算法分析与设计

第2章 排序

主讲人：甘文生 PhD

Email: wsgan001@gmail.com

暨南大学网络空间安全学院

Fall 2021

Jinan University, China

排序问题

内容提要：

- 排序问题
- 冒泡排序 & 选择排序
- 插入排序 & 希尔排序
- 快速排序 & 堆排序 《算法导论》第7章 & 第6章
- 线性排序算法 (计数排序、桶排序、基数排序) 《算法导论》第8章
- 排序算法比较

回顾: 排序算法

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

排序问题

排序 (Sorting)：将一串数据依照指定方式进行排列。

常用排序方式：数值顺序，字典顺序。

算法评价重要指标

时间复杂度（最差、平均）

设有 n 个数据，一般来说，好的排序算法性能为 $O(n \log n)$，差的性能为 $O(n^2)$，而理想的性能为 $O(n)$。

空间复杂度

算法在运行过程中临时占用的存储空间的大小。

稳定排序算法：相同的数据，排序后仍维持原有的相对次序。

排序问题

- **问题描述：**

  输入：n个数的序列 $<a_1, a_2, ..., a_n>$

  输出：输入序列的一个重排 $<a_1', a_2', ..., a_n'>$，使得 $a_1' \le a_2' \le \cdots \le a_n'$

- 输入数据的结构可以各式各样，比如n元数组、链表等；

- 排序问题是计算机科学领域中的最基本问题：

  ① 有些应用程序本身需要对信息进行排序；
  ② 应用广泛，是许多算法的关键步骤；
  ③ 已有很多成熟算法，它们采用各种技术，具有历史意义；
  ④ 可以证明其非平凡下界，是渐近最优的；
  ⑤ 可通过排序过程的下界来证明其他问题的下界；
  ⑥ 在实现过程中，经常伴随着许多工程问题（主机存储器层次结构、软件环境等）

排序问题

- 当排序记录的关键字均不相同，排序结果惟一，否则排序结果不唯一。
- **排序的稳定性：**

  ① 在待排序的文件中，若存在多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是**稳定的**；

  ② 若具有相同关键字的记录之间的相对次序发生变化，则称排序方法**不稳定**。
- 排序算法的稳定性针对所有输入实例而言，即在所有可能的输入实例中，只要有一个实例使得算法不满足稳定性要求，则该排序算法不稳定。

冒泡排序&选择排序

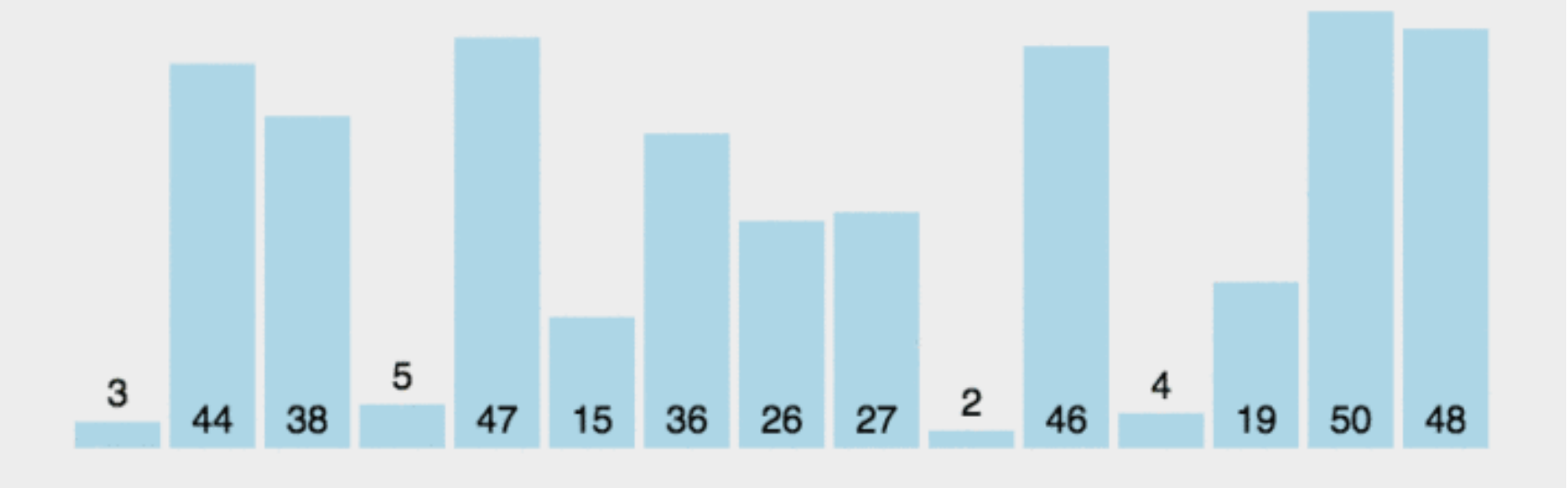
内容提要：

- 排序问题
- 冒泡排序 & 选择排序
- 插入排序 & 希尔排序
- 快速排序 & 堆排序
- 线性排序算法 (计数排序、桶排序、基数排序)
- 排序算法比较

冒泡排序

- **冒泡排序（Bubble Sort）**的工作原理：重复地走访待排序的数列，一次比较两个元素，如果它们的顺序错误就交换位置。走访数列重复地进行，直到没有元素再需要交换，则该数列已排序完成。越小的元素由交换慢慢“浮”到数列的顶端。
- 基本思想：
  1. 比较相邻的两个元素。如果第一个元素比第二个大，就交换它们；
  2. 对每一对相邻元素做同样的操作，从开始第一对到结尾的最后一对，则最后的元素是最大的数；
  3. 针对所有的元素重复以上的步骤，除了最后一个元素；
  4. 重复步骤1~3，直到排序完成。

冒泡排序的动图演示

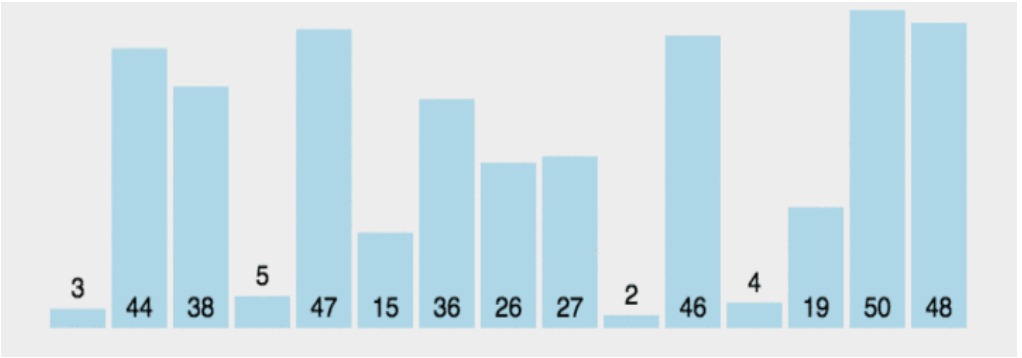


冒泡排序的算法分析

```
function bubbleSort(arr) {
    var len = arr.length;

    for (var i = 0; i < len - 1; i++) {
        for (var j = 0; j < len - 1 - i; j++) {

            if (arr[j] > arr[j+1]) { // 相邻元素两两比较
                var temp = arr[j+1];  // 元素交换
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    return arr;
}
```



算法分析：冒泡排序是最简单而直接的算法，其稳定性好，但是效率非常低，最差时间复杂度为$O(n^2)$，当问题的规模n很大时，不建议使用冒泡排序算法。

思考：冒泡排序

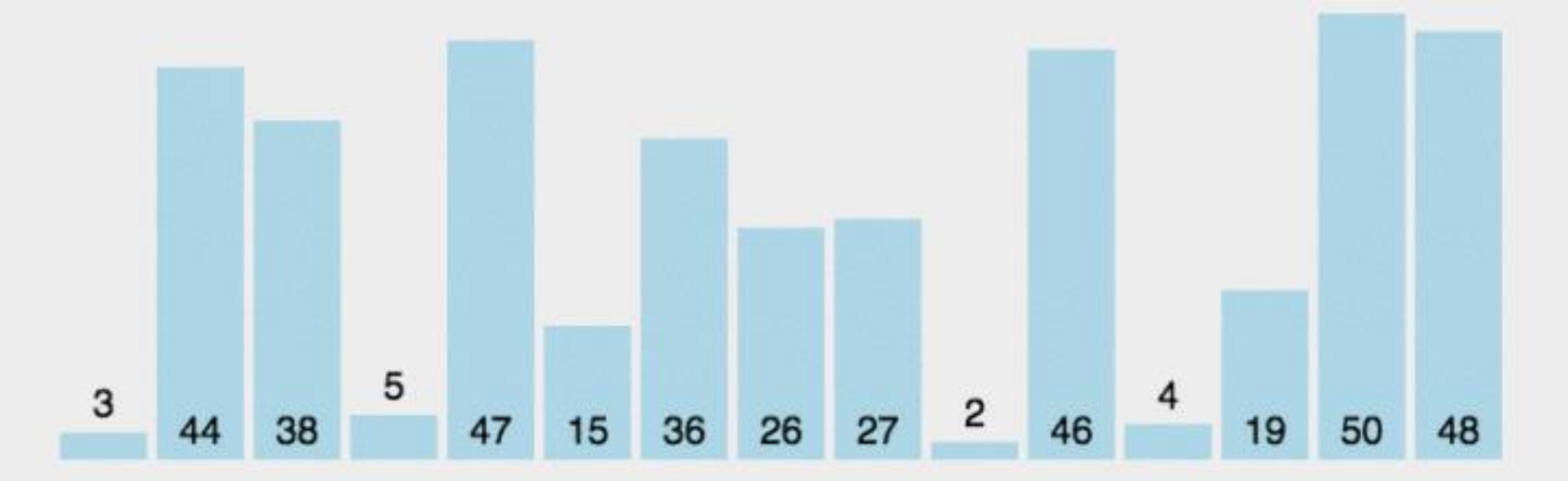
冒泡排序的优化：如果在某轮冒泡过程中没有发生元素交换，则说明整个序列已经排好序，此时不需要再进行后续的冒泡操作，可以直接结束程序。

冒泡排序的进一步优化：假设有100个元素的数组，仅前面10个无序，后面90个都已排好序且都大于前面10个元素，那么在第一轮冒泡过程后，最后发生交换的位置必定小于10，且该位置之后的元素必定已经有序。只需记录该位置，第二次从数组头部遍历到这个位置即可。

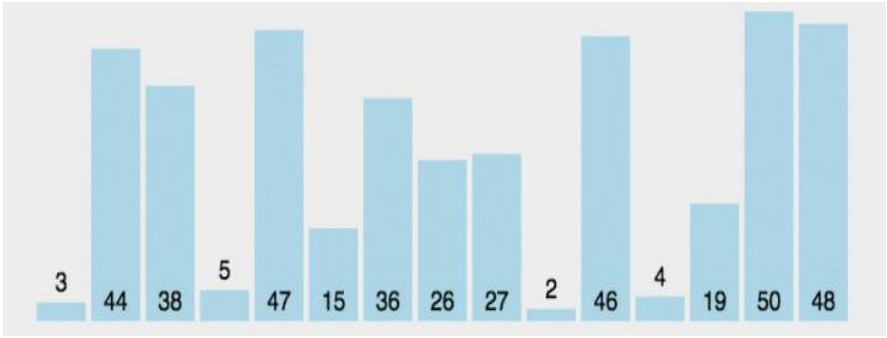
选择排序

- 选择排序(Selection Sort，也称为**最小排序**) 的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置；然后再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。**(例子：最大最小值，选标兵/保研名额)**
- 基本思想：
  - ① **初始状态：**无序区为R[1..n]，有序区为空；
  - ② **第i趟排序(i=1, 2, 3, …, n-1)**开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
  - ③ **n-1趟结束，**数组有序化了
- **n个记录**的直接选择排序，可经过**n-1趟**直接选择排序得到有序结果

选择排序的动图演示



选择排序的算法分析




```
function selectionSort(arr) {
    var len = arr.length;
    var minIndex, temp;
    for (var i = 0; i < len - 1; i++) {
        minIndex = i;
        for (var j = i + 1; j < len; j++) {
            if (arr[j] < arr[minIndex]) {  // 寻找最小的数
                minIndex = j;  // 将最小数的索引保存
            }
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
    return arr;
}
```

算法分析：表现最稳定的排序算法之一，因为无论什么数据进去都是$O(n^2)$的时间复杂度，所以数据规模越小越好。唯一的好处是不占用额外的内存空间。

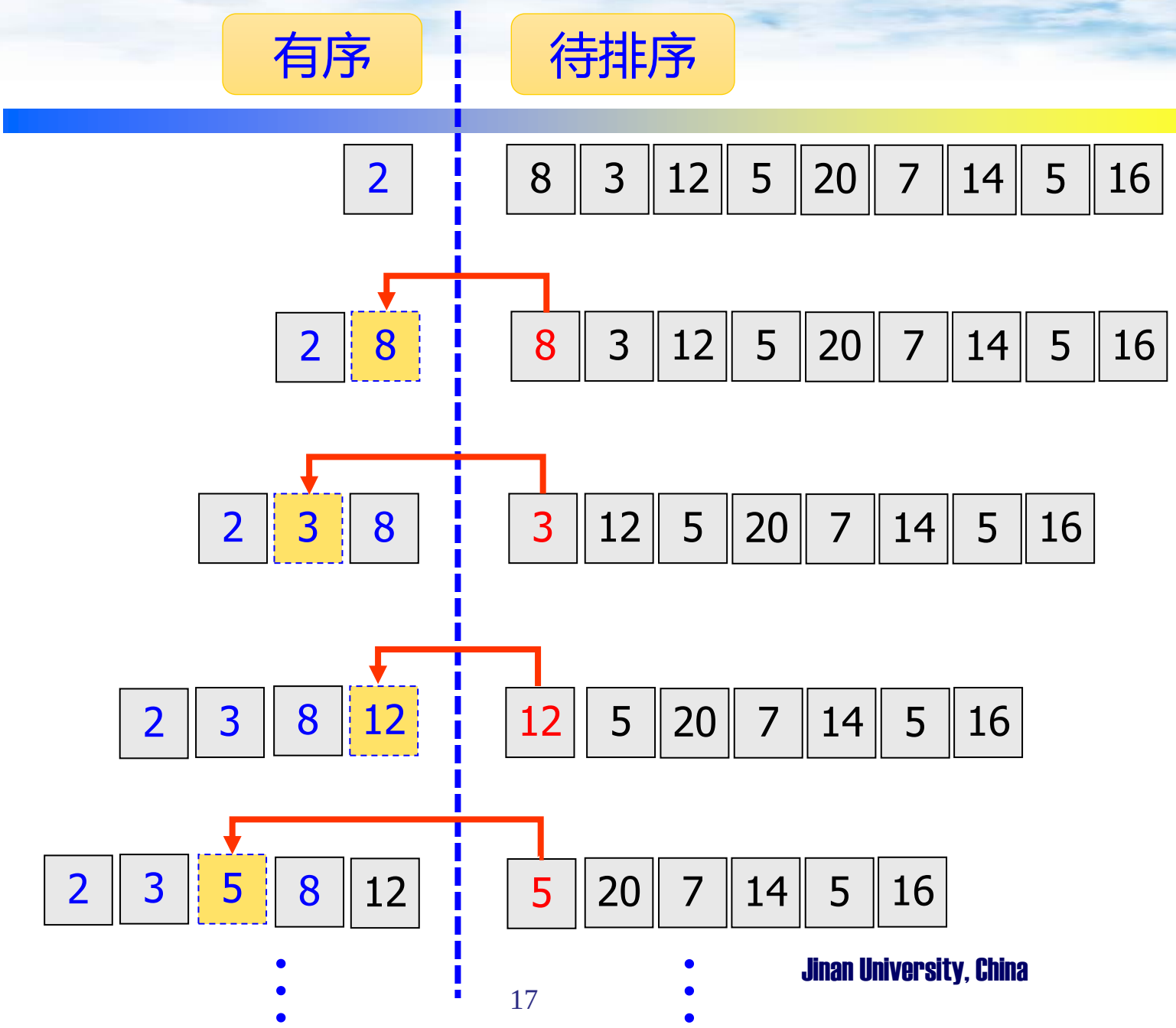
插入排序&希尔排序

内容提要：

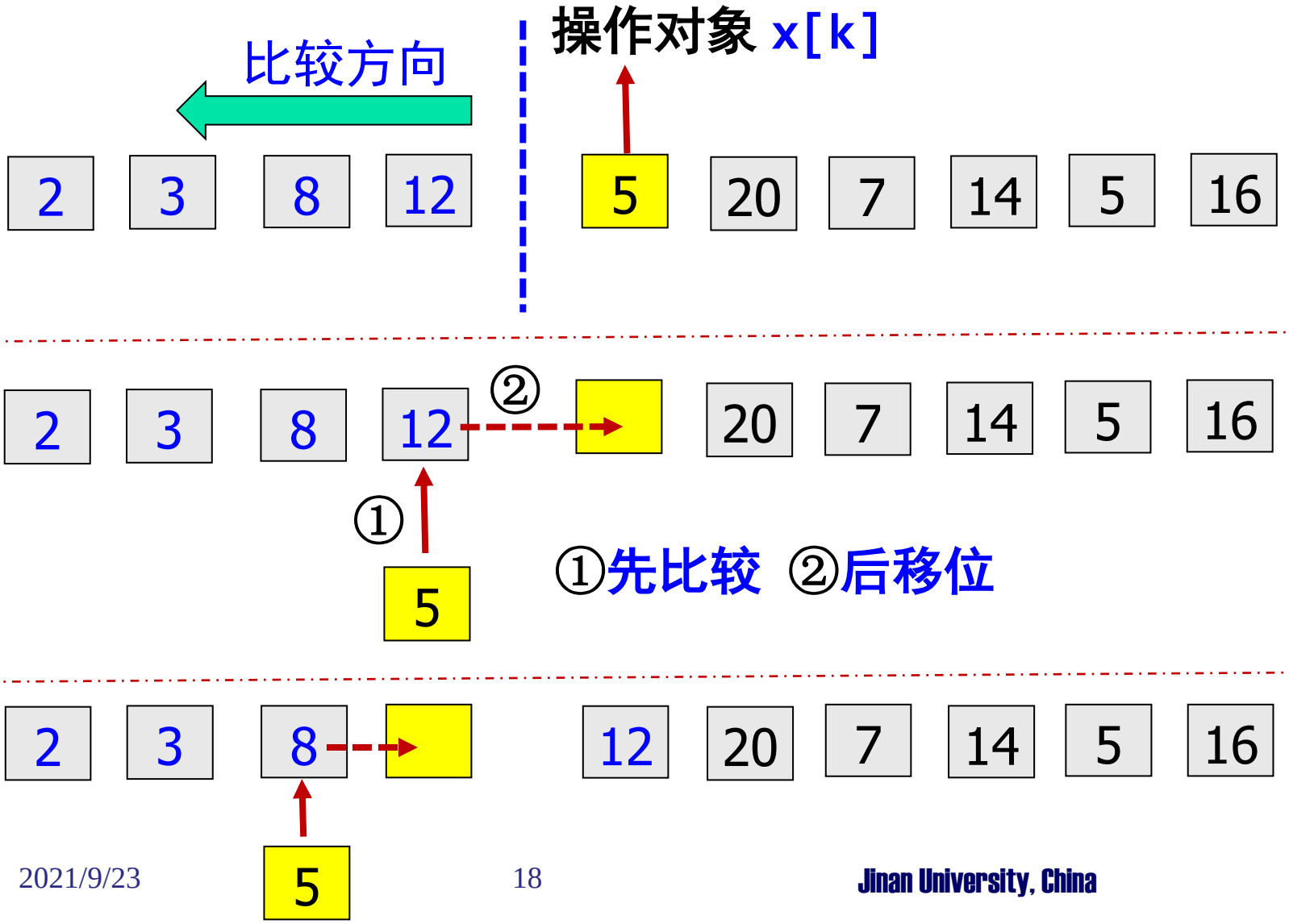
- 排序问题
- 冒泡排序 & 选择排序
- 插入排序 & 希尔排序
- 快速排序 & 堆排序
- 线性排序算法 (计数排序、桶排序、基数排序)
- 排序算法比较

插入排序

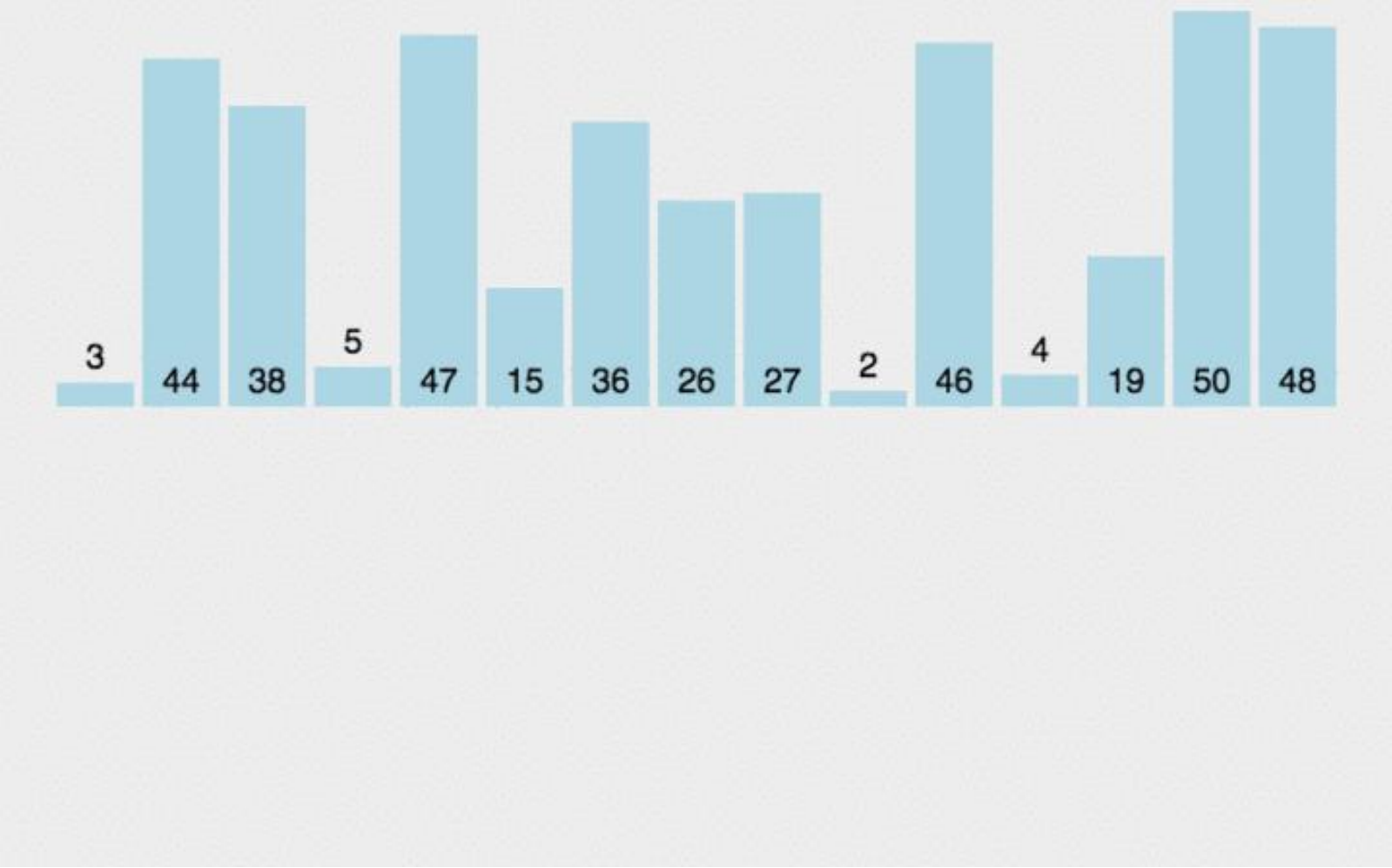
- **插入排序（Insertion-Sort）**的工作原理：通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。一般来说，插入排序采用in-place基于数组进行实现。 **(例子：打扑克牌的排序)**
- 基本思想：
  1. 从第一个元素开始，该元素可以认为已经被排序；
  2. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
  3. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
  4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
  5. 将新元素插入到该位置后；
  6. 重复步骤2~5


有序
待排序
2
8 3 12 5 20 7 14 5 16
2 8
8 3 12 5 20 7 14 5 16
2 3 8
3 12 5 20 7 14 5 16
2 3 8 12
12 5 20 7 14 5 16
2 3 5 8 12
5 20 7 14 5 16

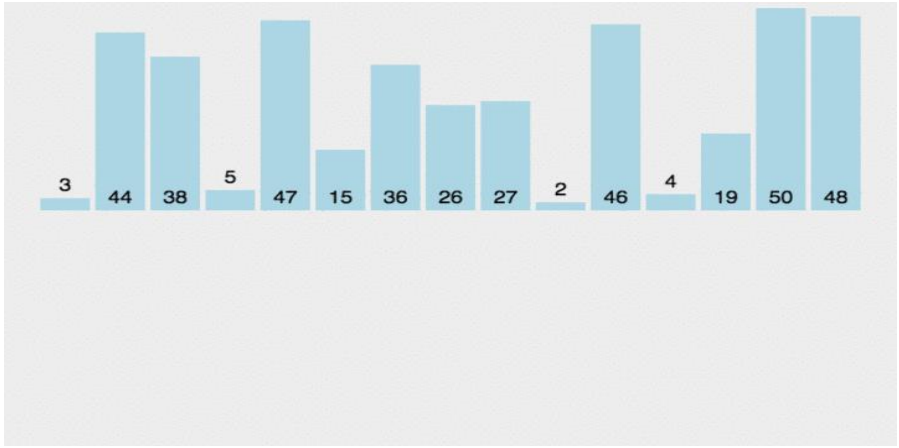
以第 4 轮为例



插入排序的动图演示



插入排序的算法分析




```
function insertionSort(arr) {
    var len = arr.length;
    var preIndex, current;

    for (var i = 1; i < len; i++) {
        preIndex = i - 1;
        current = arr[i];
        while (preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex + 1] = arr[preIndex];
            preIndex--;
        }
        arr[preIndex + 1] = current;
    }
    return arr;
}
```

算法分析：插入排序的实现，通常采用in-place排序 (即只需O(1)的额外空间)，在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

希尔排序

- **希尔排序（Shell Sort）**原理：是简单插入排序的改进版，它与插入排序的不同之处在于会优先比较距离较远的元素。希尔排序又叫**缩小增量排序(Diminishing Increment Sort）**。1959年Shell发明，**第一个突破$O(n^2)$的排序算法**。
- 基本思想：先将整个待排序的记录序列，分割成为若干组待排序的子序列，分别进行直接插入排序。

① 选择一个增量序列t_1，t_2，…，t_k，其中$t_i>t_j$，$t_k=1$；

② 按增量序列个数k，对序列进行k 趟排序；

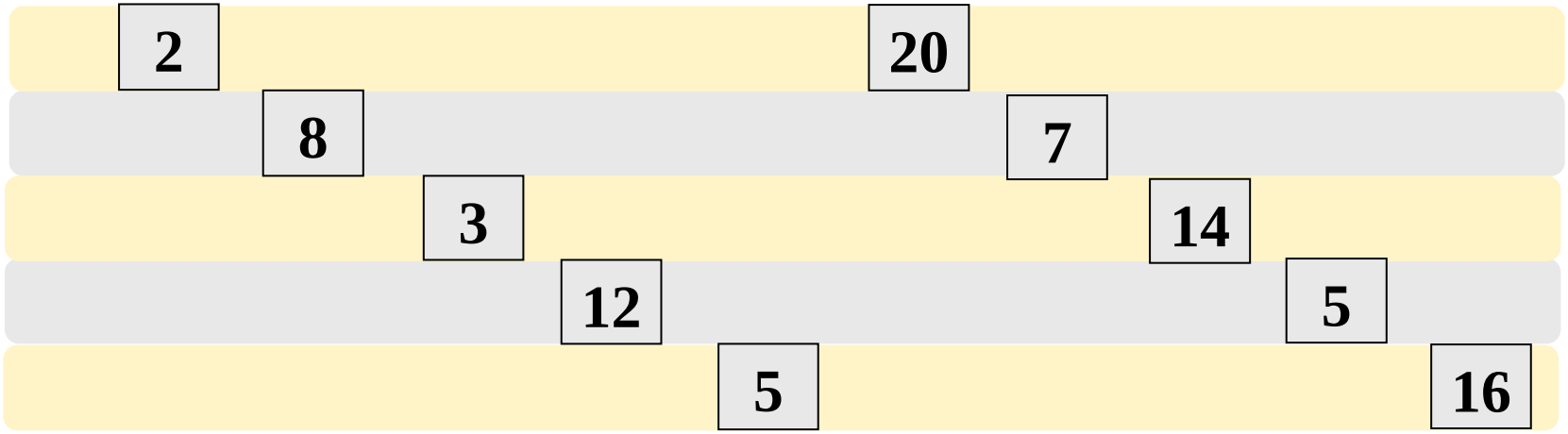
③ 每趟排序，根据对应的增量t_i，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度

出发点：插入排序在元素基本有序的情况下，效率很高。
gap：初始值设为 n/2，然后不断减半。

希尔排序的演示

2	8	3	12	5	20	7	14	5	16

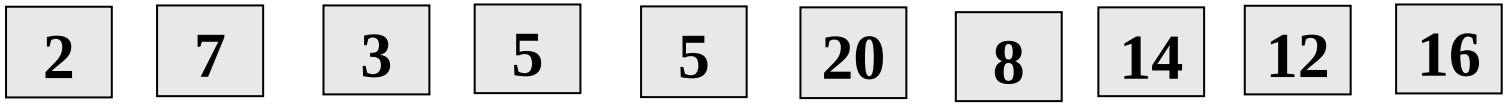
第一轮：gap = 10/2 = 5



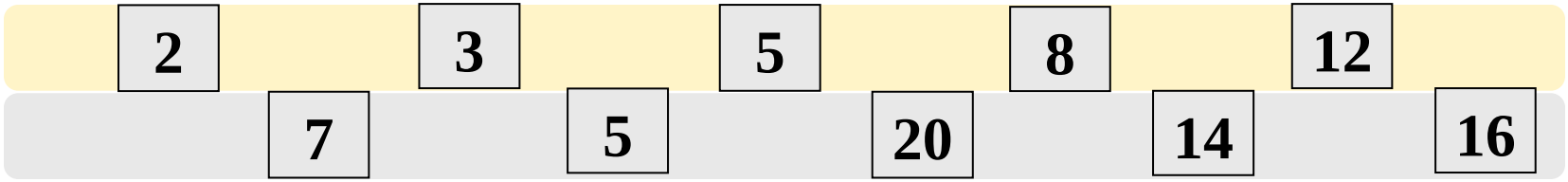
第一轮排序后

2	7	3	5	5	20	8	14	12	16

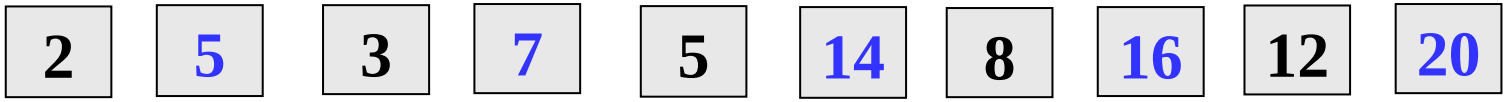
希尔排序的演示



第二轮：gap = gap/2 = 2

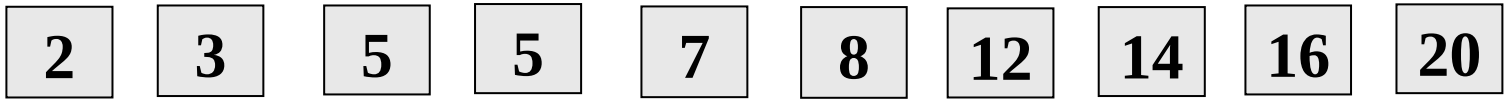


第二轮排序后

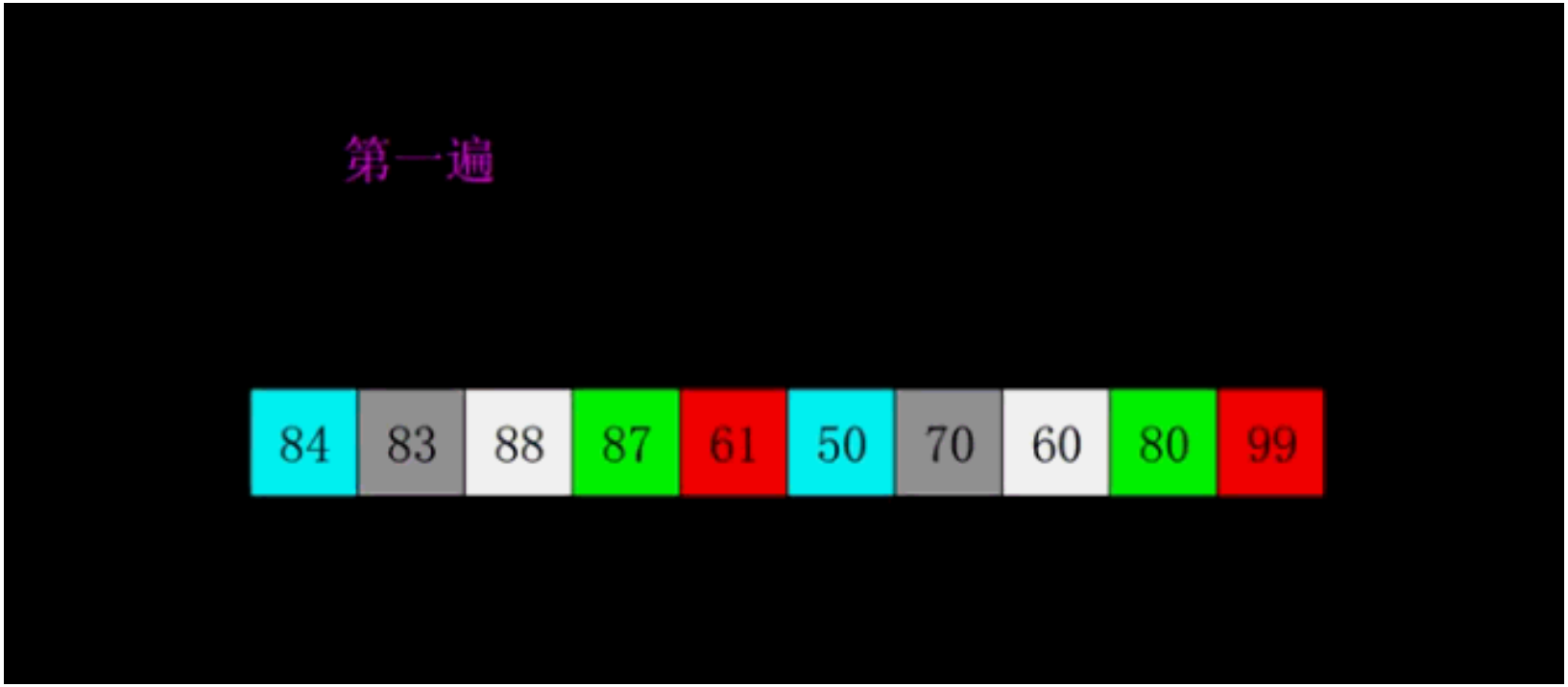


第三轮：gap = gap/2 = 1

第三轮排序后

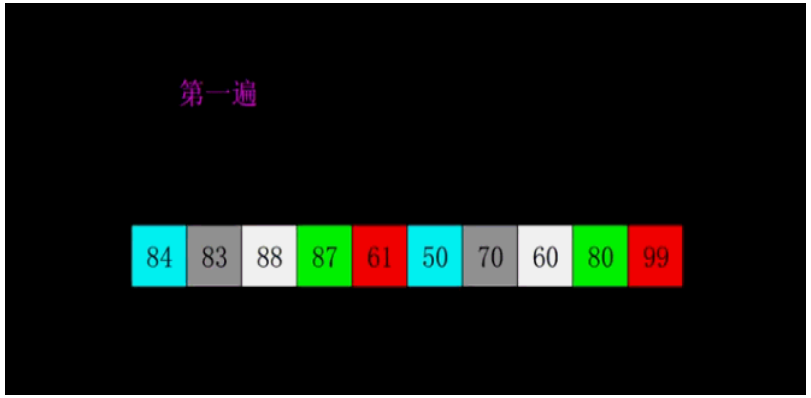


希尔排序的动图演示



希尔排序的算法分析

```
function shellSort(arr) {
    var len = arr.length;
    for (var gap = Math.floor(len / 2); gap > 0; gap =
Math.floor(gap / 2)) {
        // 注意：这里和动图演示不一样，动图是分组执
行，实际操作是多个分组交替执行
        for (var i = gap; i < len; i++) {
            var j = i;
            var current = arr[i];
            while (j - gap >= 0 && current < arr[j - gap]) {
                arr[j] = arr[j - gap];
                j = j - gap;
            }
            arr[j] = current;
        }
    }
    return arr;
}
```



算法分析：希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。动态定义间隔序列的算法是《算法（第4版）》的合著者Robert Sedgewick提出。

快速排序&堆排序

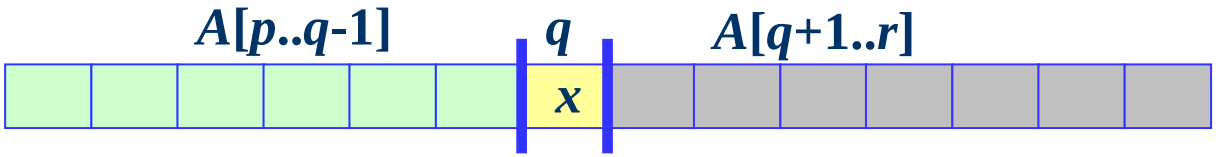
内容提要：

- 排序问题
- 冒泡排序 & 选择排序
- 插入排序 & 希尔排序
- 快速排序 & 堆排序
- 线性排序算法 (计数排序、桶排序、基数排序)
- 排序算法比较

快速排序算法

- **快速排序（Quick Sort）**由C.R.A. Hoare于1962年提出。对于包含n个数的输入数组，最坏情况运行时间为$O(n^2)$，期望运行时间为$\Theta(n \log n)$且常数因子较小。
- 基本思想: 采用**分治策略**把未排序数组分为两部分，然后分别**递归**调用自身进行排序：

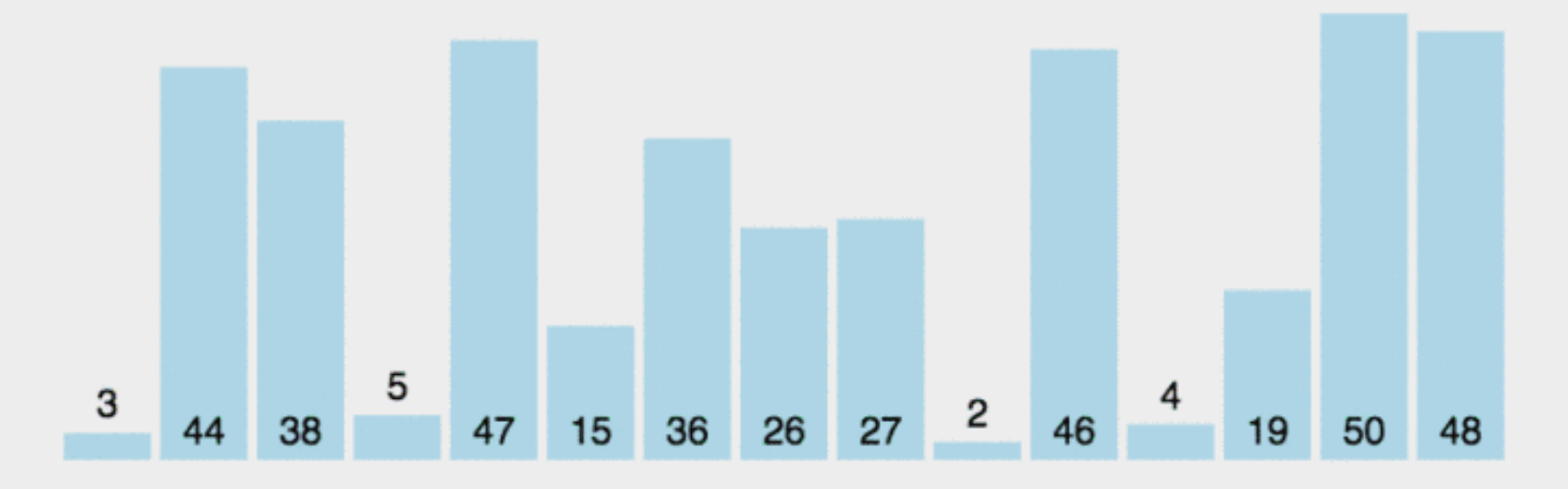
① **分解：**数组A[p…r]被划分为两个（可能空）子数组A[p…q-1]和A[p+1..r]，使得A[p…q-1]中每个元素都小于或等于A[q]和A[q+1..r]中的元素。下标q在这个划分过程中进行计算；



② **解决**：递归调用快速排序，对子数组A[p...q-1]和A[q+1..r]排序；

③ **合并**：不需要任何操作。

快速排序的动图演示



原始序列：

6	1	3	7	5	9	2	4	8	10

求第一次快速排序后的结果？

快速排序算法

- 快速排序伪代码：

```
QUICKSORT(A, p, r )
1 if   p < r
2    then  q ← PARTITION(A, p, r )
3          QUICKSORT( A, p, q-1 )
4          QUICKSORT( A, q+1, r )
```

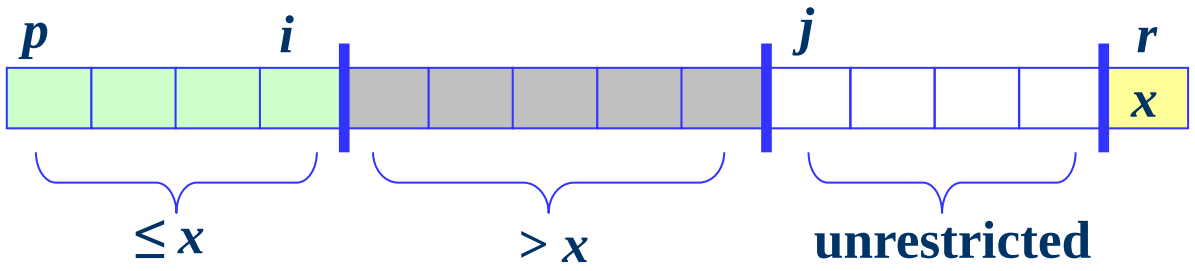
* 为排序一个完整数组，最初调用 QUICKSORT(A, 1, *length*[A])。

- **随机选定一个元素作为基准数 (pivot), 通常采用第一个元素**
- 数组划分过程PARTITION是QUICKSORT算法的关键，它对子数组A[p..r]进行就原地排序。

快速排序算法

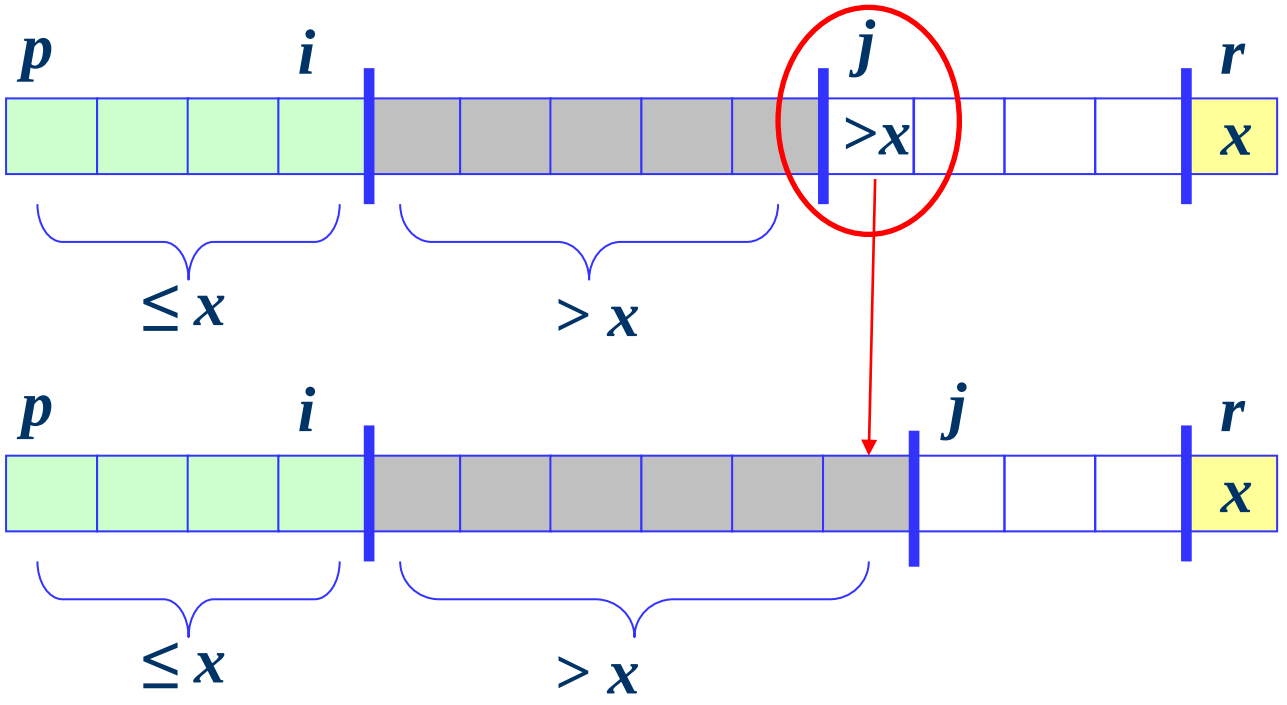
- 数组划分过程 PARTITION

```
PARTITION(A, p, r )
1 x ← A[ r ]          // x 为主元
2 i ← p - 1
3 for  j ← p to r – 1
4       do if A[ j ] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7 exchange A[i + 1] ↔ A[r]
8 return i + 1
```



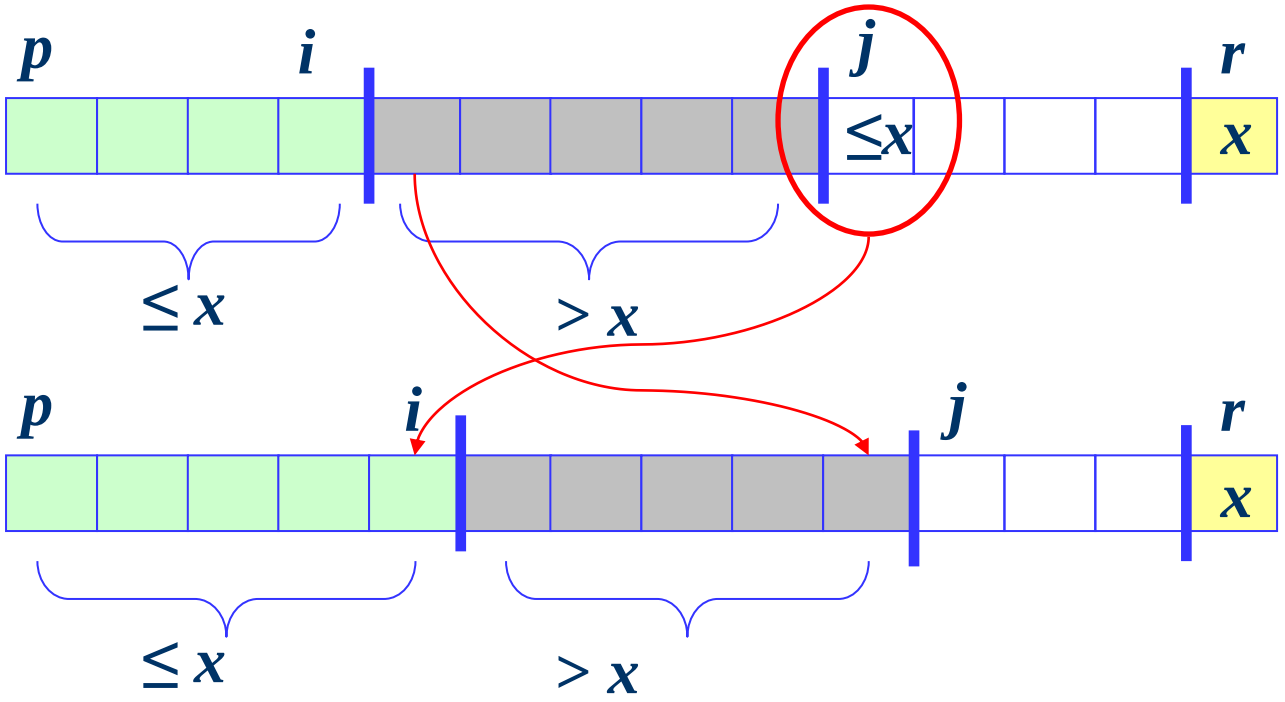
快速排序算法

i 和 *j* 如何改变：

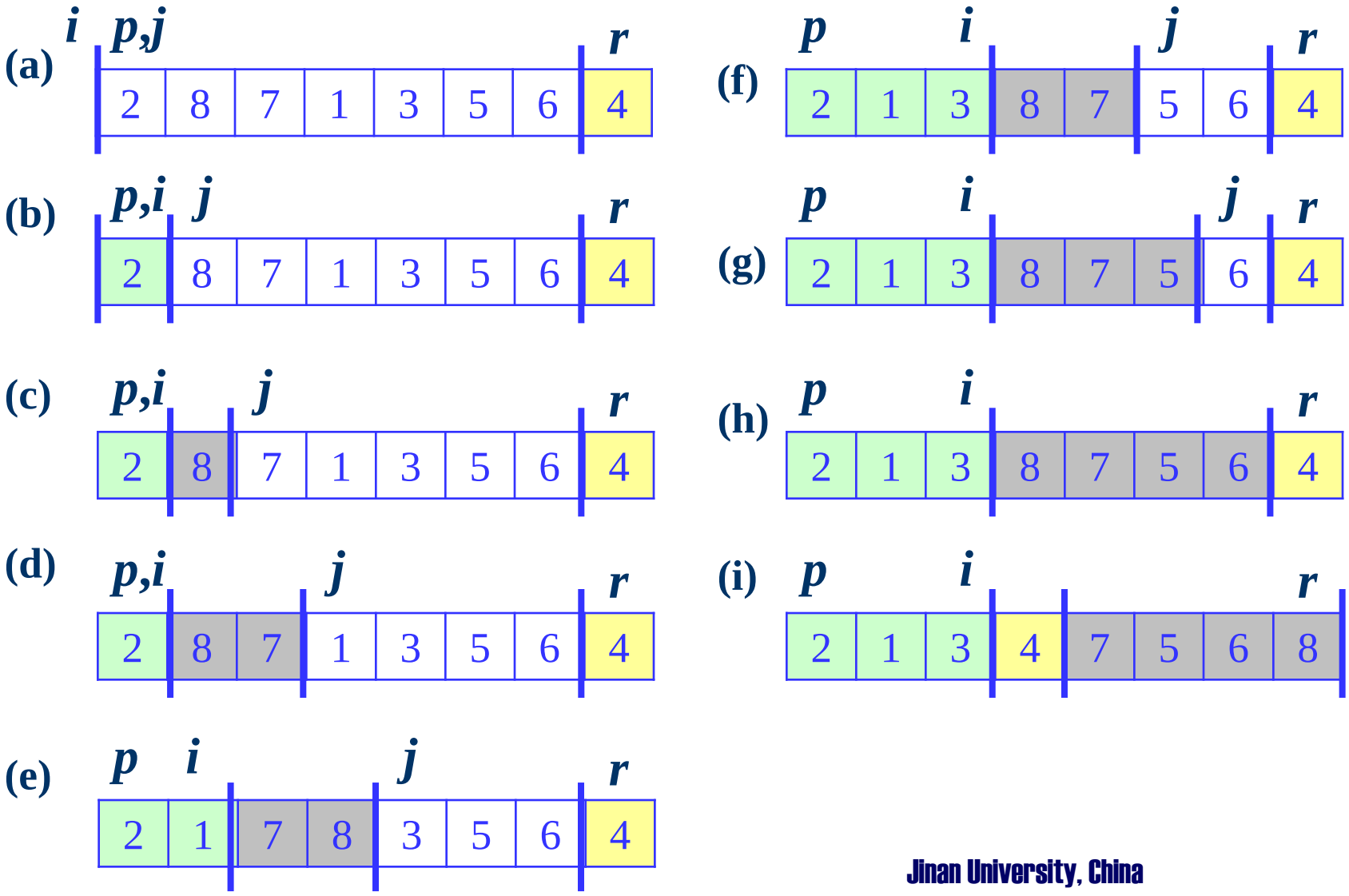


快速排序算法

i 和 *j* 如何改变:

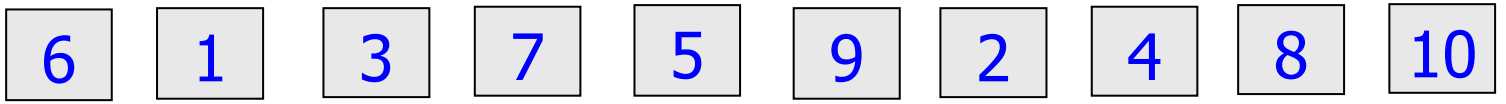


范例: (Partition, $x=A[r]=4$)



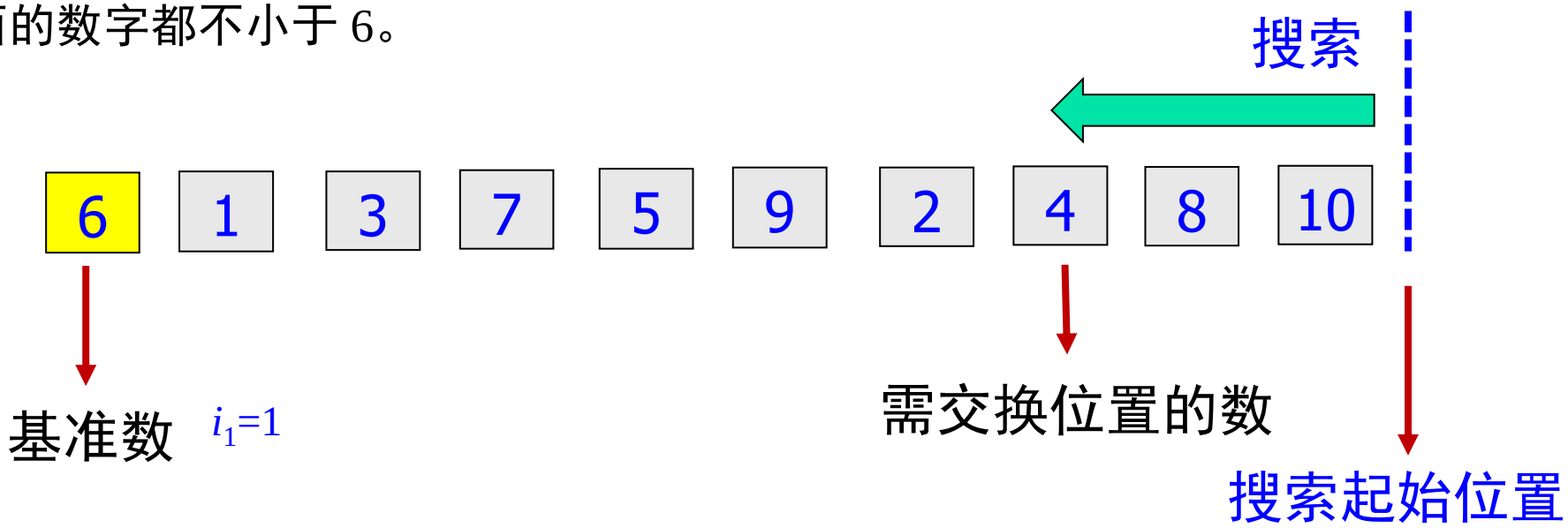
快速排序算法

原始序列：

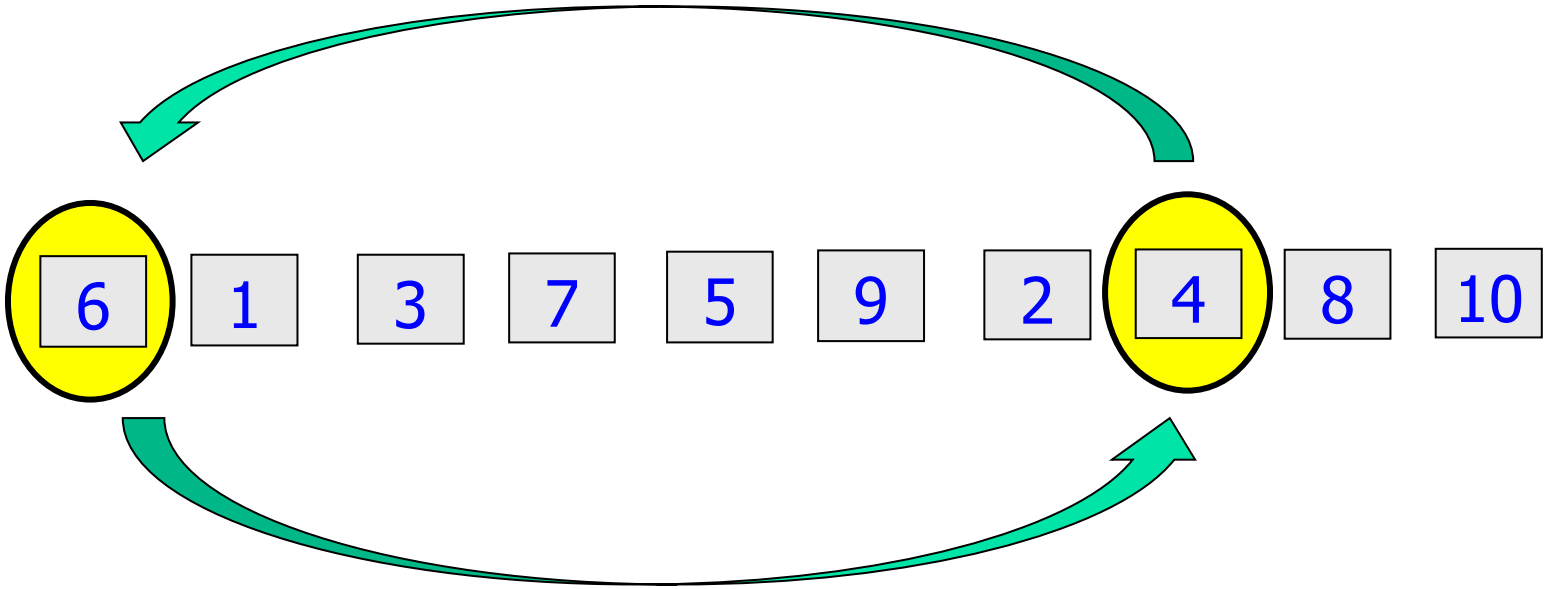


第一步：

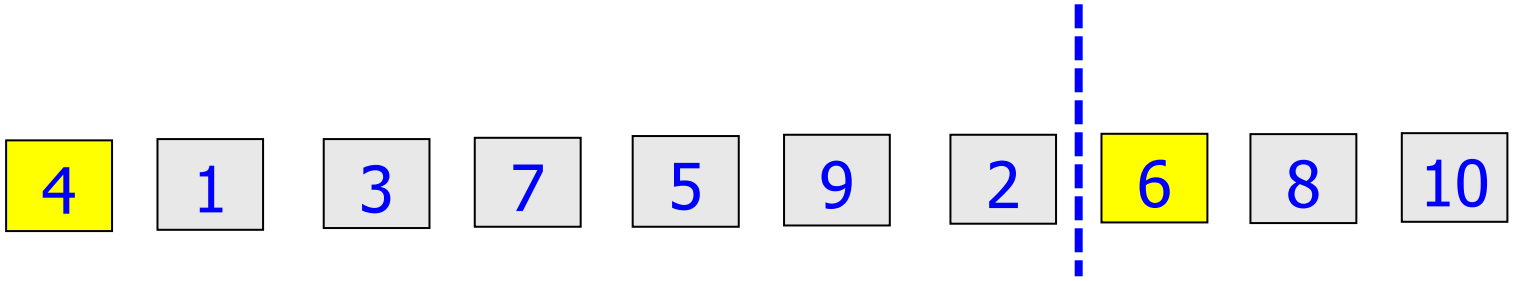
我们选第一个元素为基准数，即将 6 作为基准数。我们的目标是得到一个新序列，使得在这个新序列中，排在 6 前面的数字都小于 6，而排在 6 后面的数字都不小于 6。



快速排序算法

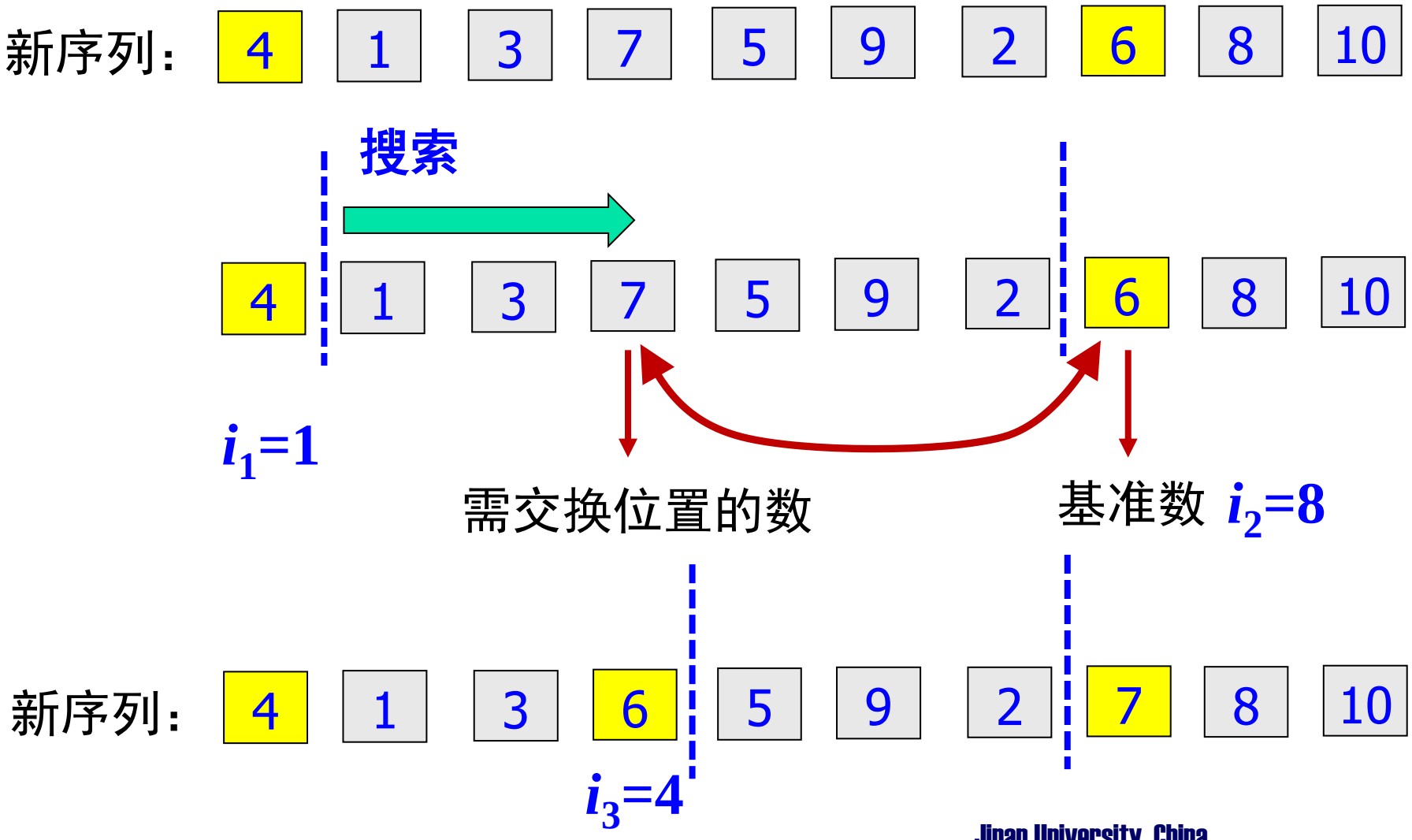


新序列：

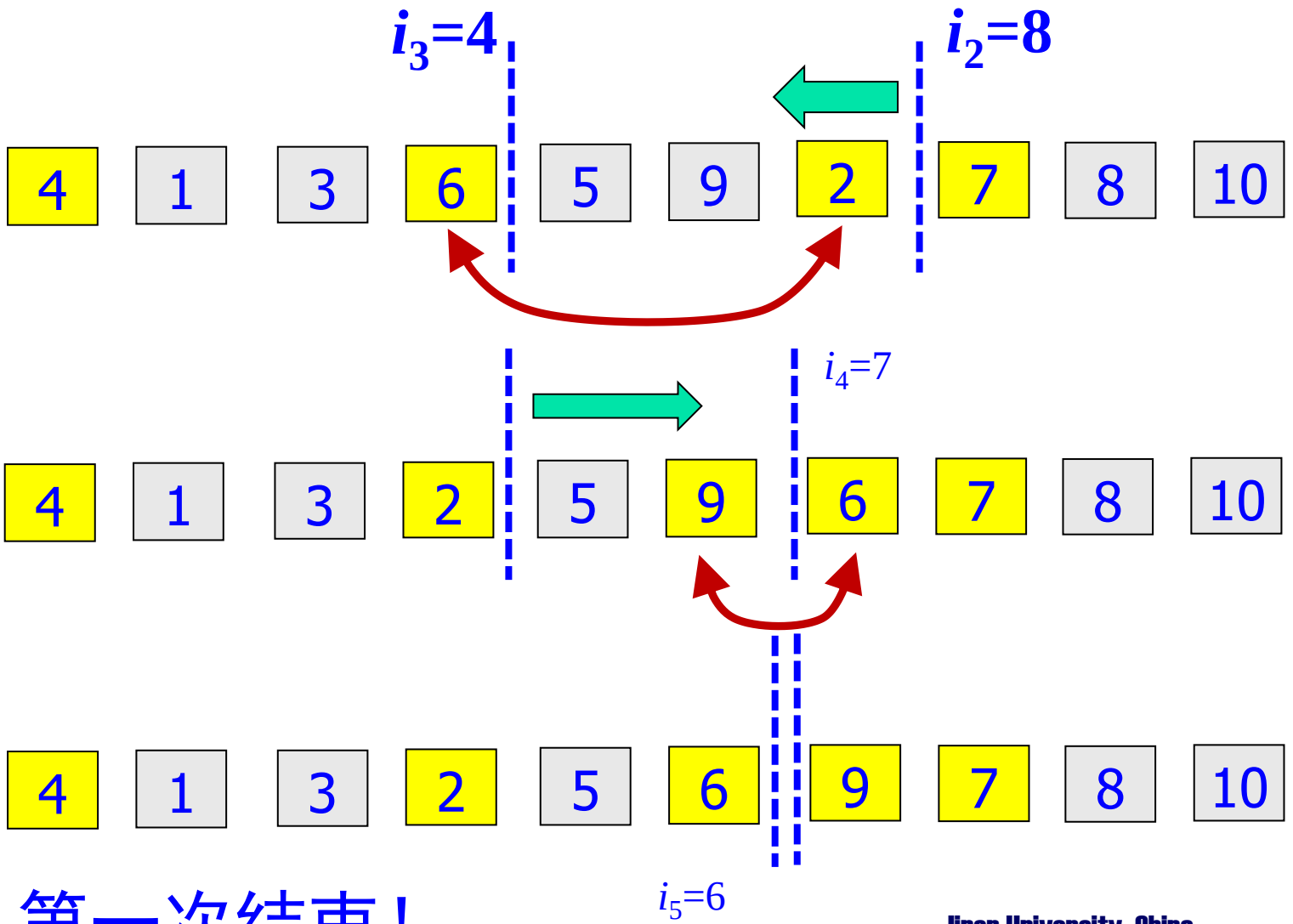


$i_1=1$　　$i_2=8$

快速排序算法


新序列：
4 1 3 7 5 9 2 6 8 10
搜索
4 1 3 7 5 9 2 6 8 10
i_1=1
需交换位置的数
基准数 i_2=8
新序列：
4 1 3 6 5 9 2 7 8 10
i_3=4

快速排序算法

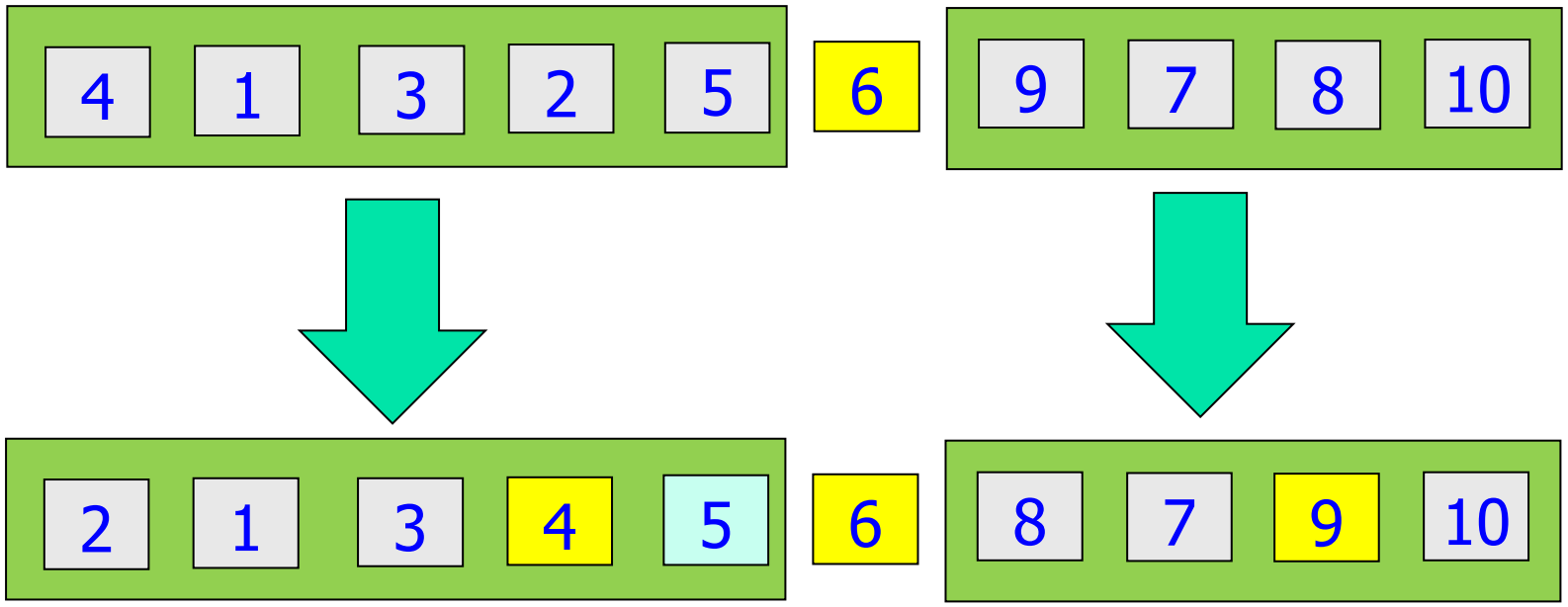


第一次结束！

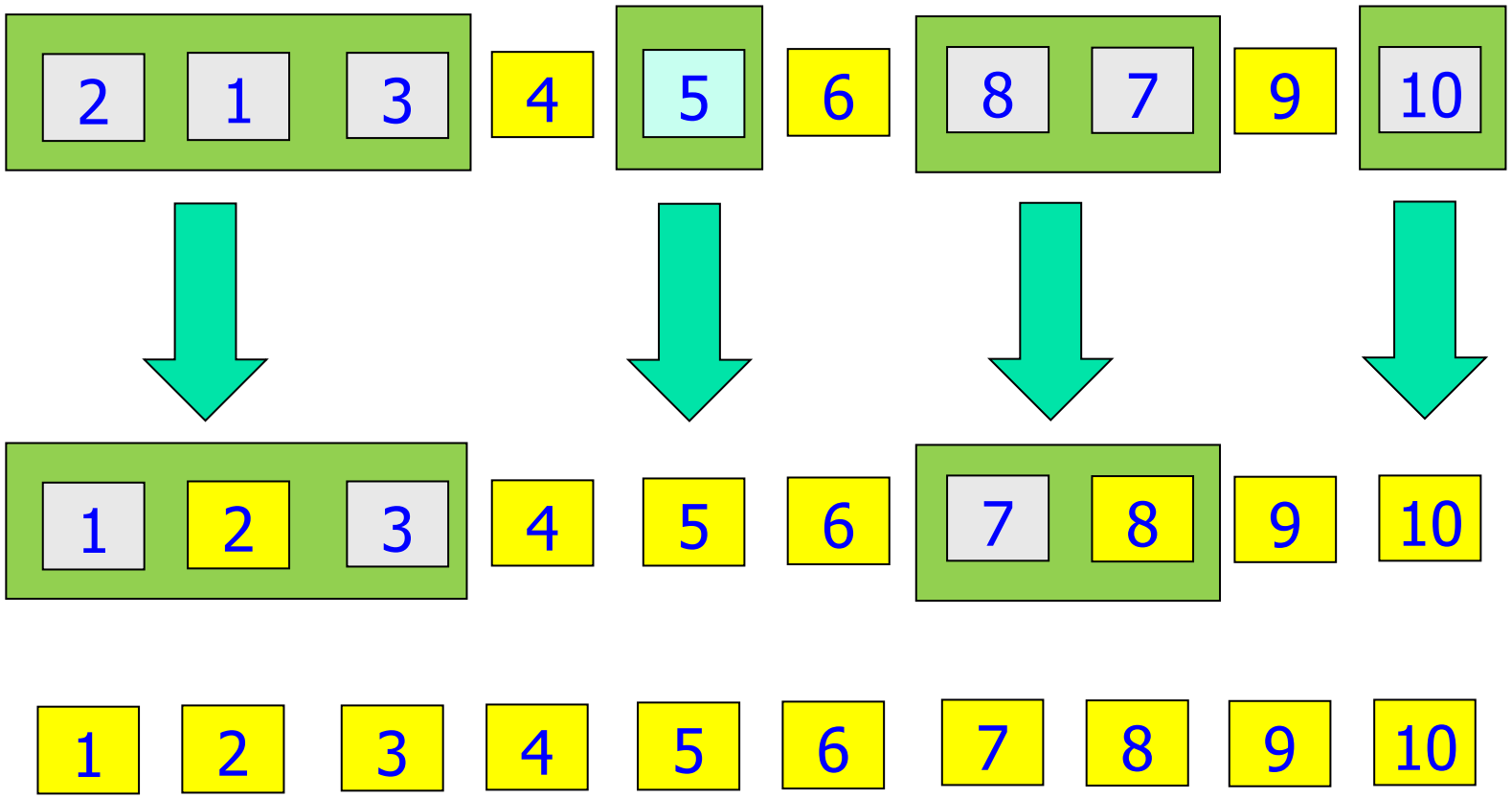
快速排序算法

第二次：

对基准数所在位置前面的子序列和后面的子序列，分别重复第一步的过程。



不断重复：递归



排序结束！

快速排序的变形

事实上，在快速排序每一步执行过程中，可以不用交换，而是直接覆盖即可。思考：如何实现？

快速排序的改进版本，如随机选择基准数，区间内数据较少时直接用其它方法排序以减小递归深度等。思考：递归深度？

有兴趣的同学可以深入研究。

留作练习

快速排序算法

- 最坏情况：$\Theta(n^2)$ (对已排序好的輸入)

$$T(n) = \max_{1 \le q \le n} \{T(q-1) + T(n-q)\} + \Theta(n)$$

$$= \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + \Theta(n)$$

猜测: $T(n) \le c\, n^2 = O(n^2)$

Substituting:

$$T(n) \le \max_{0 \le k \le n-1} \{ck^2 + c(n-k-1)^2\} + \Theta(n)$$

$$\le c \max_{0 \le k \le n-1} \{k^2 + (n-k-1)^2\} + \Theta(n)$$

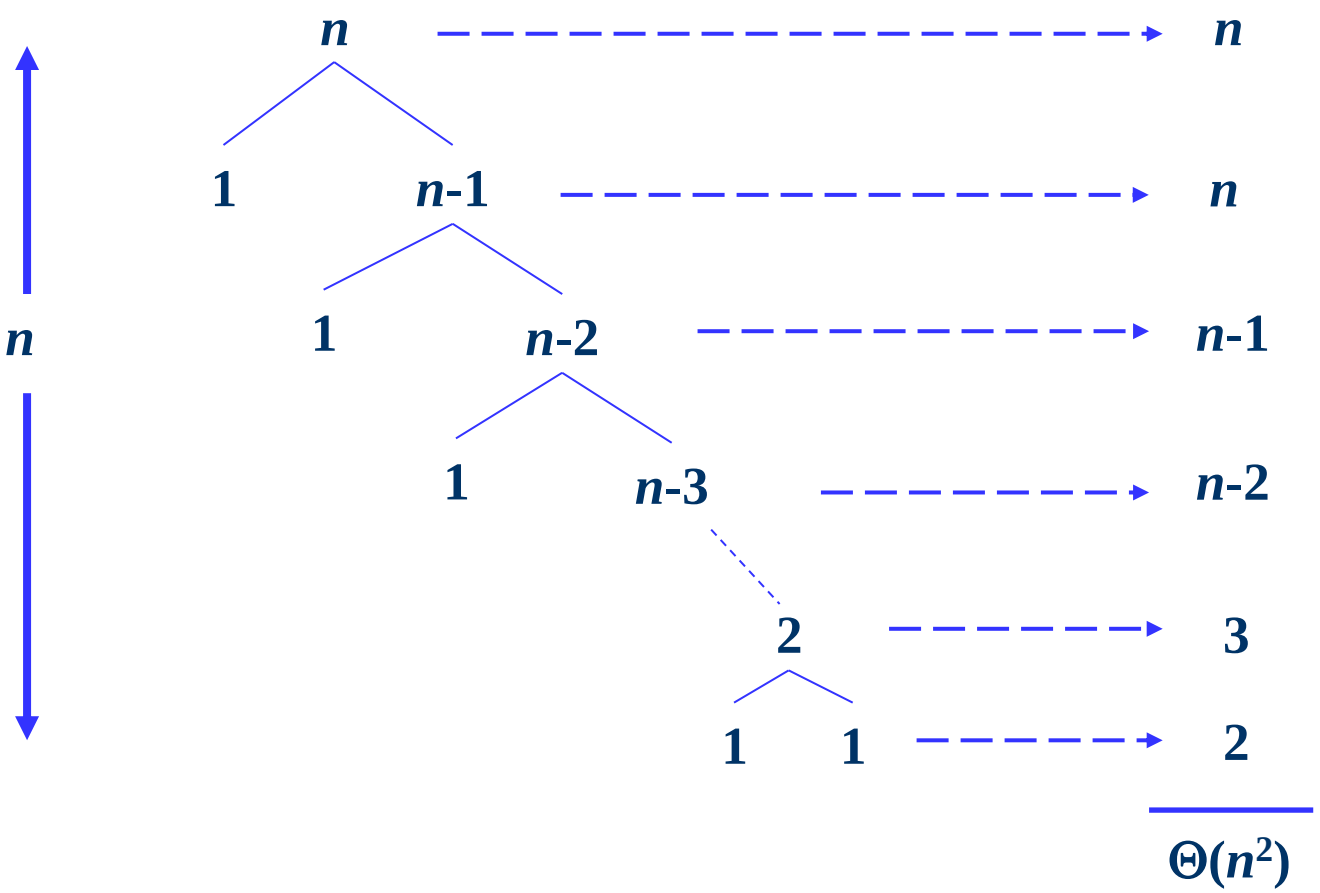
$$\le c(n\text{-}1)^2 + \Theta(n)$$

$$\le cn^2\text{-}c(2n\text{-}1) + \Theta(n)$$

$\le cn^2$ (挑选够大的 c 即可)

快速排序算法

- $T(n)=\Theta(n^2)$



快速排序算法

- 最佳情况划分： $O(n\lg n)$

  此时得到的两个子问题的大小都不可能大于$n/2$, 运行时间的递归表达式为：

$$T(n) \le 2\,T(\,n/2\,) + \Theta(\,n\,)$$

  根据主定理，该递归式的解为： $T(n) = O(\,n\lg n\,)$

- 如果以固定比例进行划分，即使该比例很不平衡（如100:1），则其运行时间仍然为$O(\,n\lg n\,)$。

快速排序算法

- 平均情况划分： $\Theta(n \lg n)$

假设所有元素都不相同，则$T(n)=O(n+X)$，X 是 Partition 中第四行的执行次数。

每次调用 Partition 的時候，如果 $A[i]<x<A[j]$ 或 $A[j]<x<A[i]$，$A[i]$ 和 $A[j]$ 将来就不会再相互比较.

快速排序算法

- 范例: 令 A = {3，9，2，7，5}。 第一个回合后，A = {3，2，5，9，7}, 之后{3，2} 再也不会和 {9，7} 比较。

- 将 A 的元素重新命名为 z_1，z_2，...，z_n，其中 z_i 是第 i 小的元素。且定义 Z_{ij} = {z_i，z_{i+1}，...，z_j}为z_i与z_j之间的元素集合。

- 定义z_i : z_j ： 当且仅当第一个从 Z_{ij} 选出來的 pivot 是 z_i 或 z_j。

快速排序算法

- 对于任意的 i 和 j，發生 $z_i : z_j$ 的概率为$2/(j-i+1)$， 因此，

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

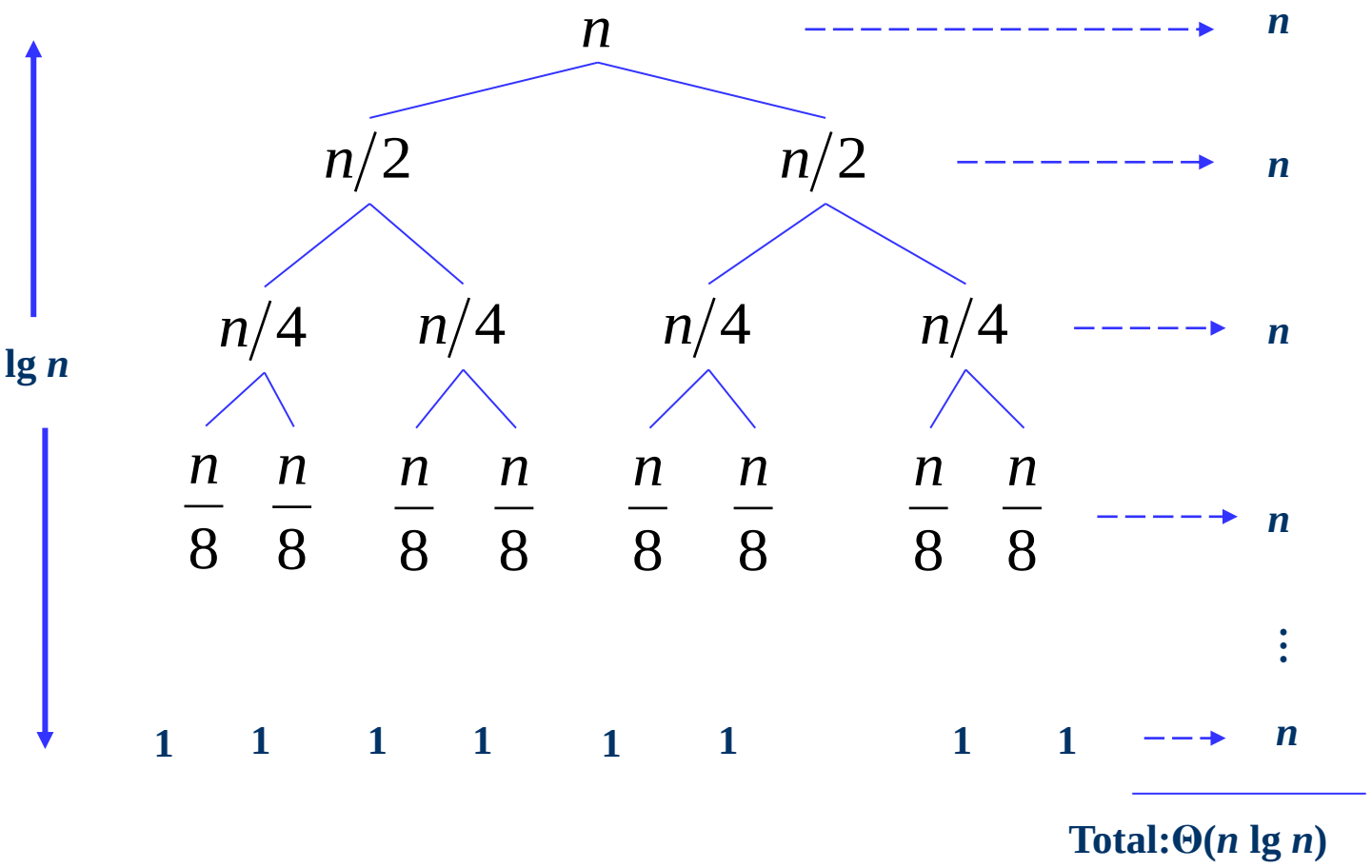
$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$ (套用 Harmonic Series)

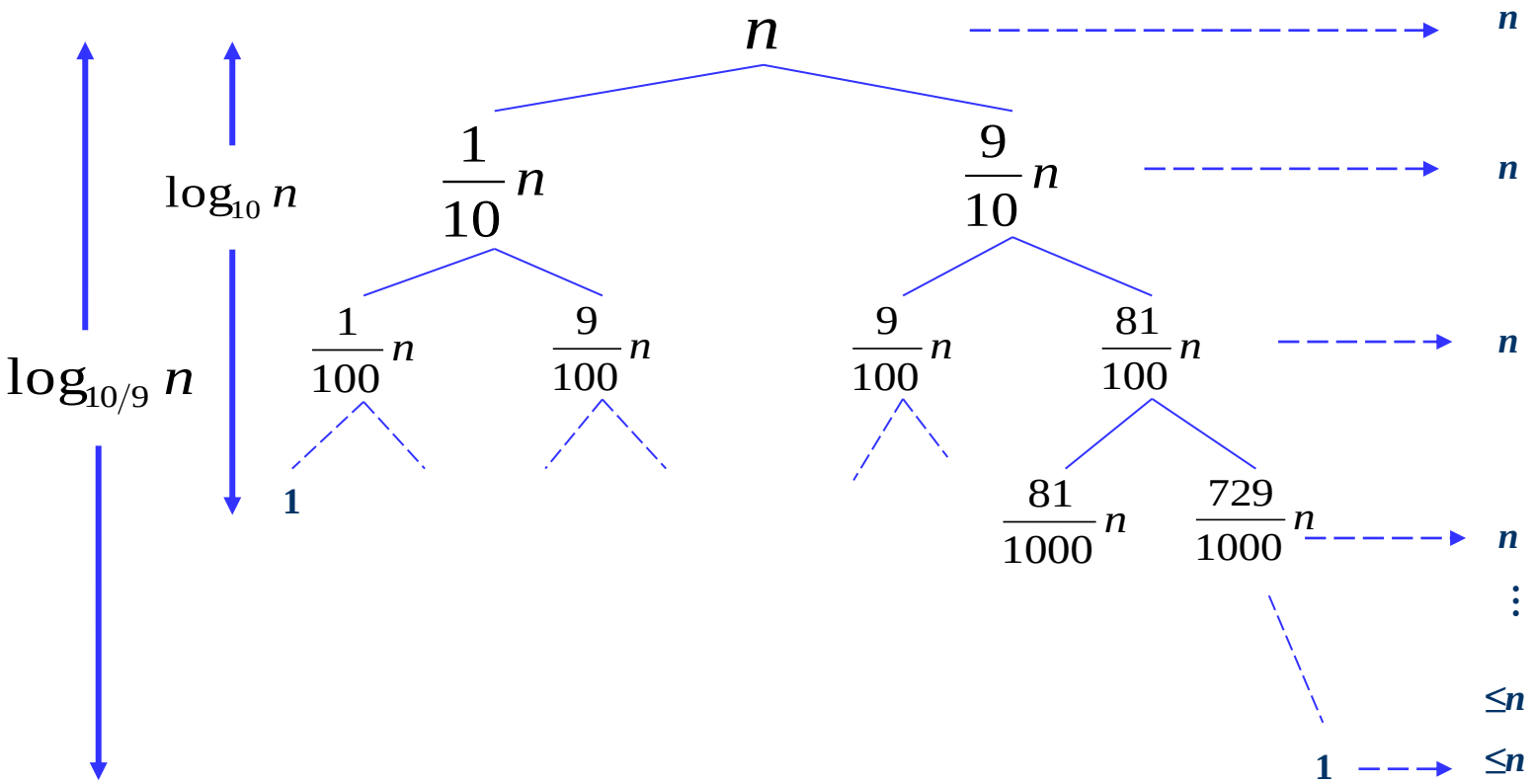
$$= \sum_{i=1}^{n-1} O(\lg n)$$

$$= O(n \lg n)$$

快速排序算法



快速排序算法


n
n
1/10 n
9/10 n
n
1/100 n
9/100 n
9/100 n
81/100 n
n
1
81/1000 n
729/1000 n
n
≤n
1
≤n
log10 n
log10/9 n
Total:Θ(n lg n)

快速排序算法

- 其他分析

$$E(n) = (n-1) + \frac{1}{n}\sum_{q=1}^{n}\{E(q-1) + E(n-q)\}$$

$$= (n-1) + \frac{2}{n}\sum_{k=1}^{n-1}E(k)$$

为了简单起见，假设：

$$E(n) = n+1 + \frac{2}{n}\sum_{k=1}^{n-1}E(k)$$

$$\Rightarrow nE(n) = n^2 + n + 2\sum_{k=1}^{n-1}E(k) \quad \text{------(1)}$$

$$\Rightarrow (n\text{-}1)E(n\text{-}1) = (n-1)^2 + (n-1) + 2\sum_{k=1}^{n-2}E(k) \quad \text{------(2)}$$

(用 *n*-1 替换掉 (1) 裡面的 *n*)

快速排序算法

(1)-(2), 可得

$$nE(n) = (n+1)E(n-1)+2n$$

$$\Rightarrow E(n) = \frac{n+1}{n}E(n-1)+2 \quad \text{(套用 iteration method)}$$

$$= \frac{n+1}{n}\left\{\frac{n}{n-1}E(n-2)+2\right\}+2 = \frac{n+1}{n-1}E(n-2)+2\frac{n+1}{n}+2$$

$$= \frac{n+1}{n-2}E(n-3)+2\frac{n+1}{n-1}+2\frac{n+1}{n}+2 = \bullet \quad \bullet \quad \bullet \quad \bullet$$

$$= \frac{n+1}{2}E(1)+2(n+1)\left(\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}\right)+2 = \Theta(n)+\Theta(n)\sum_{k=3}^{n}\frac{1}{k}+2$$

$$= \Theta(n)+\Theta(n)\times\Theta(\lg n)+2 \quad \text{(套用 Harmonic Series)}$$

$$= \Theta(n\lg n)$$

快速排序的随机化版本

- 如何防止出现最坏情况发生？

- **策略1**：显示地对输入进行排列使得快速排序算法随机化

```
RANDOMIZED-QUICKSORT(A, p, r )
1 if   p < r
2     RANDOMIZE-IN-PLACE(A)
3     QUICKSORT( A )
```

- 为了实现随机化，是否还有其它策略？

快速排序的随机化版本

- **策略2**：采用**随机取样（random sampling）**的随机化技术
- **做法**：从子数组$A[p…r]$中随机选择一个元素作为主元，从而达到可以对输入数组的划分能够比较对称。

```
RANDOMIZED-PARTITION(A, p, r )
1  i ← RANDOM( p, r )
2     exchange A[r] ↔ A[i]
3     return PARTITION( A, p, r )
```

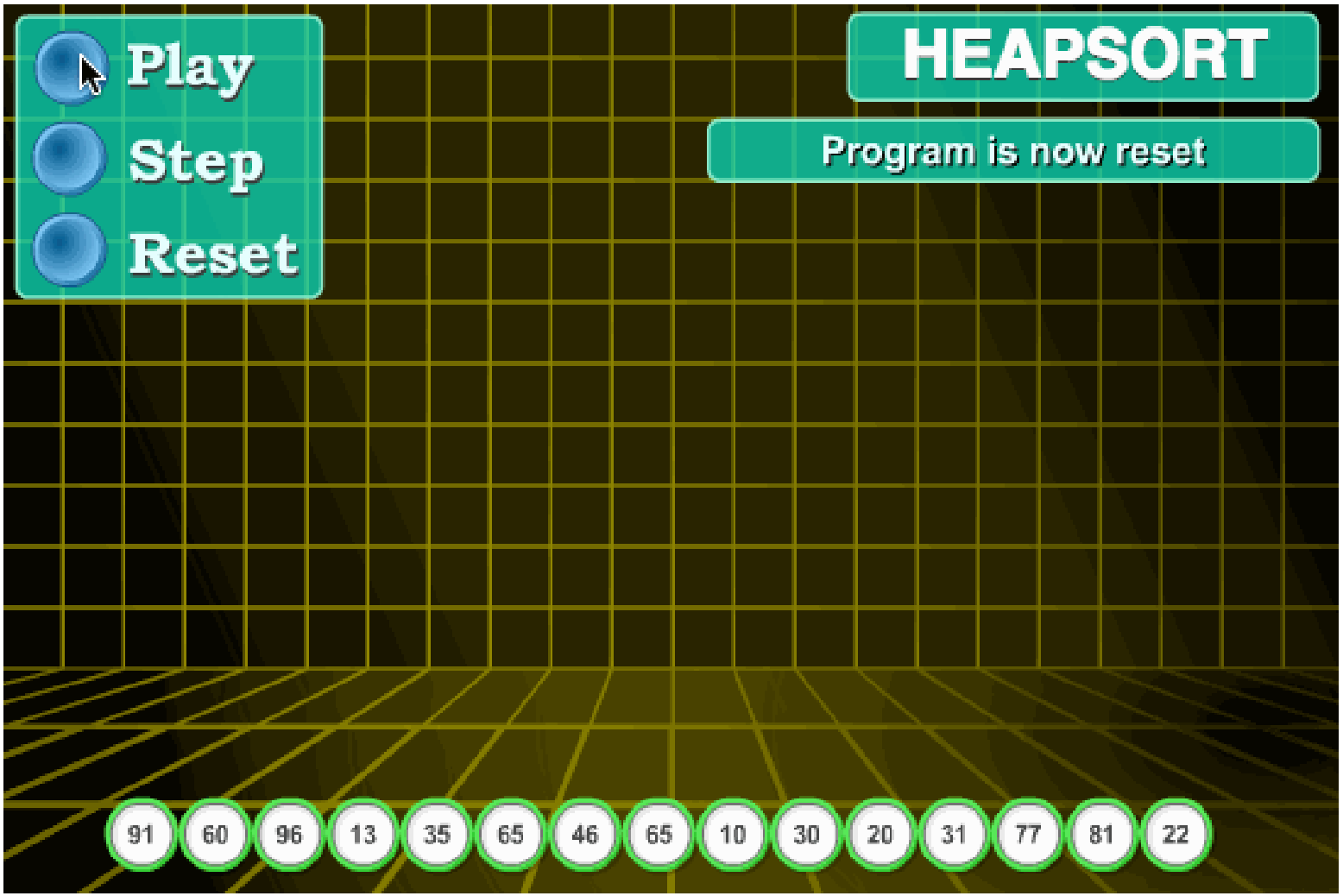
- 新排序算法调用RANDOMIZED-PARTITION

```
RANDOMIZED-QUICKSORT(A, p, r )
1 if   p < r
2    then  q ← RANDOMIZED-PARTITION(A, p, r )
3          RANDOMIZED-QUICKSORT( A, p, q-1 )
4          RANDOMIZED-QUICKSORT( A, q+1, r )
```

堆排序

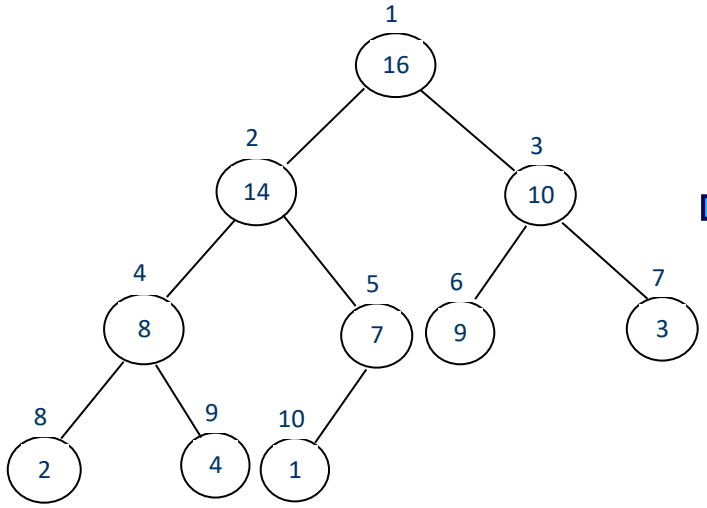
- **堆排序（Heapsort）**是指利用堆该数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：**即子结点的键值或索引总是小于（或者大于）它的父节点**。
- 基本思想：
  ① 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
  ② 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；
  ③ 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成

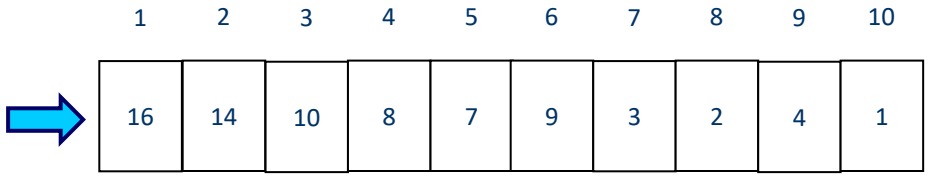
堆排序的动图演示


Play
Step
Reset
HEAPSORT
Program is now reset
91
60
96
13
35
65
46
65
10
30
20
31
77
81
22

堆数据结构

- 堆数据结构是一种数组对象，可以被视为一棵完全二叉树。树中每个节点与数组中存放该结点值的元素对应。树的每一层都是填满的，最后一层可能除外（从一个结点的左子树开始填）





- 表示堆的数组对象A具有两个性质：

① *length*[A]：是数组中的元素个数； *heap-size*[A]: 是存放在A中的堆元素个数；

② *heap-size*[A] ≤ *length*[A]

堆数据结构

- 作为数组对象的堆，给定某个结点的下标i，则：

① 父节点PARENT(i) = floor(i /2),

② 左儿子为LEFT(i) = 2 i，右儿子为RIGHT(i) = 2 i + 1;

- 堆的分类：

➢ **大根堆（最大堆）**：除根节点之外的每个节点i，有 $A[PARENT(i)] \geq A[i]$

即某结点的值不大于其父结点的值，故堆中的最大元素存放在根结点中。

➢ **小根堆（最小堆）**：除根节点之外的每个节点i，有$A[PARENT(i)] \leq A[i]$

即某结点的值不小于其父结点的值，故堆中的最小元素存放在根结点中。

- 在堆排序算法中，如果使用大根堆，则堆中最大元素位于树根；
- 小根堆通常在构造优先队列时使用。

堆数据结构

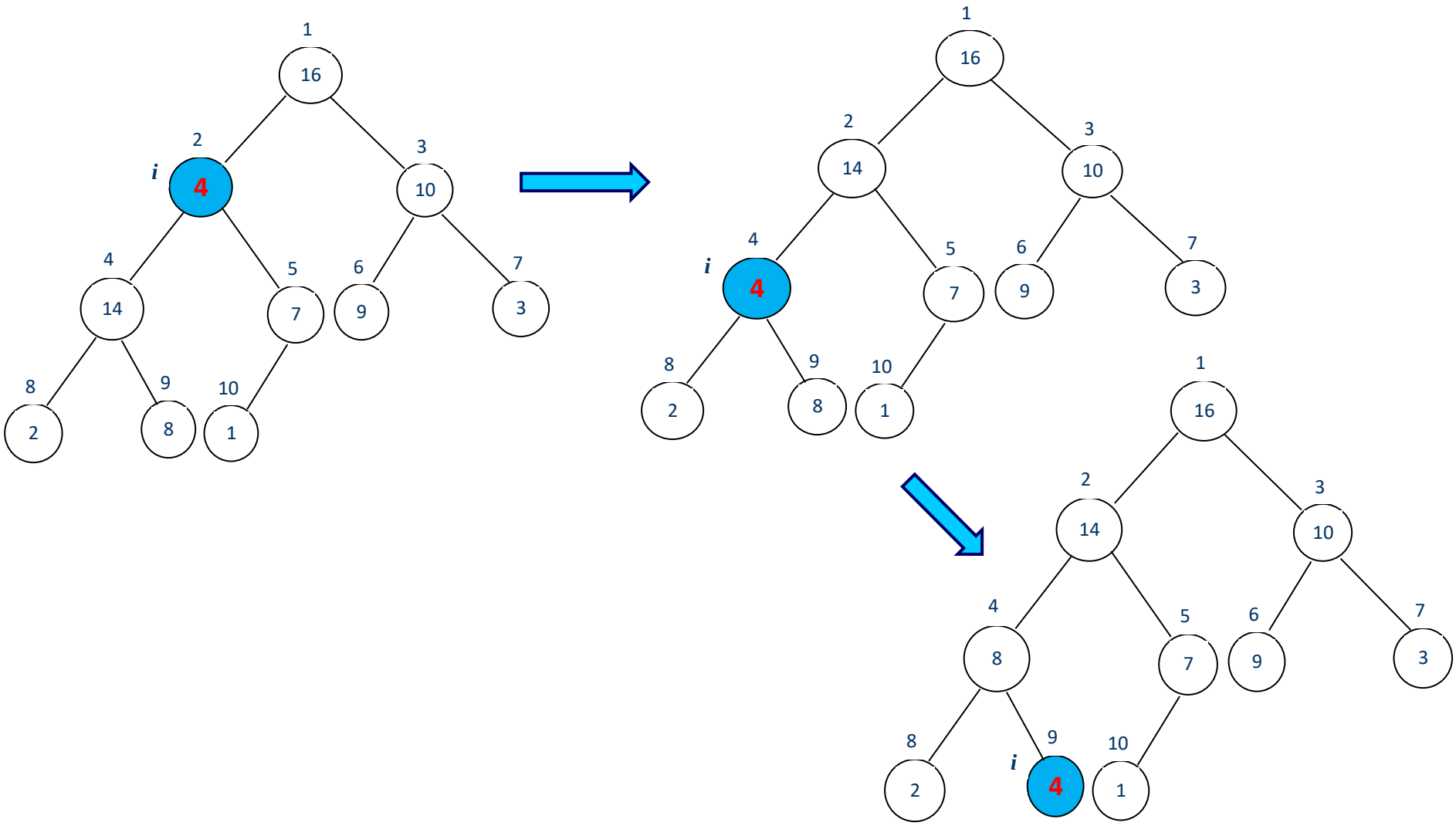
- 视为完全二叉树的堆：
  - ✓ 结点在堆中的高度定义为从本结点到叶子的最长简单下降路径上边的数目；
  - ✓ 定义堆的高度为树根的高度；
  - ✓ 具有n个元素的堆其高度为$\Theta(lg\ n)$；
- 堆结构的基本操作：
  - ✓ MAX-HEAPIFY，运行时间为$O(lg\ n)$，保持最大堆性质；
  - ✓ BUILD-MAX-HEAP，以线性时间运行，能在无序的输入数组基础上构造出最大堆；
  - ✓ HEAPSORT，运行时间为$O(nlgn)$，对一个数组进行原地排序；
  - ✓ MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY 和 HEAP-MAXIMUM过程的运算时间为$O(lg\ n)$, 可以让堆结构作为优先队列使用；

保持堆性质

- MAX-HEAPIFY函数的输入为一个数组A和小标*i*。假定以LEFT(*i*) 和RIGHT(*i*)为根的两棵二叉树都是最大堆，MAX-HEAPIFY让A[*i*]在最大堆中“下降”，使以*i*为根的子树成为最大堆。

```
MAX-HEAPIFY( A, i )
1   l ← LEFT( i );
2   r ← RIGHT( i );
3 if  l ≤ heap-size[A] and A[ l ] > A[ i ]
4       then largest ← l
5       else   largest ← i
6 if  r ≤ heap-size[ A] and A[ r ] > A[ largest ]
7       then largest ← r
8 if  largest ≠ I
9        then exchange A[ i ] ↔ A[ largest ]
10              MAX-HEAPIFY( A, largest )
```

保持堆性质


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1

保持堆性质

- **基本思想：**

1）找出A[*i*], A[LEFT(*i*)]和A[RIGHT(*i*)]中最大者，其下标存在*largest*；

2）交换A[i]和A[*largest*]使得结点i和其子女满足最大堆性质；

3）下标为*largest*的结点在交换后的值是A[*i*]，以该结点为根的子树有可能违反最大堆性质，对该子树递归调用MAX-HEAPIFY;

保持堆性质

- **时间复杂度分析：**

当MAX-HEAPIFY作用在一棵以结点*i*为根的、大小为*n*的子树上时，其运行时间为调整元素A[*i*]、A[LEFT(*i*)]和A[RIGHT(*i*)]的关系时所用时间Θ(1)，再加上对以i的某个子节点为根的子树递归调用MAX-HEAPIFY所需的时间。i结点的子树大小最多为2*n*/3（此时，最底层恰好半满），运行时间递归表达式为：

$$T(n) \le T(2n/3) + \theta(1)$$

根据主定理，该递归式的解为 $T(n) = O(\lg n)$

即：MAX-HEAPIFY作用于一个高度为h的结点所需的运行时间为 $O(h)$

建堆操作

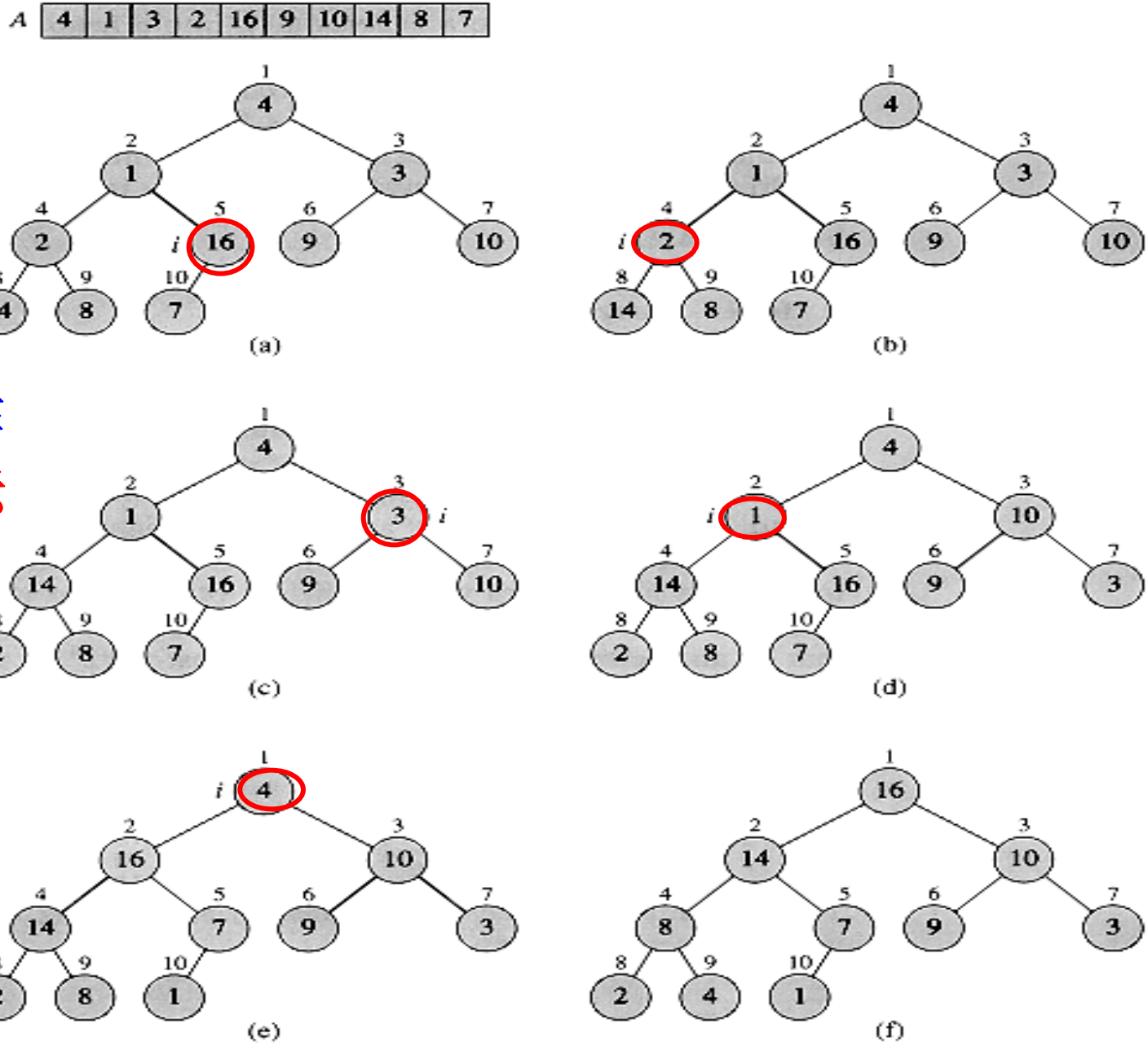
- 输入一个无序数组A，BUILD-MAX-HEAP把数组A变成一个最大堆（自底向上地建堆），伪代码如下：

```
BUILD-MAX-HEAP ( A )
1    heap-size[A] ← length[A];
2 for i ← FLOOR( length[A]/2 )  down to 1
3       do MAX-HEAPIFY( A, i )
```

- 注意：循环下标的i 变化, i = FLOOR(*length*[A]/2)
- 基本思想：数组A[(n/2 +1) … n]中的元素都是树中的叶子结点，因此每个都可以看作是只含一个元素的堆。BUILD-MAX-HEAP对数组中每一个其它内结点从后往前都调用一次MAX-HEAPIFY。

建堆操作

循环下标的*i* 变化



建堆操作

- **时间复杂度分析：**

在树中不同高度的结点处运行MAX-HEAPIFY的时间不同，其作用在高度为h的结点上的运行时间为O(h)，故BUILD-MAX-HEAP时间代价为：

$$T(n) \le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h+1}} \right)$$

$$\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} \right) = O(n)$$

这说明，BUILD-MAX-HEAP可以在线性时间内，将一个无序数组建成一个最大堆。

堆排序算法

基本思想：

① 调用BUILD-MAX-HEAP将输入数组A[1…n]构建成一个最大堆；

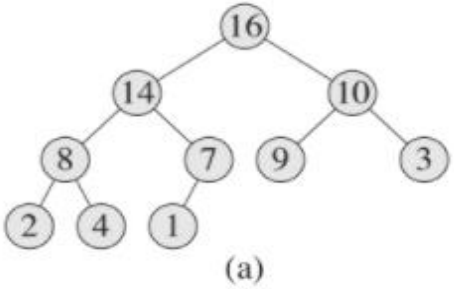
② 互置A[1]和A[n]位置，使得堆的最大值位于数组正确位置；

③ 减小堆的规模；

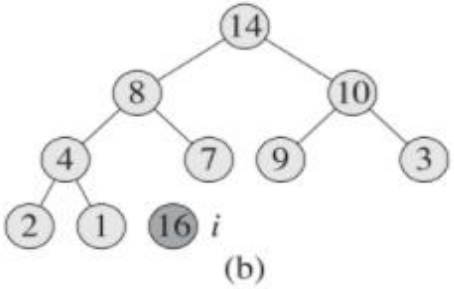
④ 重新调整堆，保持最大堆性质。

```
HEAPSORT( A )
1  BUILD-MAX-HEAP(A)
2  for  i ← length[A] downto 2
3     do  exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] -1
5           MAX-HEAPIFY( A, 1)
```

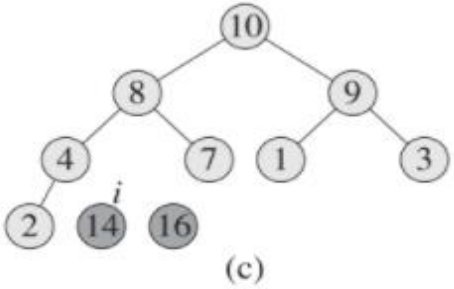
注意：循环下标的*i* 变化



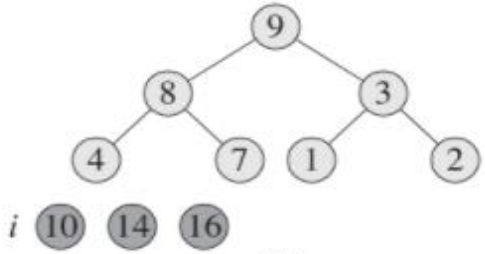
(a)



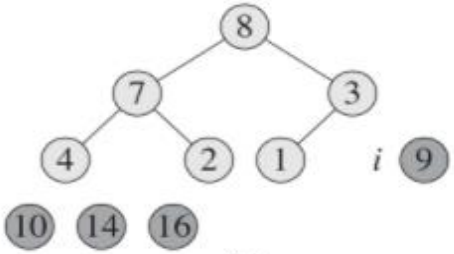
(b)



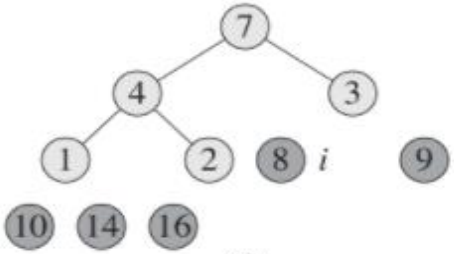
(c)



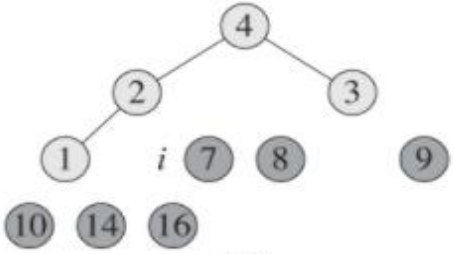
(d)



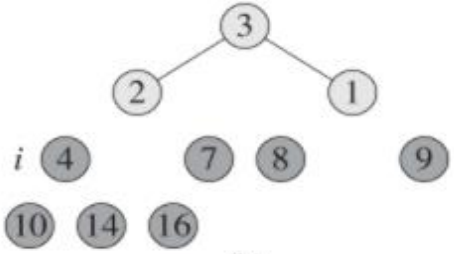
(e)



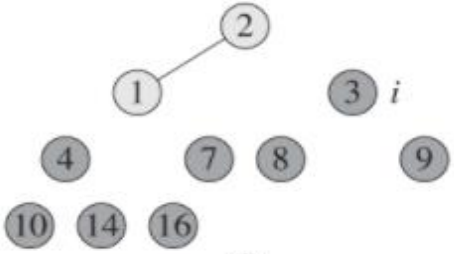
(f)



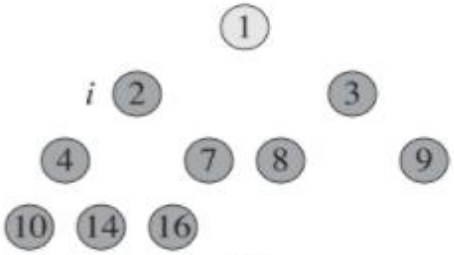
(g)



(h)



(i)



(j)

(k)

HEAPSORT(A)操作过程

堆排序算法

- **时间复杂度分析：**

  调用BUILD-MAX-HEAP时间为$O(n)$, n-1次MAX-HAPIFY调用的每一次时间代价为$O(lgn)$。

  HEAPSORT过程的总时间代价为：$O(nlgn)$。

- 是一种in-place**原地排序算法**

思考：堆排序算法与插入排序算法设计策略关系是否类似？

（减治法：不断减小被处理的问题规模）

思考题

① 以下算法：冒泡排序、堆排序、选择排序、插入排序、归并排序和快速排序，哪些是稳定的排序算法？为什么

② 上述哪些是原地(in-place)排序算法？(注意：快速排序，递归方式 & 栈；非递归方式)

③ 快速排序、插入排序、堆排序、选择排序，哪个算法所需要的交换次数最少？(选择排序：n次比较)

思考题

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

线性排序算法

内容提要：

- 排序问题
- 冒泡排序 & 选择排序
- 插入排序 & 希尔排序
- 快速排序 & 堆排序 《算法导论》第7章 & 第6章
- 线性排序算法 (计数排序、桶排序、基数排序) 《算法导论》第8章
- 排序算法比较

排序算法时间的下界

- 本节探讨排序算法所耗用的时间复杂度下限。
- 比较排序：排序结果中，各元素的次序基于输入元素间的比较，这类算法成为比较排序。
- 任何比较排序算法，排序n个元素时至少耗用$\Omega(n\lg n)$次比较，其时间复杂度至少为$\Omega(n\lg n)$
- 合并排序和堆排序是渐近最优的
- 但不使用比较为基础的排序算法，在某些情形下可以在$O(n)$的时间内执行完毕。

排序算法时间的下界

- 比较排序：排序结果中各元素的次序基于输入元素间的比较，这类算法是比较排序。最好的时间复杂度为$O(nlogn)$
- 线性排序（Linear Sort）：时间复杂度为 $O(n)$（突破了比较排序的最好时间复杂度），达到线性，排序不基于比较。线性排序有三种：计数排序、桶排序、基数排序。

排序算法	时间复杂度	空间复杂度	是否稳定
桶排序	O(n)	O(n)	稳定
计数排序	O(n)	O(n+k)	稳定
基数排序	O(k*n)	O(n)	稳定

排序算法时间的下界

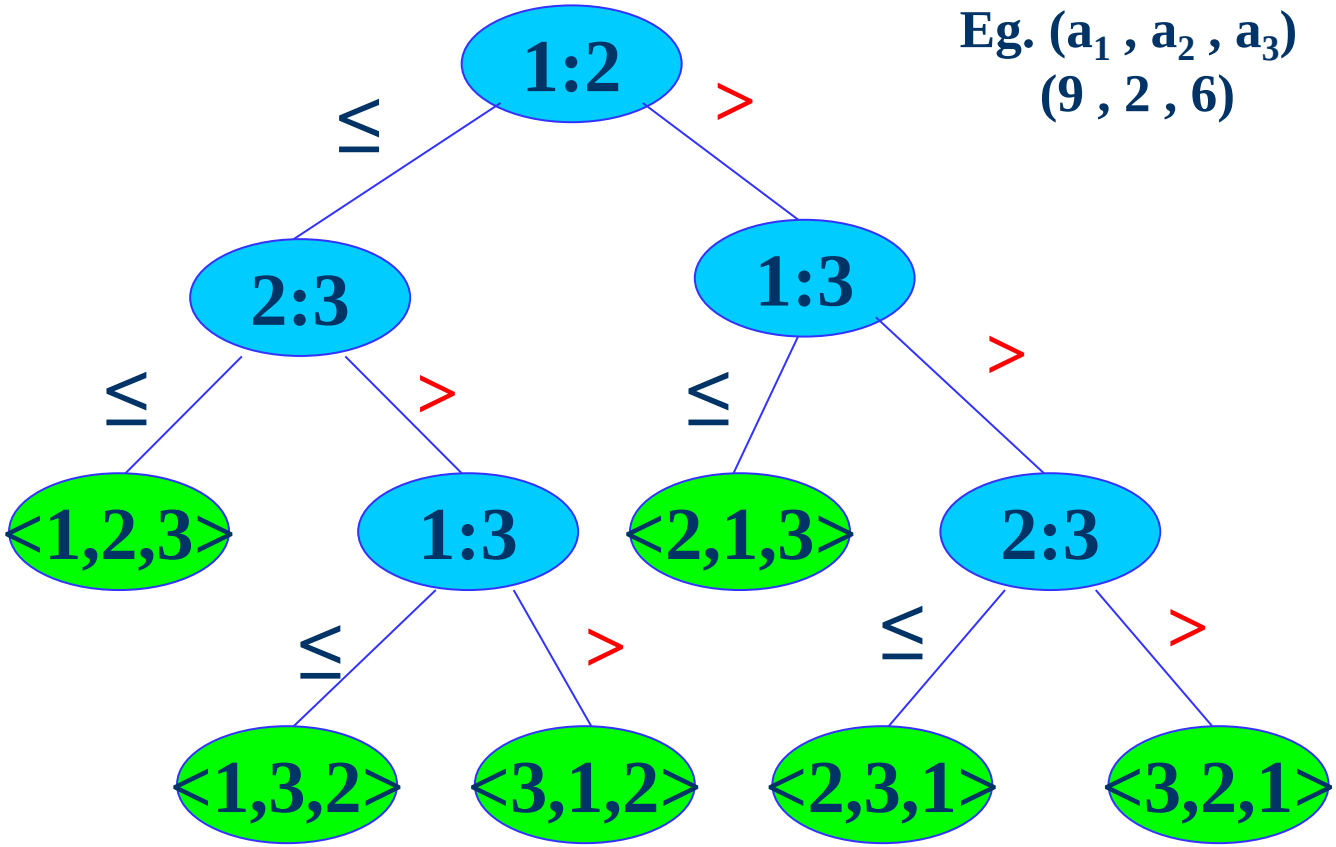
- 一个以元素比较为基础的排序算法可以按照比较的顺序建出一个决策树（Decision Tree）。
- 决策树是一棵满二叉树，表示某排序算法作用于给定输入所做的所有比较，忽略控制结构、数据移动等。

$$(\pi(1),\ \pi(2),\ \cdots,\ \pi(1))$$

决策树模型：

- 每个内结点注明i : j (1≤i, j≤n),每个叶结点注明排列 $(\pi(1), \pi(2), \cdots, \pi(n))$
- 排序算法的执行对应于遍历一条从树的根到叶节点的路径。
- 在每个内结点处做比较 $a_i \le a_j$, 内结点的左子树决定着 $a_i \le a_j$ 以后的比较，而右子树决定着 $a_i > a_j$ 以后的比较
- 到达一个叶结点时，排序算法就已确定了顺序 $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$
- 每一个从根节点到叶子结点的路径对应于比较排序算法的一次实际执行过程。
- 排序算法正确工作的必要条件：n个元素的 $n!$中排列都要作为决策树的一个叶子。

排序算法时间的下界



排序算法时间的下界

- 任何一个以元素比较为基础，对n个元素排序的排序算法，所对应的决策树的高度至少为$\Omega(n\log n)$。
  - 证明：因为可能有$n!$种可能的排序结果，故对应至少有$n!$个叶子结点, 而高度为h的二叉树最多有 2^h 个叶子结点。因此$h \geq \log(n!) \geq \Theta(n\log n)$。(后者由斯特林公式得证)
- 比较排序算法的最坏情况比较次数与其决策树的高度相等
- **定理1** 任意比较排序算法在最坏情况下，都需要做$\Omega(n\log n)$次比较。
- 堆排序和合并排序是渐近最优的排序算法 （它们的运行时间上界$O(n\log n)$与 定理1给出的最坏情况下界$\Omega(n\log n)$一致）
- 快速排序不是渐近最优的比较排序算法。但是，快速排序其执行效率平均而言较堆排序和合并排序还好。

计数排序

- 计数排序 (Counting Sort) 不是一个比较排序算法，该算法于1954年由 Harold H. Seward提出，不需要基于比较进行排序，但必须依赖于待排序集合中的元素性质的一些假设：如果所有待排序的元素均为整数，介于1到k之间。当 $k = O(n)$, 时间复杂度：$O(n+k)$

- 基本思想：对每一个输入元素x，统计出小于x的元素的个数。然后，根据这一信息直接把元素x放到它在最终输出数组中的位置上。

- 分为三个步骤：计算每个数出现了几次；求出每个数出现次数的 前缀和；利用出现次数的前缀和，从右至左计算每个数的排名。

计数排序

➢计数排序 (Counting Sort) 不是一个比较排序算法，该算法于1954年由 Harold H. Seward提出，不需要基于比较进行排序，但必须依赖于待排序集合中的元素性质的一些假设：如果所有待排序的元素均为整数，介于1到k之间。当 $k = O(n)$, 时间复杂度：$O(n+k)$

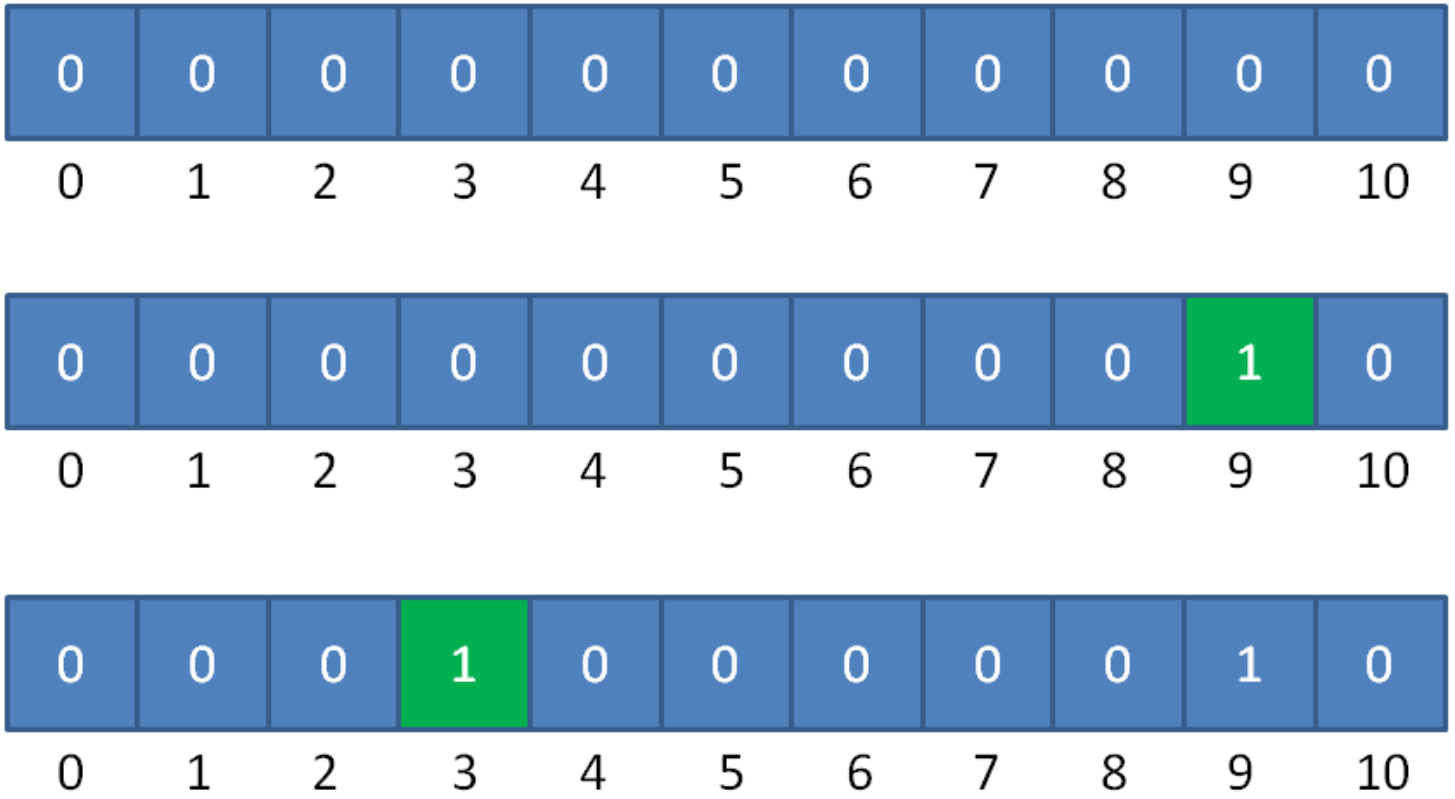
➢基本思想：对每一个输入元素x，统计出小于x的元素的个数。然后，根据这一信息直接把元素x放到它在最终输出数组中的位置上。

➢在计数排序算法的代码中，需三个数组：

- 输入数组A[1…n]
- 存放排序结果的数组B[1…n]
- 提供临时存储区的数组C[0…k]

计数排序

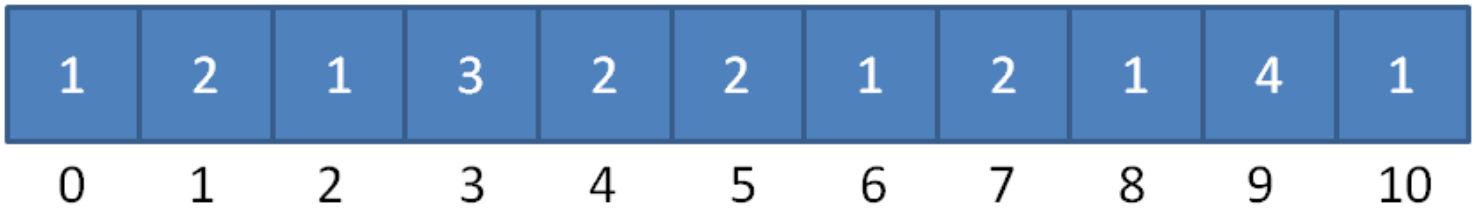
- **20个无序的随机整数：9，3，5，4，9，1，2，7，8，1，3，6，5，3，4，0，10，9，7，9**



计数排序

- **20个无序的随机整数：9，3，5，4，9，1，2，7，8，1，3，6，5，3，4，0，10，9，7，9**

数列遍历完毕时：

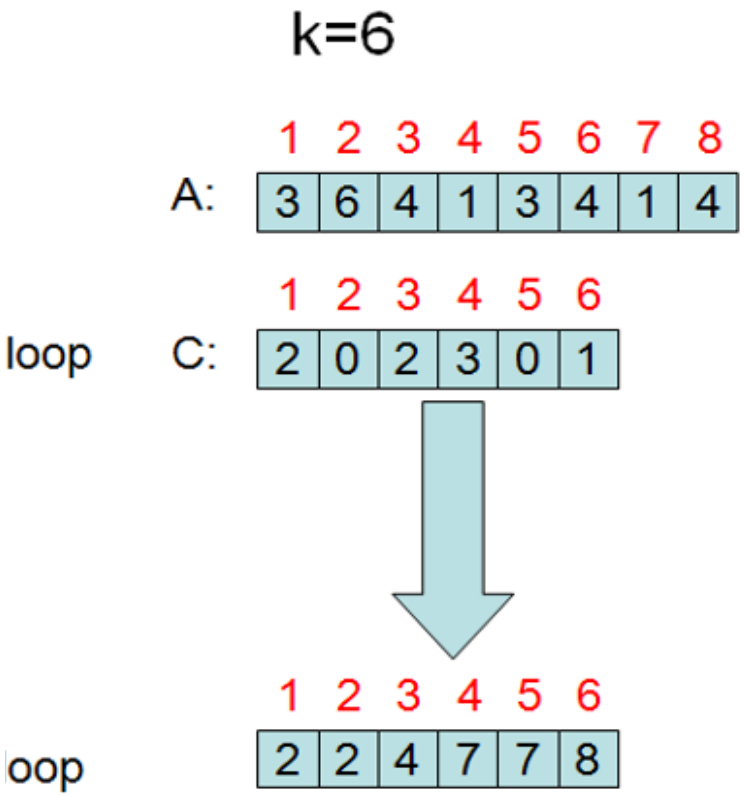


基于统计结果，直接遍历数组，输出数组元素的下标值：

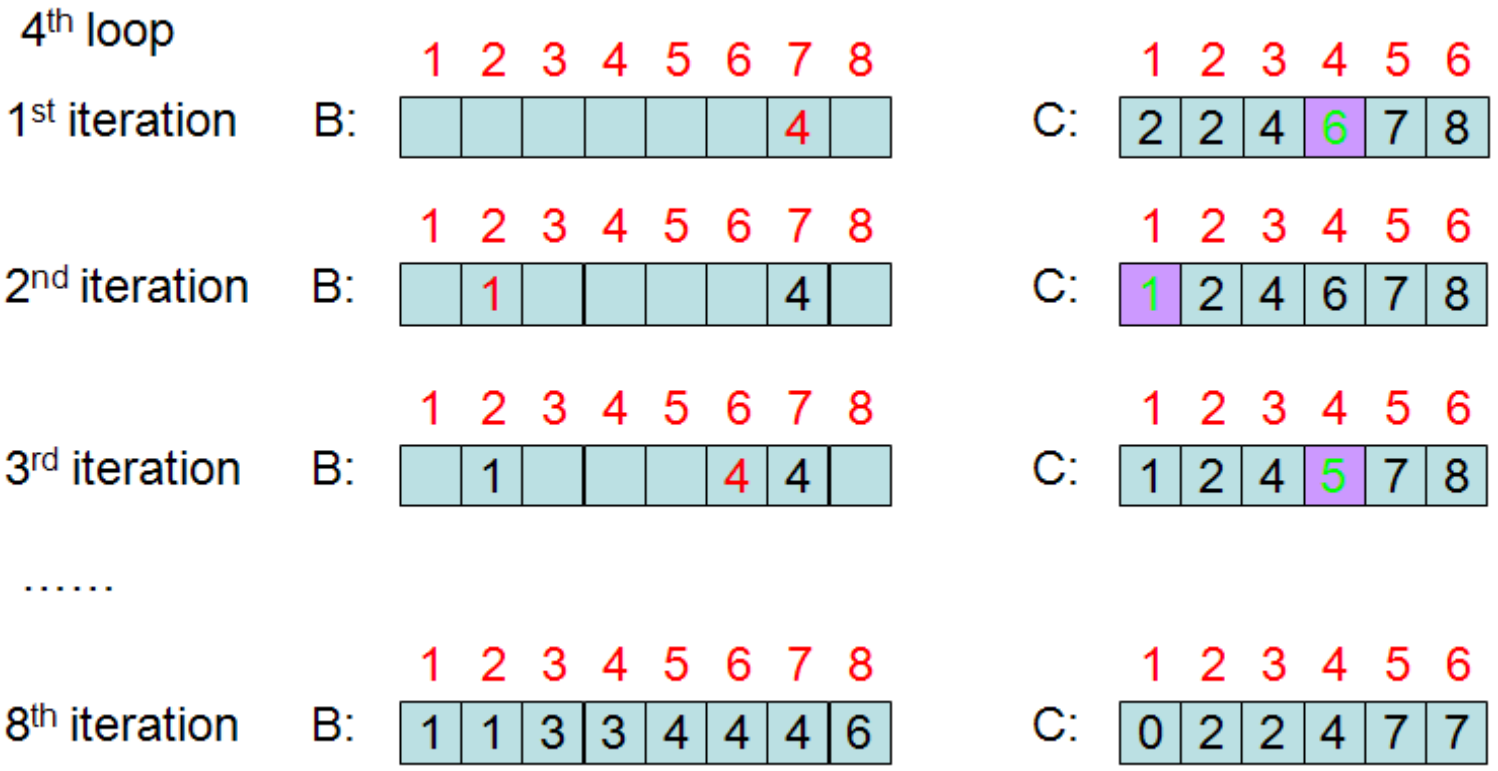
0，1，1，2，3，3，3，4，4，5，5，6，7，7，8，9，9，9，9，10

计数排序

```
Input: A[1..n], where A[j]∈{1,2,…,k}
Output:B[1..n], sorted
CountingSort(A,B,k)
{  for i = 0 to k
       do C[i]←0
   for j = 1 to length[A]
       do C[A[j]]←C[A[j]]+1
    //C[i]包含等于i的元素个数
   for i = 1 to k
       do C[i]←C[i]+C[i-1]
    //C[i]包含小于或等于i的元素个数
   for j = length[A] downto 1
       do B[C[A[j]]]←A[j]
          C[A[j]]←C[A[j]]-1
}
```



计数排序



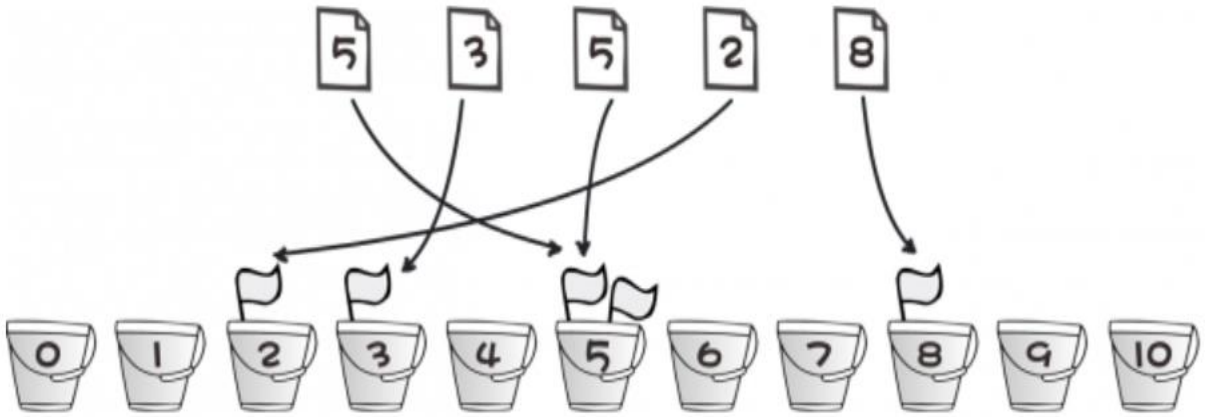
- 排序算法是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的次序 相同。
- 经常被当做基数排序算法的一个子过程 。

桶排序（Bucket Sort）

桶排序（Bucket Sort）是计数排序的泛化，当元素均匀地分布在某个区间时，可以在O(*n*)的时间内完成排序。假设输入数据由一个随机过程产生，该过程将元素均匀的分布在区间[0，1)上。

基本思想：

1) 把区间[0,1)划分成n个相同大小的子区间（称为桶）
2) 将n个输入数分布到各个桶中去
3) 先对各桶中元素进行排序，然后依次列出各桶中的元素



桶排序

```
BUCKET-SORT( A )
1   n ← length[A]
2   for i ← 1 to n
3         do insert A[i] into list B[floor(nA(i))]
4   for i ← 0 to n-1
5         do sort list B[i] with insertion sort
6   concatenate the lists B[0], B[1], …, B[n-1] together in order
```

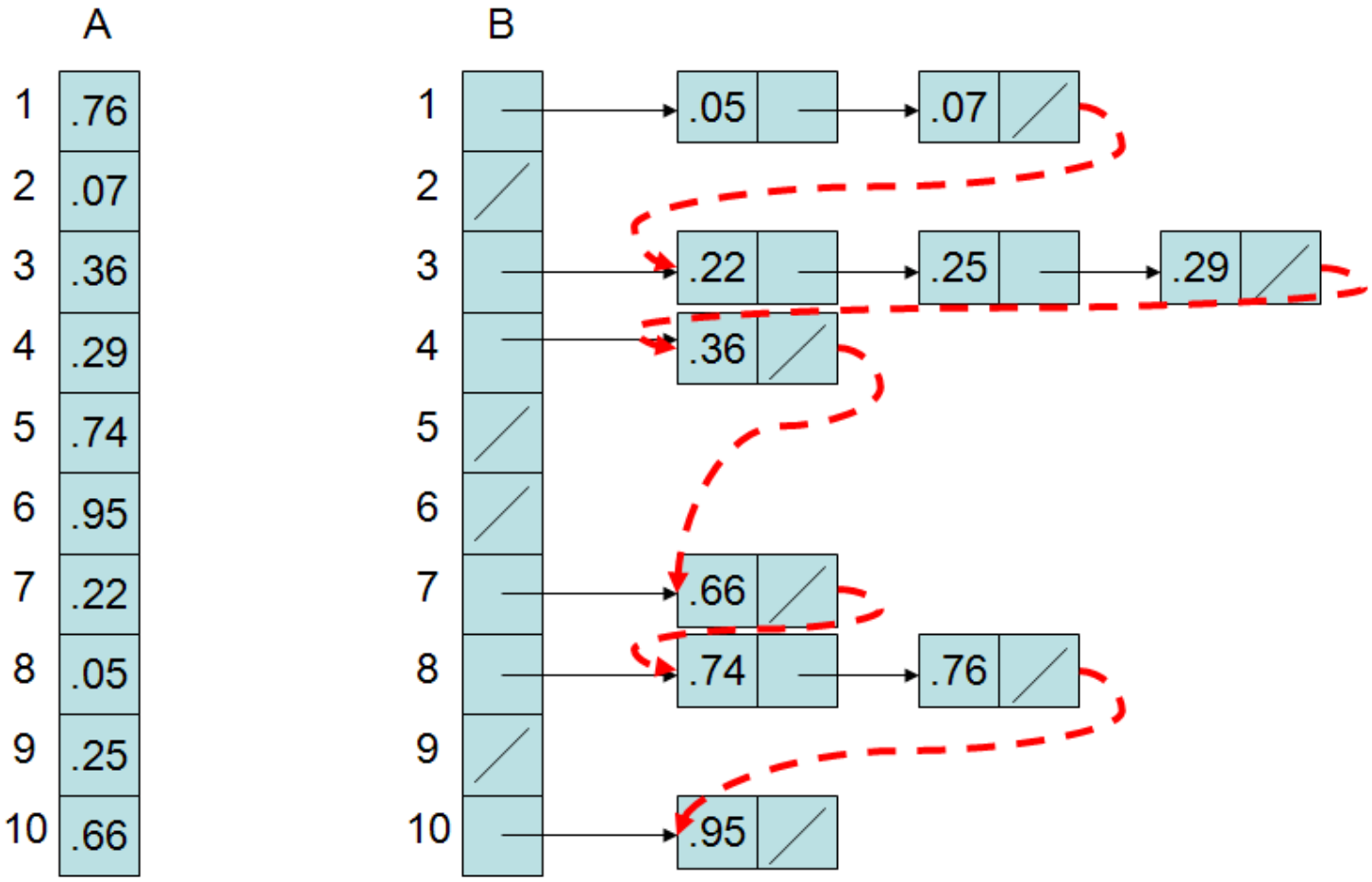
假定要排序的n个元素A[1..n]均是介于[0,1]之间的数值，桶排序步骤如下：

1）准备n个桶(bucket)，B[1..n]，将元素x依照x所在的区间放进对应的桶中：即第$\lceil nx \rceil$个桶 。

2）元素放进桶时，使用链表来存储，并利用插入排序法排序。

3）只要依序将链表串联起来，即可以得到已排序的n个元素。

桶排序


A
1 .76
2 .07
3 .36
4 .29
5 .74
6 .95
7 .22
8 .05
9 .25
10 .66
B
1 .05 .07
2
3 .22 .25 .29
4 .36
5
6
7 .66
8 .74 .76
9
10 .95

桶排序

时间复杂度分析：

- 假定分到第i个桶的元素个数是n_i。
- 最差情形：$T(n)=O(n)+\sum_{1\le i\le n}O(n_i^2)$
  $=O(n^2)$.

- 平均情形：$T(n)=O(n)+\sum_{1\le i\le n}O(E[n_i^2])$
  $=O(n)+\sum_{1\le i\le n}O(1)$
  $=O(n)$
- $E[n_i^2]=\Theta(1)$ 的证明请参考课本！

基数排序（Radix Sort）

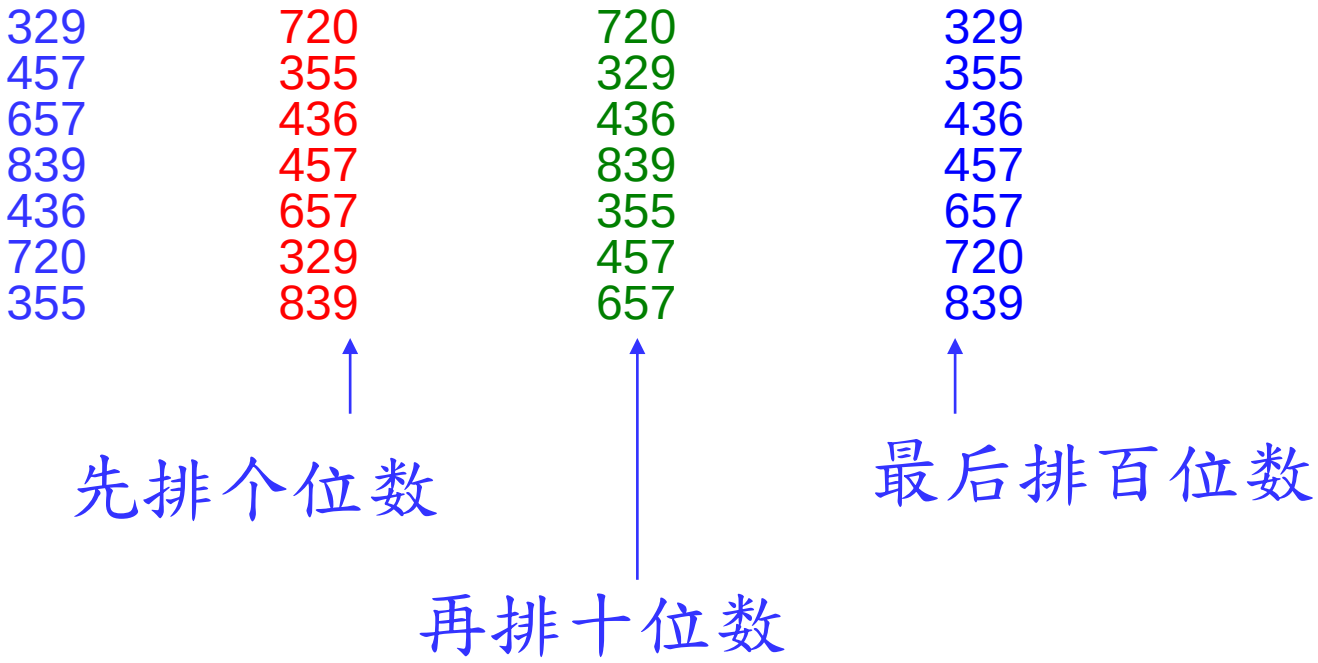
➢**Radix Sort(基数排序)：**假设所有待排序元素均为整数，至多*d*位。先按最低有效位进行排序，再按次低有效位排序，重复这个过程，直到对所有的d位数字都进行了排序。

➢基数排序关键是按位排序要稳定。

➢算法

```
Radix-Sort(A,d)
{for i = 1 to d
     do use stable sort to sort A on digit i
}
```

基数排序


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
839
先排个位数
再排十位数
最后排百位数

基数排序

```
Radix-Sort(A, d)
{ for i = 1 to d
      do use stable sort to sort A on digit i
}
```

- 此处需应用稳定的排序算法，如果使用Counting Sort则每次迭代只需要$\Theta(n+k)$的时间（假设每位数可以取k种可能的值）

- 因此，总共花费$O(d(n+k))$的时间。

- 如果 d是常数，$k = O(n)$，则基数排序能在线性时间内完成排序。

排序算法的比较

内容提要：

- 排序问题
- 选择排序&堆排序
- 插入排序&希尔排序
- 冒泡排序&快速排序
- 线性时间排序
- 排序算法比较

各种排序算法评价

- 排序算法之间的比较主要考虑以下几个方面：
  - ✓ 算法的时间复杂度
  - ✓ 算法的辅助空间
  - ✓ 排序的稳定性
  - ✓ 算法结构的复杂性
  - ✓ 参加排序的数据的规模
  - ✓ 排序码的初始状态

➢ 稳定的排序算法：冒泡排序、插入排序、归并排序和基数排序。

➢ 不稳定的排序算法：选择排序、快速排序、希尔排序、堆排序。

各种排序算法评价

- 当数据规模n较小时，n^2和$n\log_2 n$的差别不大，则采用简单的排序方法比较合适
  - ✓ 如直接插入排序或直接选择排序等
  - ✓ 由于直接插入排序法所需记录的移动较多，当对空间的要求不多时，可以采用表插入排序法减少记录的移动

- 当文件的初态已基本有序时，可选择简单的排序方法
  - ✓ 如直接插入排序或起泡排序等

各种排序算法评价

- 当数据规模n较大时，应选用速度快的排序算法
  - 快速排序法最快，被认为是目前基于比较的排序方法中最好的方法
  - 当待排序的记录是随机分布时，快速排序的平均时间最短。但快速排序有可能出现最坏情况，则快速排序算法的时间复杂度为$O(n^2)$，且递归深度为n，即所需栈空间为O(n)

比较排序 vs. 线性时间排序

- 从整体上来说，计数排序，桶排序都是非基于比较的排序算法，而其时间复杂度依赖于数据的范围，桶排序还依赖于空间的开销和数据的分布。而基数排序是一种对多元组排序的有效方法，具体实现要用到计数排序或桶排序。

- 相对于快速排序、堆排序等基于比较的排序算法，计数排序、桶排序和基数排序限制较多、灵活性差。但这三种线性排序算法充分利用了待排序数据的特性，能够达到线性时间。
-
- **平方阶 ($O(n^2)$) 排序**：各类简单排序，如插入、直接选择和冒泡排序。

- **线性对数阶 (O(nlogn)) 排序**： 快速排序、堆排序和归并排序；

- **O(n+k)) 排序**，k是介于 0 和 1 之间的常数。如希尔排序

- **线性阶 (O(n)) 排序**: 基数排序，此外还有桶排序、箱排序。

思考

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定