666java.com

基数排序 RADIX SORT

待排序整数大小不超过 M.
(可以推广到按字典序排序字符串)

思路:

使用计数排序按最低位(least-significant digit)进行排序,
使用计数排序按次低位进行排序, ...

无需为所有可能的值准备桶,
只需要准备每一位可能出现的值所对应的桶

如: 10 个标记为 0, 1, ..., 9 的桶

海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:

21 | 345 | 13 | 101 | 50 | 234 | 1

桶:



海量资源：666java.com

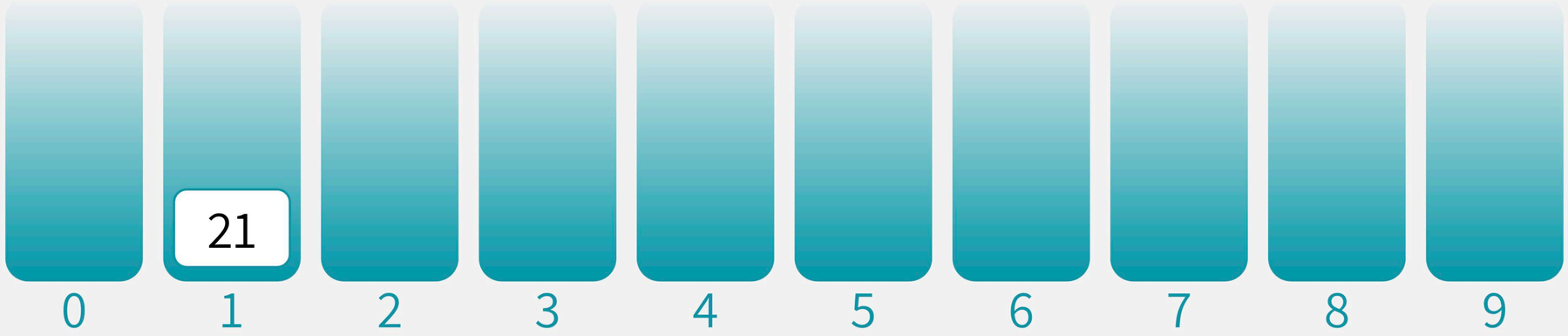
基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



海量资源：666java.com

基数排序 RADIX SORT

第 1 步: 使用计数排序按最低位进行排序

输入:



桶:



输出:



输出时, 确保桶中元素按照先进先出 (FIFO) 的顺序
(计数排序是稳定的，实现时每个桶是一个队列)

海量资源：666java.com

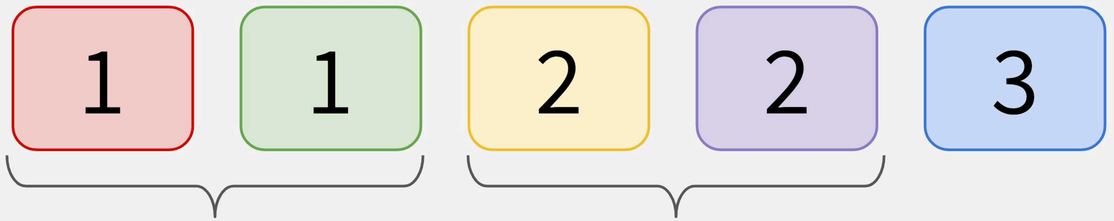
题外话: 排序的稳定性

一个排序算法是**稳定的**当且仅当大小相同的元素排序后依然保持排序前的相对位置

输入:



排序后的输出:
(排序是稳定的)



输入中红1在绿1前, 输出中依然保持这个顺序

输入中黄2在紫2前, 输出中依然保持这个顺序

海量资源：666java.com

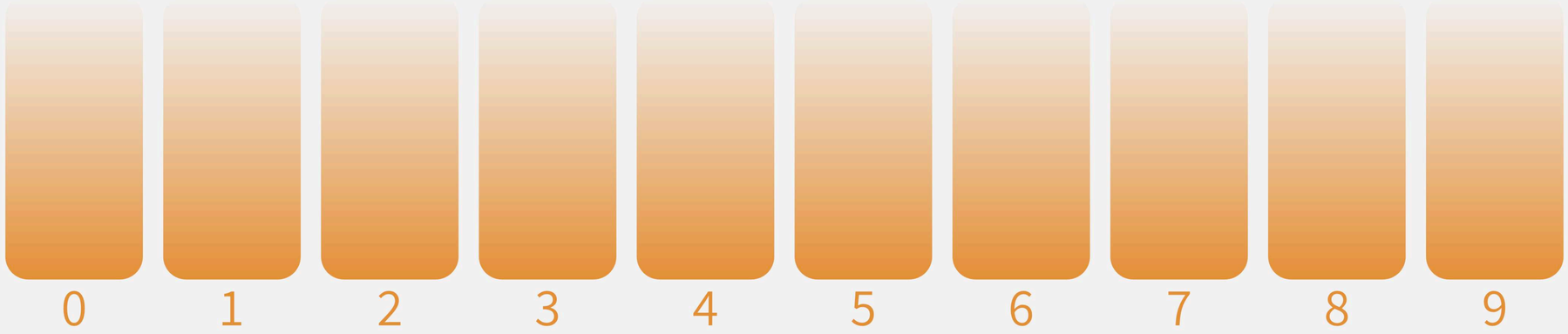
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

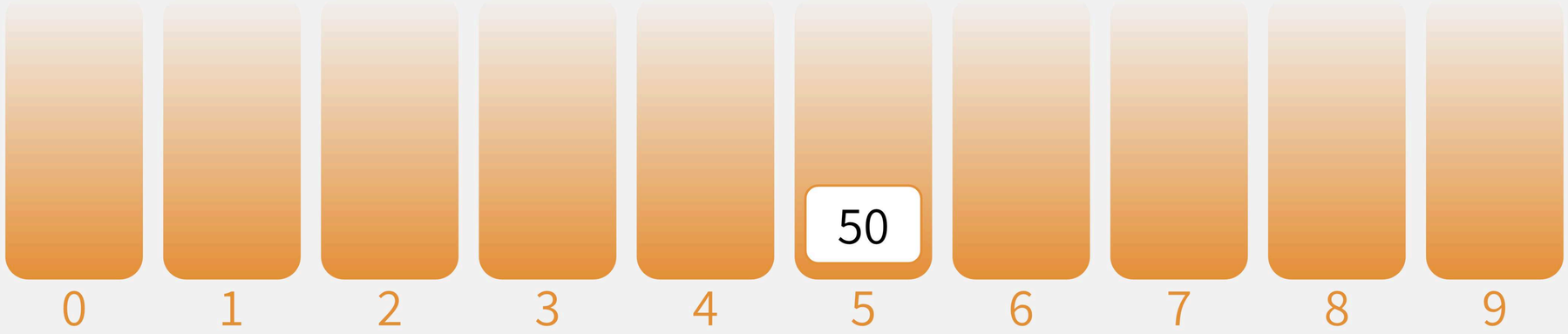
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

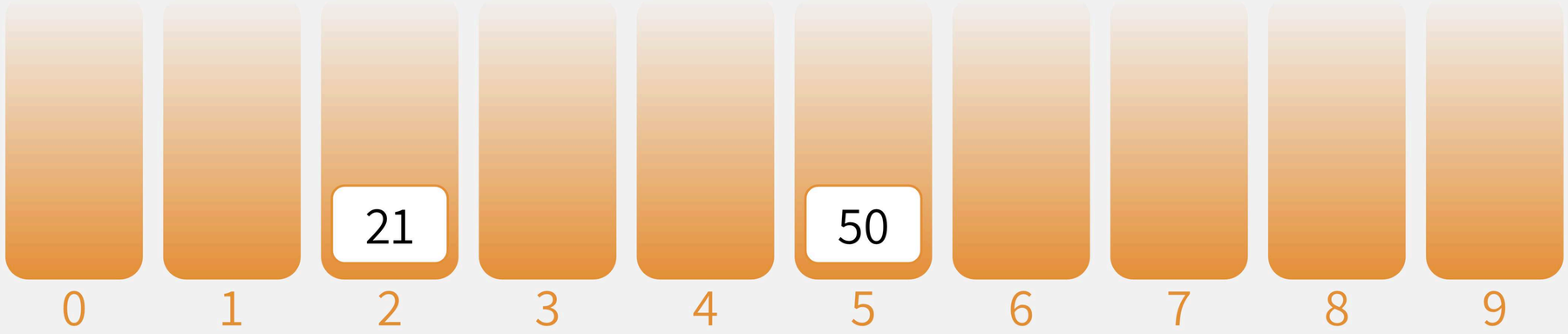
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

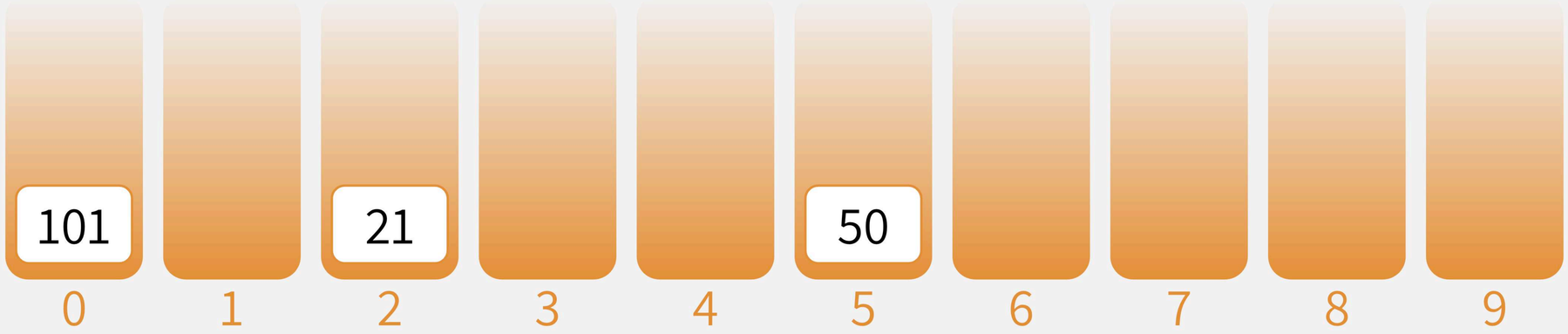
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

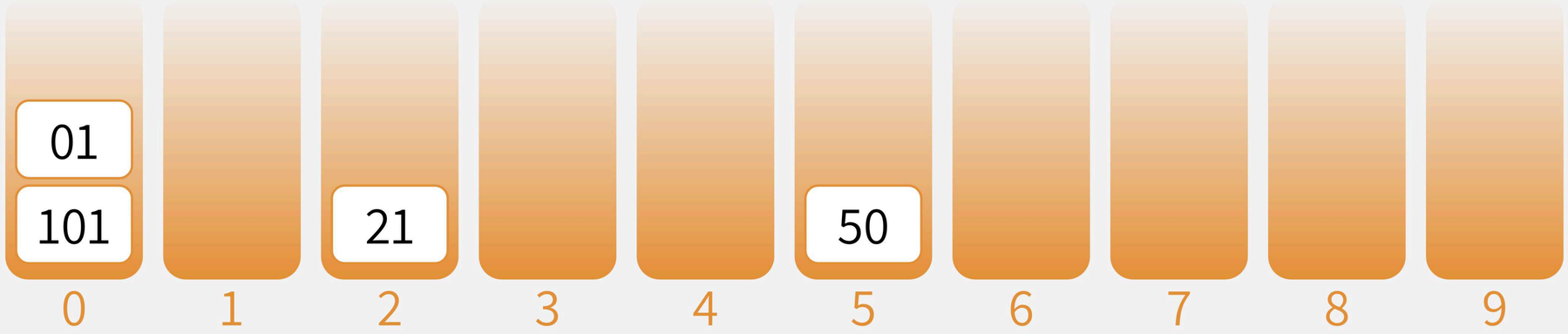
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

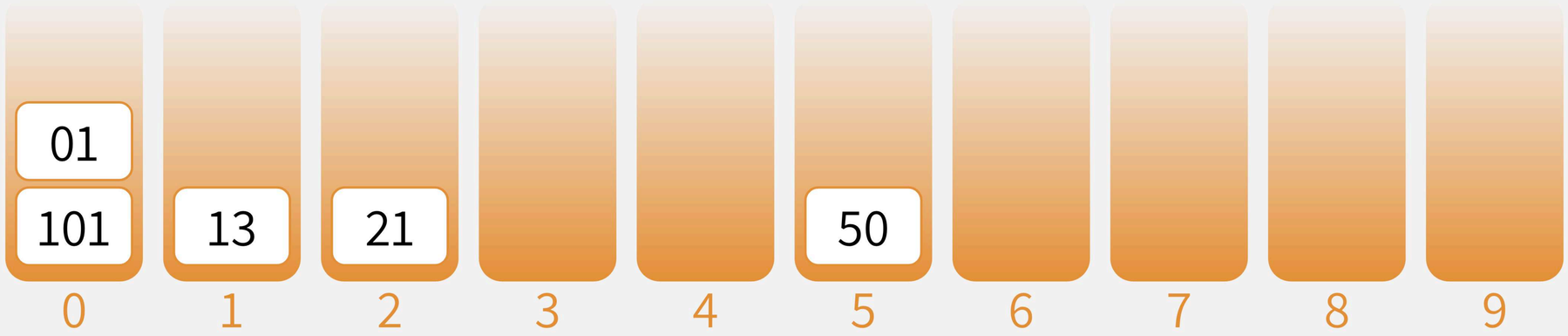
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

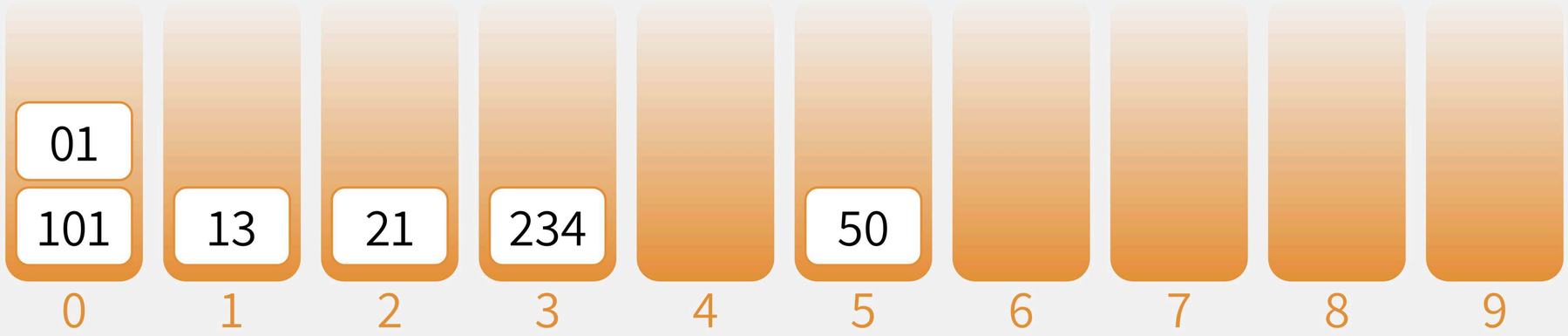
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)



桶:



海量资源：666java.com

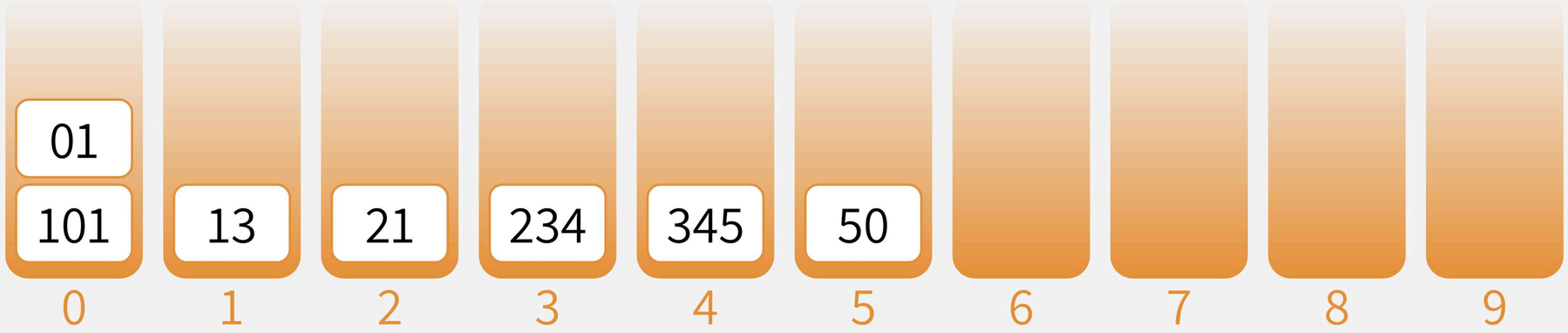
基数排序 RADIX SORT

第 2 步: 使用计数排序按倒数第 2 位进行排序

输入:
(第 1 步的输出)

50 21 101 01 13 234 345

桶:



输出:

101 01 13 21 234 345 50

输出时, 确保桶中元素按照先进先出 (FIFO) 的顺序
(计数排序是稳定的，实现时每个桶是一个队列)

海量资源：666java.com

基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)

101 | 01 | 13 | 21 | 234 | 345 | 50

桶:



海量资源：666java.com

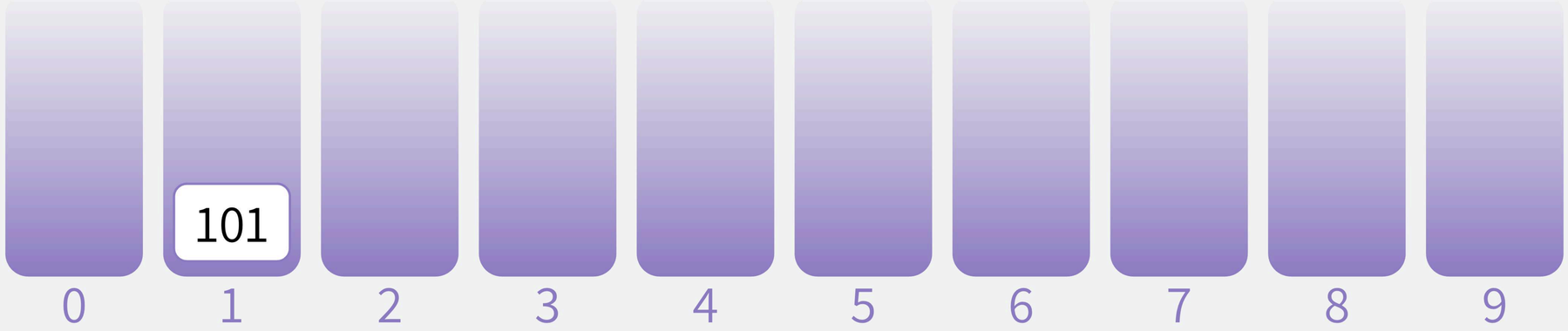
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)

101 | 01 | 13 | 21 | 234 | 345 | 50

桶:



海量资源：666java.com

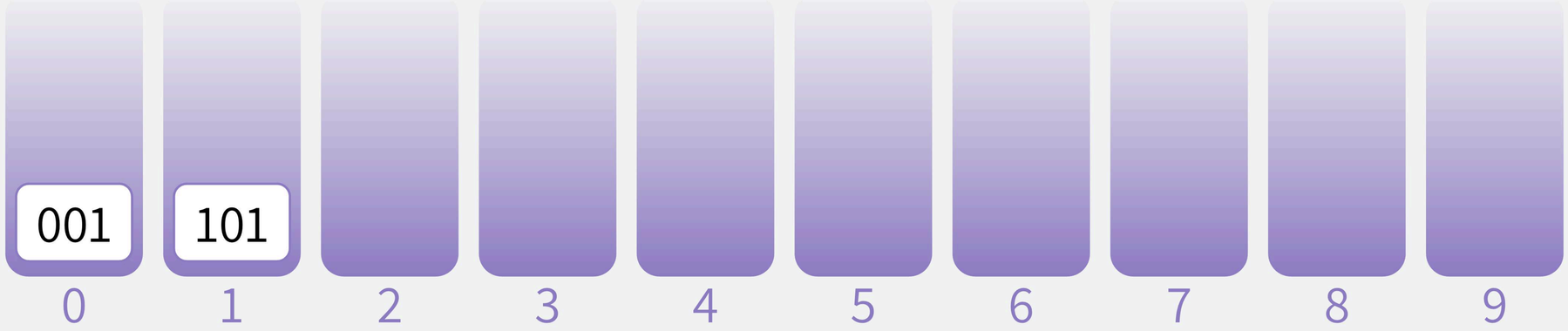
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)

101 001 13 21 234 345 50

桶:



海量资源：666java.com

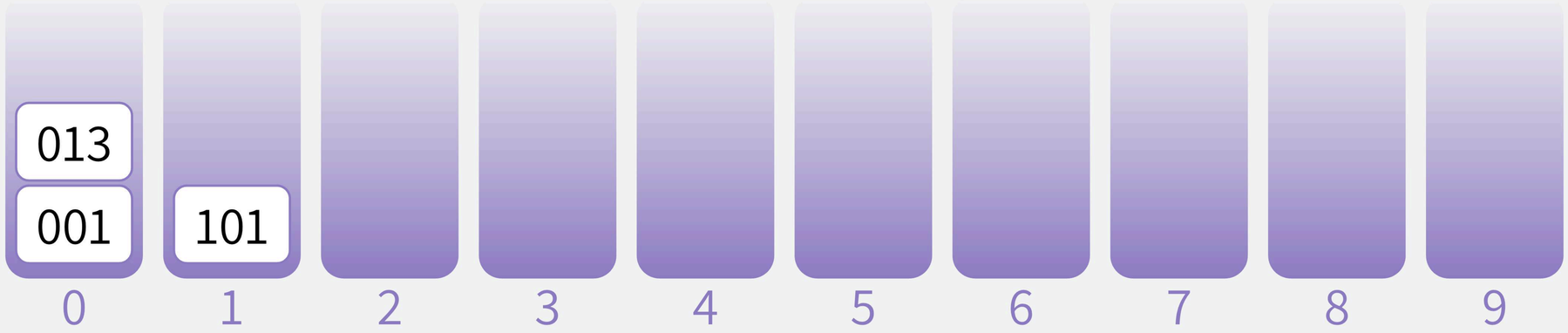
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)



桶:



海量资源：666java.com

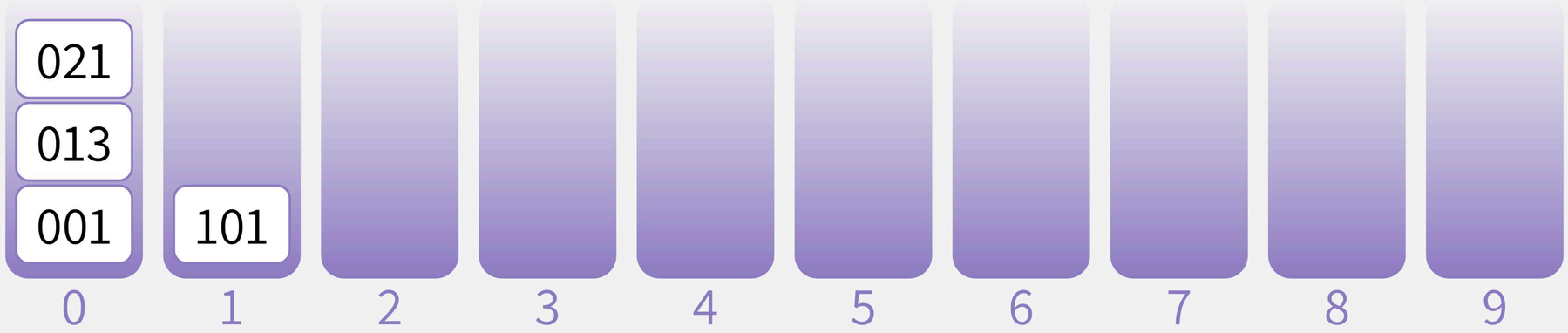
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)



桶:



海量资源：666java.com

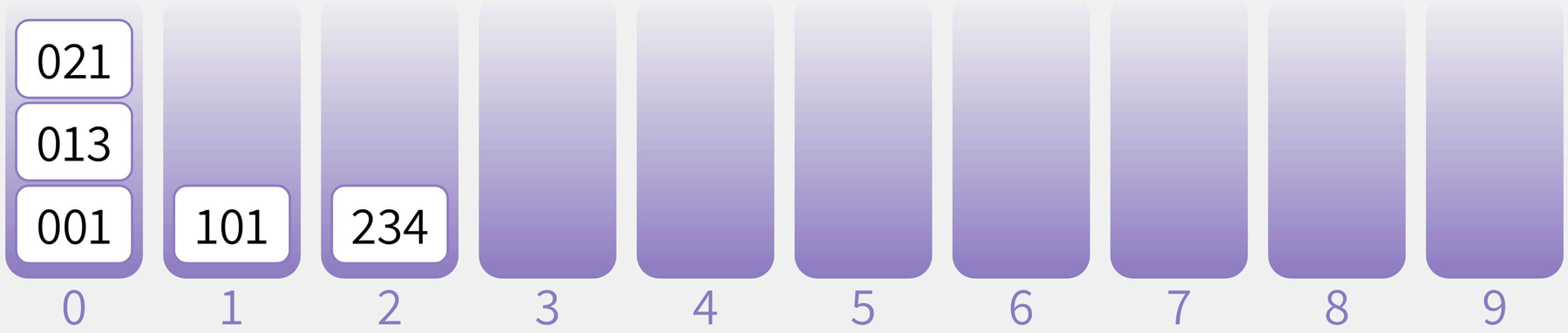
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)



桶:



海量资源：666java.com

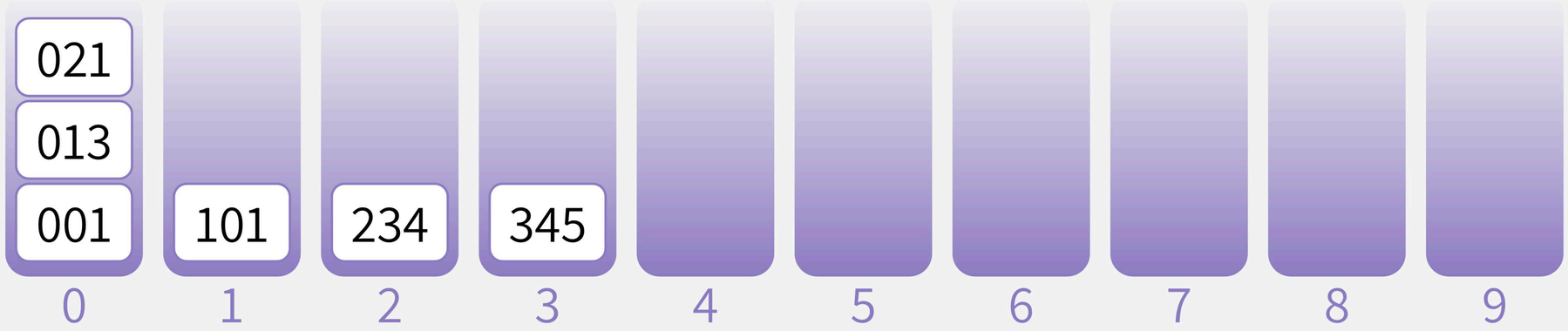
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)

101 001 013 021 234 345 50

桶:



海量资源：666java.com

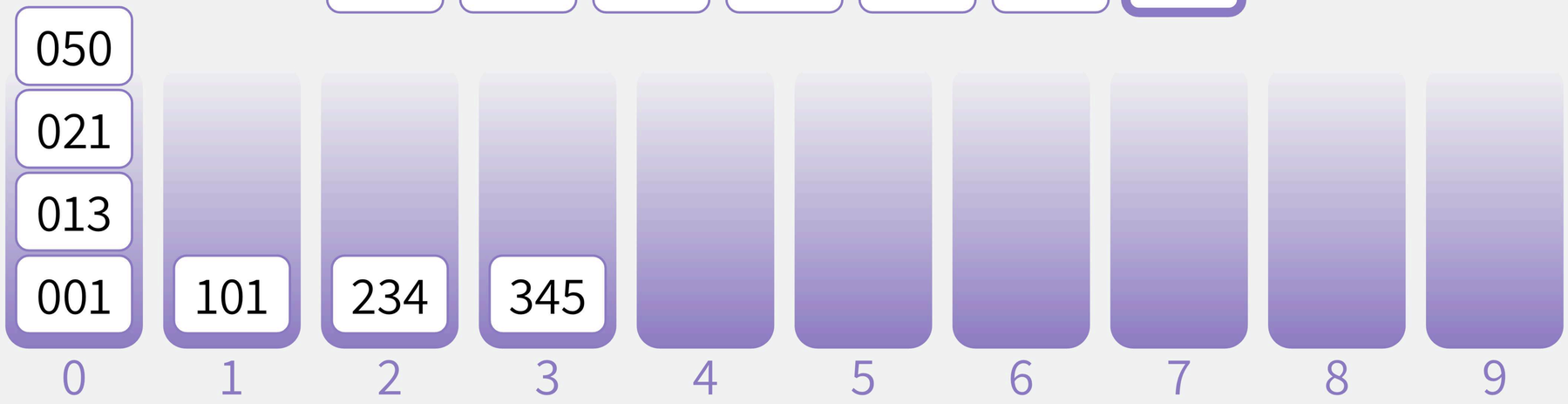
基数排序 RADIX SORT

第 3 步: 使用计数排序按倒数第 3 位进行排序

输入:
(第 2 步的输出)

101 001 013 021 234 345 050

桶:



输出:

001 013 021 050 101 234 345

成功了! 但是为什么呢???

海量资源：666java.com

基数排序的正确性

海量资源：666java.com

为什么基数排序可行?

输入: 21 345 13 101 50 234 1

海量资源：666java.com

为什么基数排序可行?

输入: 21 345 13 101 50 234 1

第 1 轮排序后元素按最末位排序

50 2**1** 10**1** **1** 1**3** 23**4** 34**5**

海量资源：666java.com

为什么基数排序可行?

输入: 21 345 13 101 50 234 1

第 1 轮排序后元素按最末位排序

50 21 101 1 13 234 345

第 2 轮排序后元素按末 2 位排序

101 01 13 21 234 345 50

海量资源：666java.com

为什么基数排序可行?

输入: 21 345 13 101 50 234 1

第 1 轮排序后元素按最末位排序

50 21 101 1 13 234 345

第 2 轮排序后元素按末 2 位排序

101 01 13 21 234 345 50

第 3 轮排序后元素按末 3 位排序

001 013 021 050 101 234 345

海量资源：666java.com

基数排序的运行时间

海量资源：666java.com

基数排序运行时间

假设我们要排序 **n** 个(不超过)**d** 位 的 10 进制数 (例如 n = 7, d = 3):

21	345	13	101	50	234	1

一共会有多少次迭代?

每次迭代耗时多少?

总共的运行时间为?

海量资源：666java.com

基数排序运行时间

假设我们要排序 **n** 个(不超过)**d** 位 的 10 进制数 (例如 n = 7, d = 3):

21	345	13	101	50	234	1

一共会有多少次迭代?

d 次迭代

每次迭代耗时多少?

总共的运行时间为?

海量资源：666java.com

基数排序运行时间

假设我们要排序 **n** 个(不超过)**d** 位 的 10 进制数 (例如 n = 7, d = 3):

21	345	13	101	50	234	1

一共会有多少次迭代?

d 次迭代

每次迭代耗时多少?

初始化 10 个桶 + 将 n 个数放入 10 个桶中 ⇒ **O(n)**

总共的运行时间为?

海量资源：666java.com

基数排序运行时间

假设我们要排序 **n** 个(不超过)**d** 位 的 10 进制数 (例如 n = 7, d = 3):

21	345	13	101	50	234	1

一共会有多少次迭代?

d 次迭代

每次迭代耗时多少?

初始化 10 个桶 + 将 n 个数放入 10 个桶中 ⇒ **O(n)**

总共的运行时间为?

O(nd)

海量资源：666java.com

如何评价 O(nd)?

排序 n 个 10 进制，范围在 {1,2, ...,M} 中的数时, O(nd) 并不是很好:

一共会有多少次迭代?

每次迭代耗时多少?

总共的运行时间为?

海量资源：666java.com

如何评价 O(nd)?

排序 n 个 10 进制，范围在 {1,2, ...,M} 中的数时, O(nd) 并不是很好:

当 M = 1234 时:

$\lfloor \log_{10} 1234 \rfloor + 1$
$= 3 + 1 = 4$

一共会有多少次迭代?

d = **$\lfloor \log_{10} M \rfloor + 1$** 次迭代

每次迭代耗时多少?

初始化 10 个桶 + 将 n 个数放入 10 个桶中 ⇒ **O(n)**

总共的运行时间为?

O(nd) = **O(n log M)**

近似的结果 (向下取整并 +1)

海量资源：666java.com

如何评价 O(nd)?

排序 n 个 10 进制，范围在 {1,2, ...,M} 中的数时, O(nd) 并不是很好:

当 M = 1234 时:

$\lfloor \log_{10} 1234 \rfloor + 1$

$= 3 + 1 = 4$

一共会有多少次迭代?

d = **$\lfloor \log_{10} M \rfloor + 1$** 次迭代

每次迭代耗时多少?

初始化 10 个桶 + 将 n 个数放入 10 个桶中 ⇒ **O(n)**

总共的运行时间为?

O(nd) = **O(n log M)**

近似的结果 (向下取整并 +1)

当 M 接近 n 或更大时, 这并没有比归并排序更快

海量资源：666java.com

如何评价 O(nd)?

排序 n 个 10 进制，范围在 [ ] 的数时，O(nd) 并不是很好:

当 M = 1234 时:

$\lfloor \log_{10} 1234 \rfloor + 1$

$= 3 + 1 = 4$

初始化 10 [ ] 中 ⇒ **O(n)**

O(nd) **(n log M)**

近似的结果 (向下取整并 +1)

问题来了...

如何才能更快?

当 M 接近 n 或更大时，这并没有比归并排序更快

海量资源：666java.com

使用不同的基数

使用 10 作为基数时性能不好... 基数如何影响算法的运行时间?

海量资源：666java.com

使用不同的基数

使用 10 作为基数时性能不好… 基数如何影响算法的运行时间?

记基数为 **r**

一共会有多少次迭代?

d = $\lfloor \log_r M \rfloor + 1$ 次迭代

每次迭代耗时多少?

初始化 **r** 个桶 + 将 n 个数放入 **r** 个桶中 ⇒ $O(n + r)$

总共的运行时间为?

$O(d \cdot (n+r)) = O((\lfloor \log_r M \rfloor + 1) \cdot (n + r))$

海量资源：666java.com

使用不同的基数

使用 10 作为基数时性能不好… 基数如何影响算法的运行时间?

记基数为 **r**

一共会有多少次迭代?

$d = \lfloor \log_r M \rfloor + 1$ 次迭代

每次迭代耗时多少?

初始化 **r** 个桶 + 将 n 个数放入 **r** 个桶中 ⇒ $O(n + r)$

总共的运行时间为?

$O(d \cdot (n+r)) = O((\lfloor \log_r M \rfloor + 1) \cdot (n + r))$

更大的基数 r ⇒ 更少次迭代, 但是更多个桶需要初始化!

海量资源：666java.com

使用不同的基数

一个比较好的选择: **取 r = n**

海量资源：666java.com

使用不同的基数

一个比较好的选择: **取 r = n**

一共会有多少次迭代?

d = **$\lfloor \log_n M \rfloor + 1$** 次迭代

每次迭代耗时多少?

初始化 **n** 个桶 + 将 n 个数放入 **n** 个桶中 ⇒ **O(n+n) = O(n)**

总共的运行时间为?

$O(d \cdot n) = O((\lfloor \log_n M \rfloor + 1) \cdot n)$

海量资源：666java.com

使用不同的基数

一个比较好的选择: **取 r = n**

一共会有多少次迭代?

$d = \lfloor \log_n M \rfloor + 1$ 次迭代

每次迭代耗时多少?

初始化 **n** 个桶 + 将 n 个数放入 **n** 个桶中 ⇒ $O(n+n) = O(n)$

总共的运行时间为?

$O(d \cdot n) = O((\lfloor \log_n M \rfloor + 1) \cdot n)$

这一项是常数

若 $M \leq n^c$ (c 是常数), 则 $O((\lfloor \log_n M \rfloor + 1) \cdot n) = O(n)$

海量资源：666java.com

使用不同的基数

一个比较好的选择: **取 r = n**

使用基数 **r = n** 时 (而不是简单取 10) 基数排序的运行时间取决于 M 比 n 大多少. 运行时间为:

$$O((\lfloor \log_n M \rfloor + 1) \cdot n)$$

当 $M \le n^c$ 上式为 O(n)

所需桶的个数为 r = n

若 $M \le n^c$ (c 是常数), 则 $O((\lfloor \log_n M \rfloor + 1) \cdot n) = O(n)$

海量资源：666java.com

基数排序总结

基数排序可以在 **O(n)** 时间内排序 **n 个大小不超过 n^{100}** (或 n^c, c 为任意常数) **的整数**.

海量资源：666java.com

基数排序总结

基数排序可以在 **O(n)** 时间内排序 **n 个大小不超过 n^{100}** (或 n^c, c 为任意常数) **的整数**.

当需要排序的数远大于 n (或 n^c), 例如 2^n, 基数排序无法在线性时间内完成.

海量资源：666java.com

基数排序总结

基数排序可以在 **O(n)** 时间内排序 **n 个大小不超过 n^{100}** (或 n^C, c 为任意常数) **的整数**.

当需要排序的数远大于 n (或 n^C), 例如 2^n, 基数排序无法在线性时间内完成.

运行时间取决于所选取的基数
记 **n** 为输入元素个数, **M** = 输入元素可能的最大值, **r** 为基数:

基数排序的运行时间 = $O((\lfloor \log_r M \rfloor + 1) \cdot n)$

海量资源：666java.com

为什么需要比较排序?

比较排序适用于任何可以比较的类型
排序数值时, 比较排序可以处理高精度和极大的值:

π	**1234/9876**	**e**	**43!**	**4.10598425**	**n^n**	**31**

基数排序需要关注每一位的大小 — π 和 e 均有无穷多位、 n^n 非常大...

基数排序需要额外空间, 因为我们需要生成对应每个基数的桶.

海量资源：666java.com

总结

- 任意基于比较的排序算法的运行时间至少为 **Ω(n log n)**
- 采用不同的计算模型可以实现线性时间内的排序
  - **计数排序**: 原理非常简单, 但仅适用于输入可能值很少的情况 (否则需要大量的桶)
  - **基数排序**: 按位进行计数排序, 当输入元素最大值有限时能在线性时间内完成

海量资源：666java.com

接下来

- 树 (二叉搜索树 & 平衡树)