算法设计与分析

LECTURE 3

Lecture 3

排序

Outline

- 排序算法
  - 快速排序
  - 合并排序
  - 堆排序

Lecture 3

快速排序

Outline

- 排序问题
- 插入排序
  - 插入排序分析
- 快速排序
  - 快速排序分析

排序问题

- 排序
  - 比如，将所有学生的成绩排序

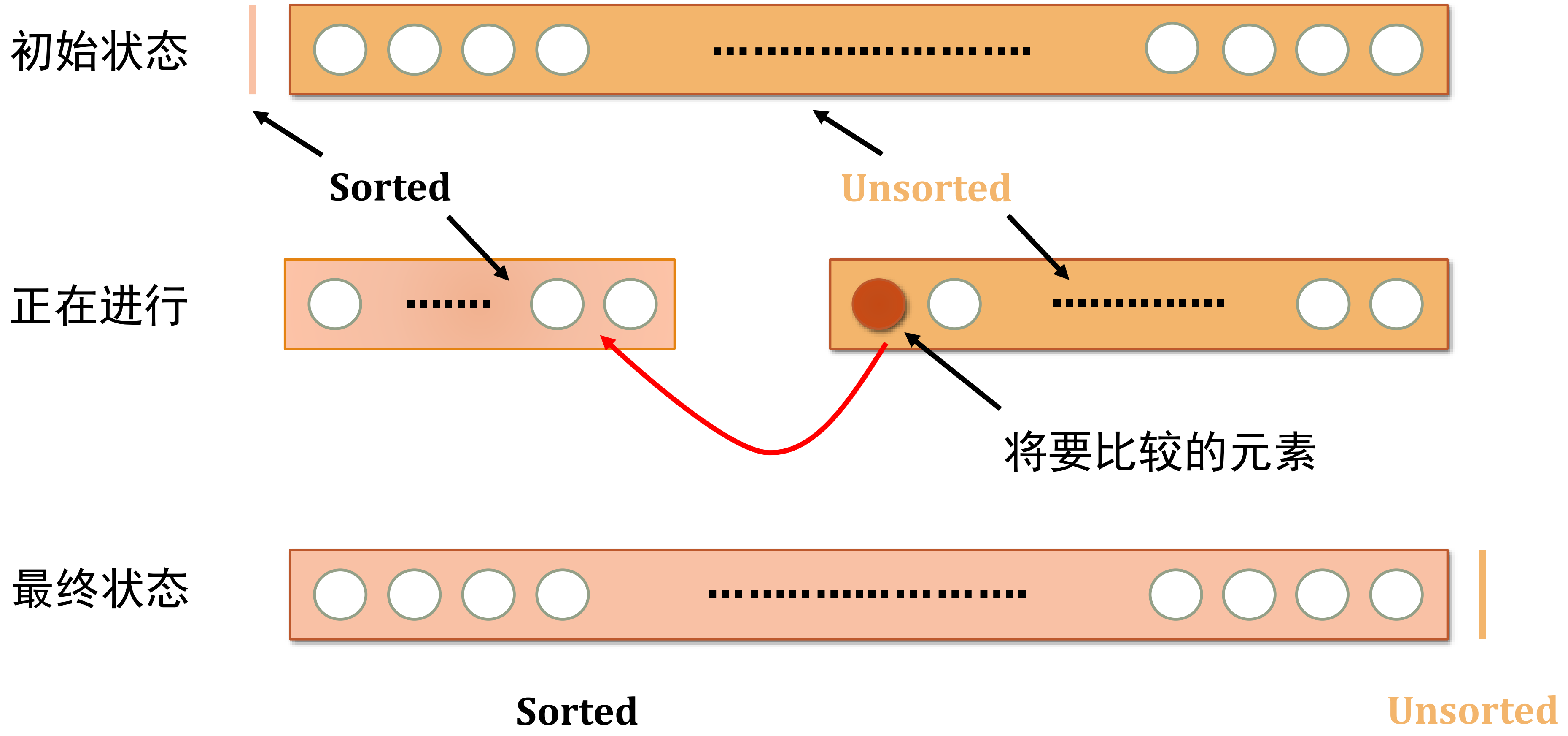
- 基本假设
  - 排什么？
    - 问题规模 n, 元素各不相同
  - 什么顺序排？
    - 递增：从小到大
  - 输入是什么？
    - 每个可能的输入同概率出现

基于比较的排序

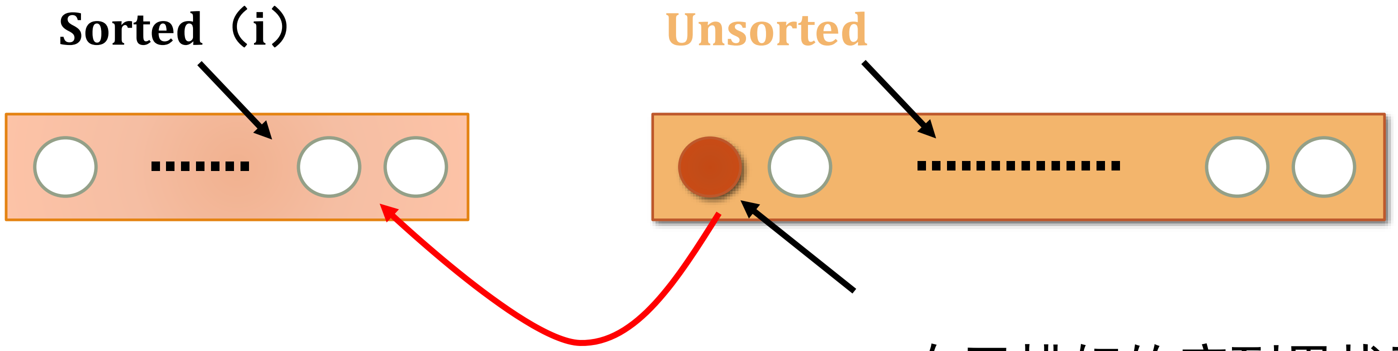
- 关键操作
  - 两个元素的比较

- 计算代价
  - 关键操作的个数（元素比较）

插入排序



最坏情况分析

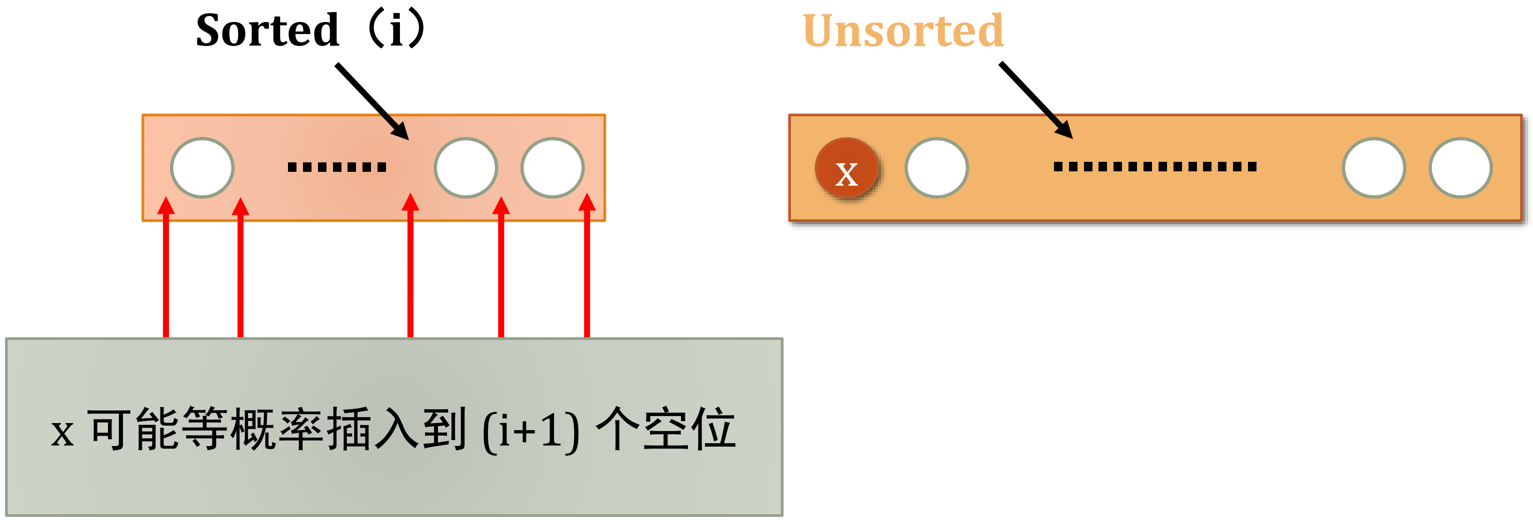


在已排好的序列里找到合适的位置，最坏情况需 i 次比较

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

$\boldsymbol{\Theta(n^2)}$

平均情况分析



- 基本假设
  - n, 元素各不相同
  - 每个可能的输入同概率出现

平均情况分析

- 元素 x 在所有 i+1 个可能被插入的位置中等概率出现，插入元素 x 所需比较次数的期望值为：

$$c_{i+1} = \underbrace{\frac{1}{i+1}\sum_{j=1}^{i} j}_{\text{右边}i\text{个位置}} + \underbrace{\frac{1}{i+1} i}_{\text{最左边1个位置}}$$

$$= \frac{i}{2} + 1 - \frac{1}{i+1}$$

- 平均情况时间复杂度为：

$$A(n) = \sum_{i=1}^{n-1} c_{i+1}$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - (\ln n + \gamma + \epsilon(n) - 1)$$

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

插入排序的不足

- 插入排序基于遍历来实现的，它的最坏和平均情况时间复杂度均为 $O(n^2)$

思考：插入排序改进的空间在哪里？哪些计算是多余的？如何减少？

逆序对

- 为了可以通过更有力的数学工具来深入分析插入排序的不足，这里引入逆序对的概念。

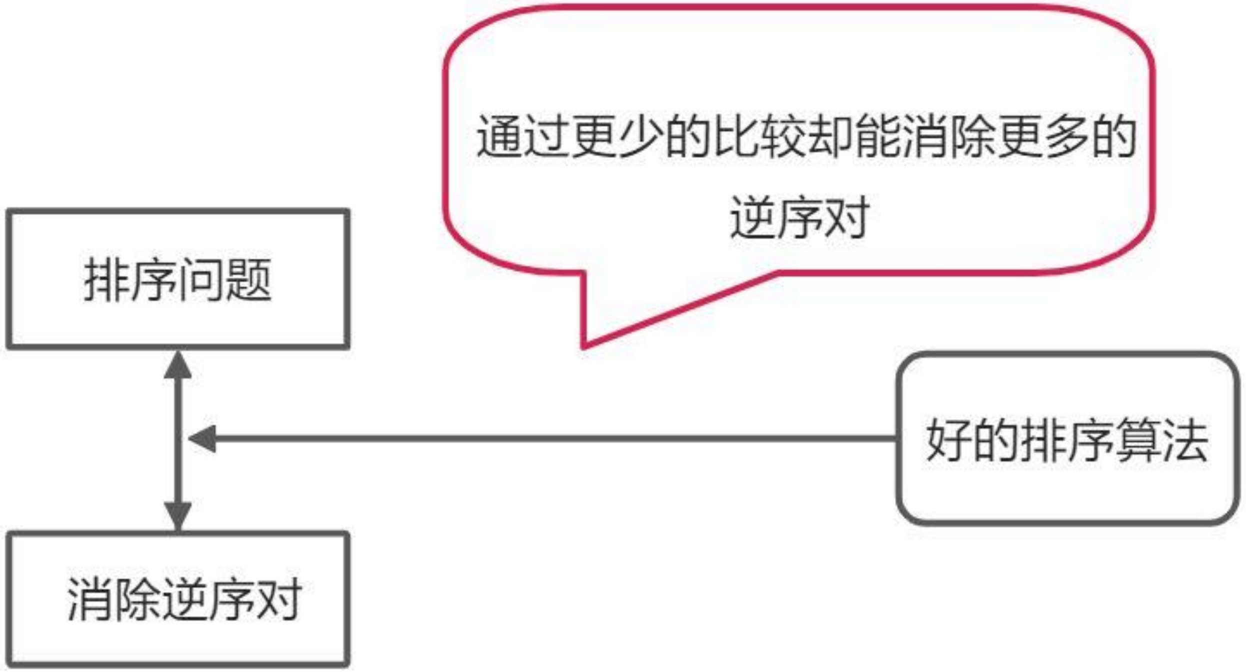
逆序对与排序

- **定义**（逆序对）给定一组各不相同的两两可比较的元素。对于这些元素的一个排列 $\{a_1, a_2, \ldots, a_n\}$ 而言，定义二元组 (a_i, a_j) 为一个逆序对，如果 $i < j$，且 $a_i > a_j$。

基于逆序对的概念，可以等价地换一个视角来描述排序问题和排序算法。

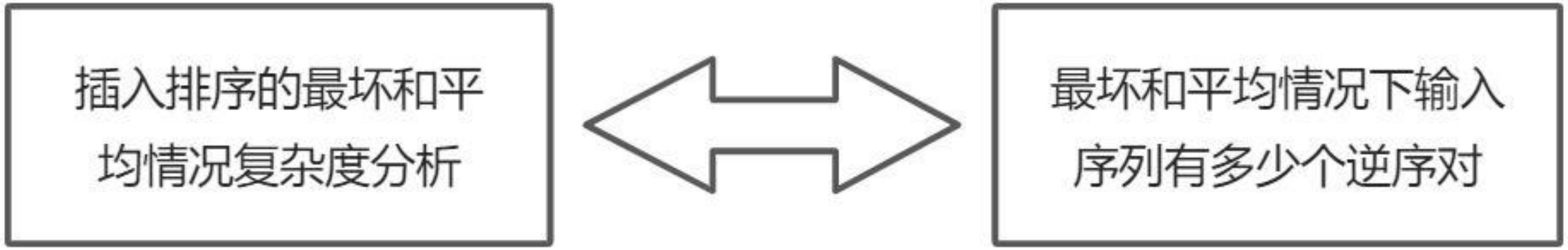
逆序对与排序

- 对于输入的待排序元素的序列，它包含若干个逆序对，而排序算法就是通过元素的比较，调整元素的位置，消除输入序列中的逆序对。



插入排序分析—逆序对角度

- 从**消除逆序对**的角度来看插入排序的一个关键特征：每次总是将相邻的元素进行比较，至多只能消除序列中的一个逆序对。



插入排序分析—逆序对角度

- 最坏情况下：对于一个从大到小排列的输入序列，其中任意两个元素均构成逆序对，所以最坏情况下输入序列中可能有 $\binom{n}{2} = O(n^2)$ 个逆序对。

- 插入排序一次比较只能消除一个逆序对。

- 插入排序最坏情况时间复杂度是 $O(n^2)$ 。

插入排序分析—逆序对角度

- 平均情况下：假设所有可能的输入等概率地出现，平均情况下逆序对的个数就是输入序列中所有二元组个数的一半，即 $\frac{1}{2}\binom{n}{2} = O(n^2)$。

- 输入序列 $\{a_1, a_2, \ldots, a_n\}$ 和转置 $\{a_n, a_{n-1}, \ldots, a_1\}$.

- 输入序列的任意一对元素 (a_i, a_j)，在转置序列中对应的元素为 (a_j, a_i)。

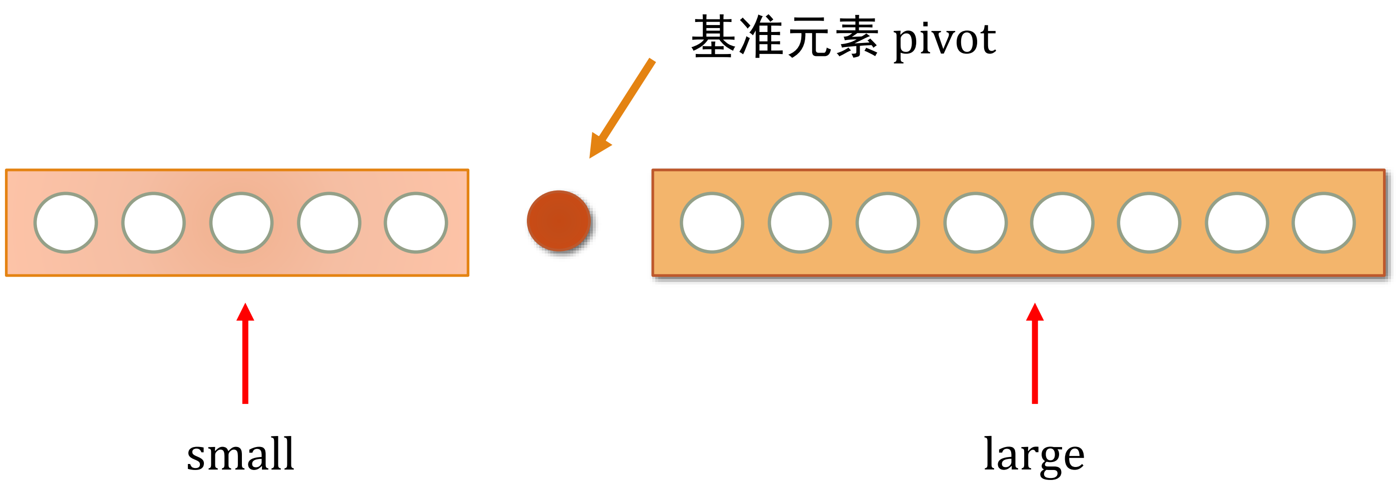
- 这两个二元组，必然只有一个逆序对。

插入排序分析—逆序对角度

基于以上分析，插入排序的主要问题是一次比较至多只能消除一个逆序对。**后续改进的出发点就是如何更“聪明”地进行元素的比较，使得一次比较能够消除更多的逆序对。**

快速排序

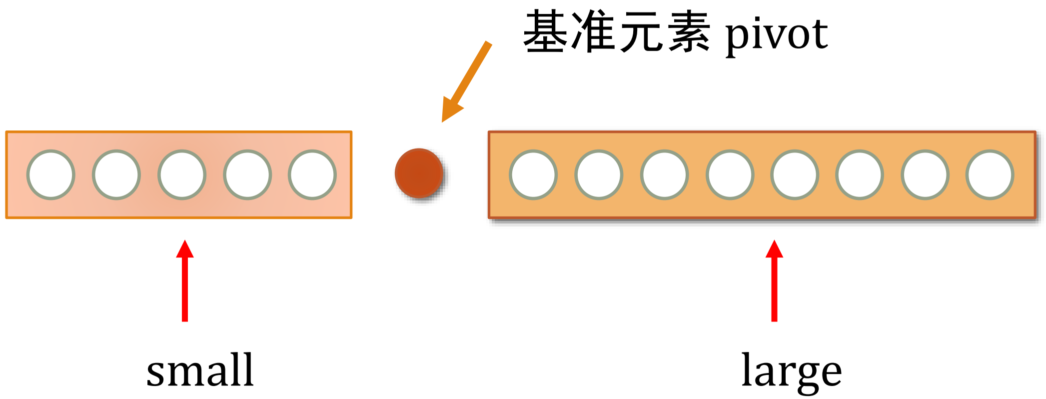
- 把要排序的数组分成两部分："small"和"large"；再递归实现
- 小的元素尽量放左边，大的元素尽量放右边



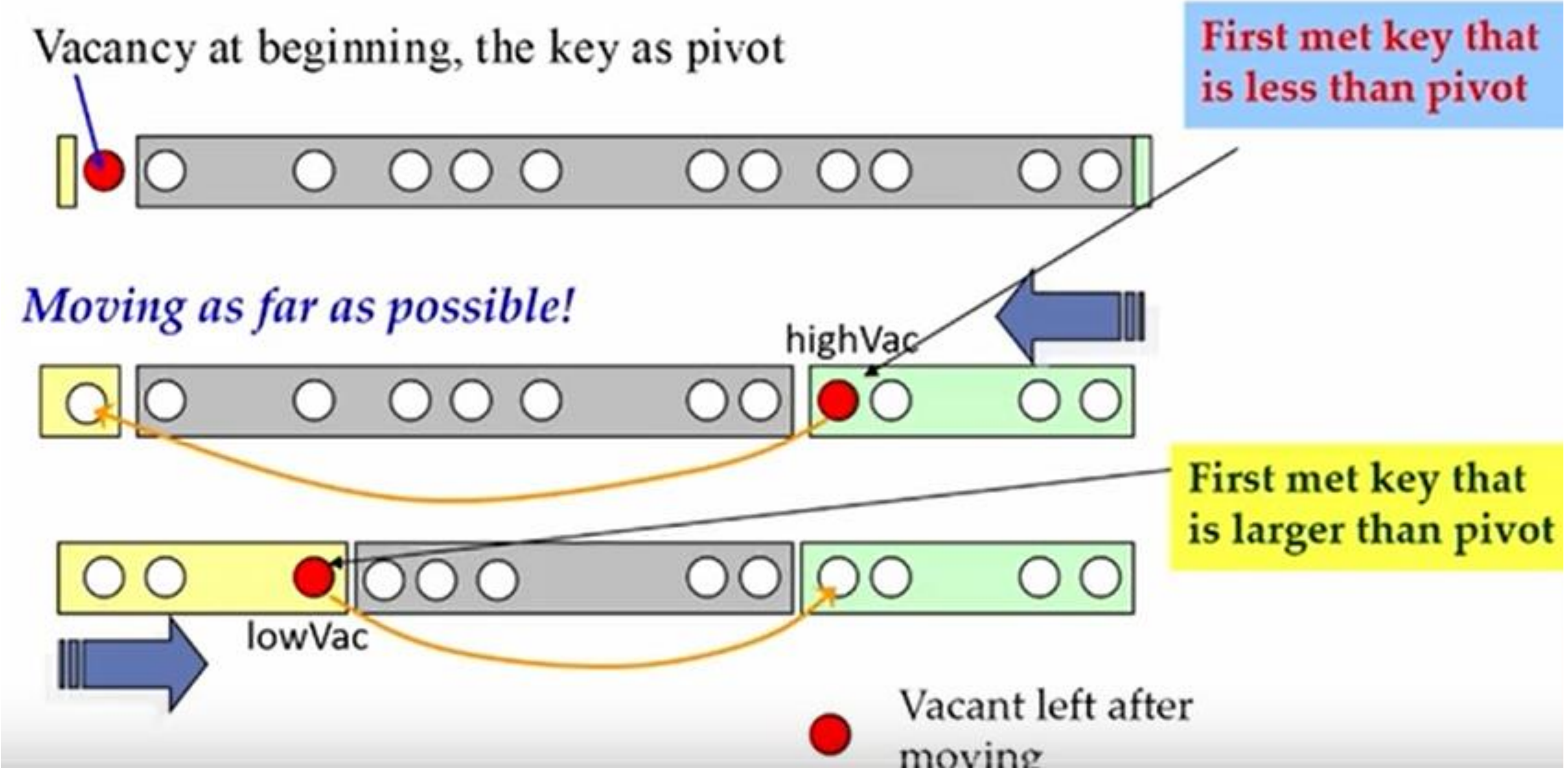
快速排序——分治算法

- Divide
  - “small” 和 “large”
- Conquer
  - 递归排序 “small” 和 “large”
- Combine

难分易合

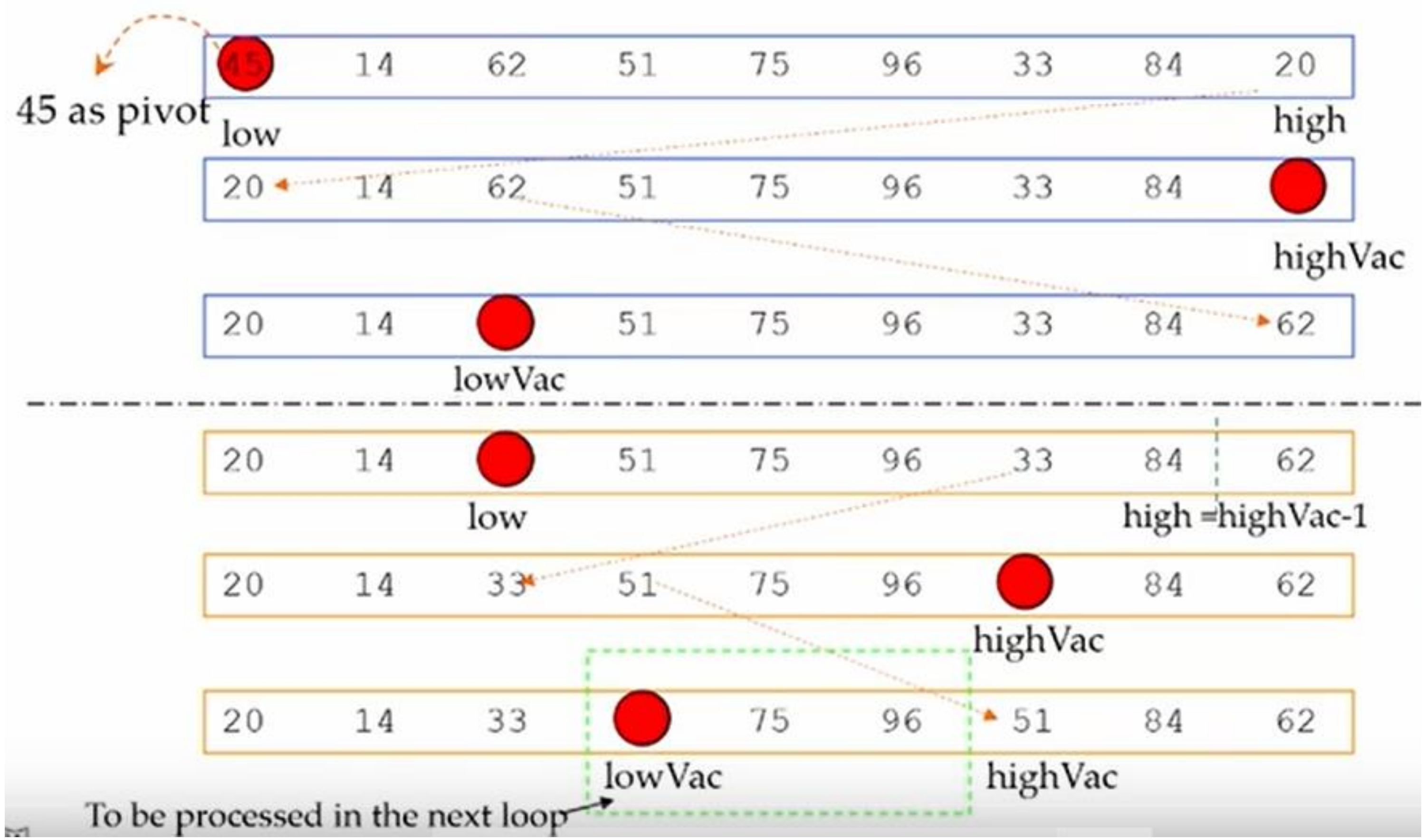


快速排序



Vacancy at beginning, the key as pivot
First met key that is less than pivot
Moving as far as possible!
highVac
First met key that is larger than pivot
lowVac
Vacant left after moving

快速排序



45 as pivot
45 14 62 51 75 96 33 84 20
low
high
20 14 62 51 75 96 33 84
highVac
20 14 51 75 96 33 84 62
lowVac
20 14 51 75 96 33 84 62
low
high =highVac-1
20 14 33 51 75 96 84 62
highVac
20 14 33 75 96 51 84 62
lowVac
highVac
To be processed in the next loop

快速排序

i	j							
	50	4	30	15	21	12	3	18

i		j						
	50	4	30	15	21	12	3	18

i		j					
4	50	30	15	21	12	3	18

i			j				
4	50	30	15	21	12	3	18

	i			j			
4	15	30	50	21	12	3	18

	i				j		
4	15	30	50	21	12	3	18

		i				j	
4	15	12	50	21	30	3	18

			i				
4	15	12	3	21	30	50	18

			i				
4	15	12	3	18	30	50	21

最坏情况时间复杂度

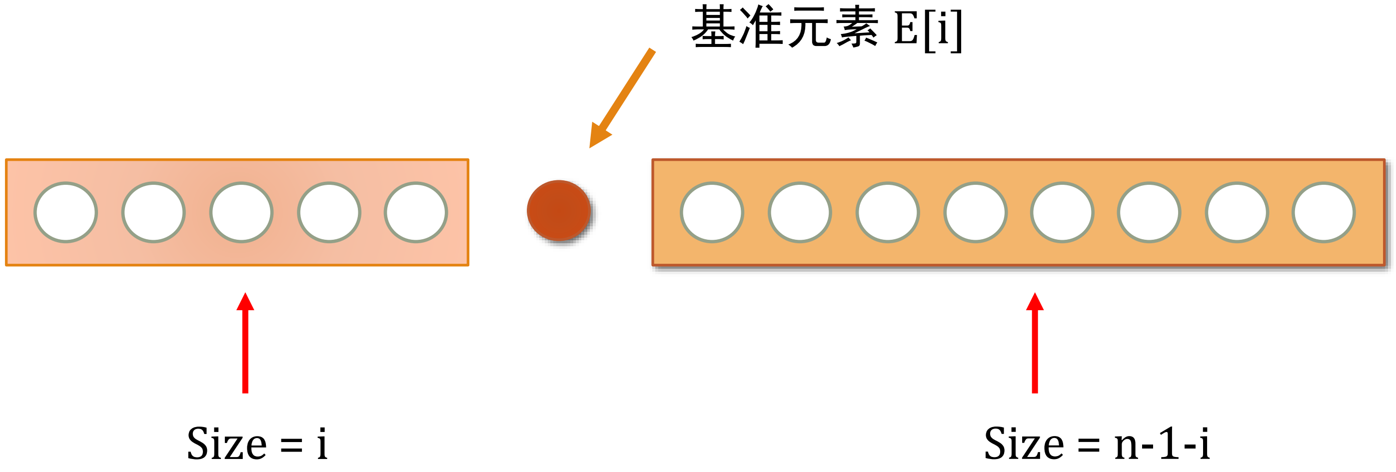
- 如果基准元素是最小的，“large”部分仍旧保留 k-1个元素，“small”为空。

- 如果数组元素已是排好序的，那么比较次数为：

$$\sum_{k=2}^{n}(k-1)=\frac{n(n-1)}{2}\in O(n^2)$$

平均情况时间复杂度

- $i \in \{0,1,2,\ldots,n-1\}$，每个值概率为 1/n



- $A(n) = (n-1) + \sum_{i=0}^{n-1} \frac{1}{n}[A(i) + A(n-1-i)]$
- $A(1) = A(0) = 0$

平均情况时间复杂度

- $\sum_{i=0}^{n-1} A(i) = \sum_{i=0}^{n-1} \frac{1}{n} A(n-1-i)$
- $A(n) = (n-1) + \frac{2}{n} \sum_{i=0}^{n-1} A(i)$

- 解递归方程
  - Guess and prove
  - 直接求解

平均情况时间复杂度

- 假设每次划分的结果都是完全均衡的。
- $A(n) = 2A\left(\frac{n}{2}\right) + O(n)$
- 根据 Master 定理： $A(n) = o(n \log n)$

平均情况时间复杂度

We have: $$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \quad \textit{and}$$

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

Combining the 2 equations in some way, we can remove all A(i) for i=1,2,…,n-2

$$\begin{aligned} & nA(n) - (n-1)A(n-1) \\ & = n(n-1) + 2\sum_{i=1}^{n-1} A(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2} A(i) \\ & = 2A(n-1) + 2(n-1) \end{aligned}$$

$$So,\ nA(n) = (n+1)A(n-1) + 2(n-1)$$

空间复杂度

- 原地排序 O(1)

- 考虑递归栈代价
  - 最坏情况，递归深度为 n-1
  - 递归栈最大为 $\Theta(n)$

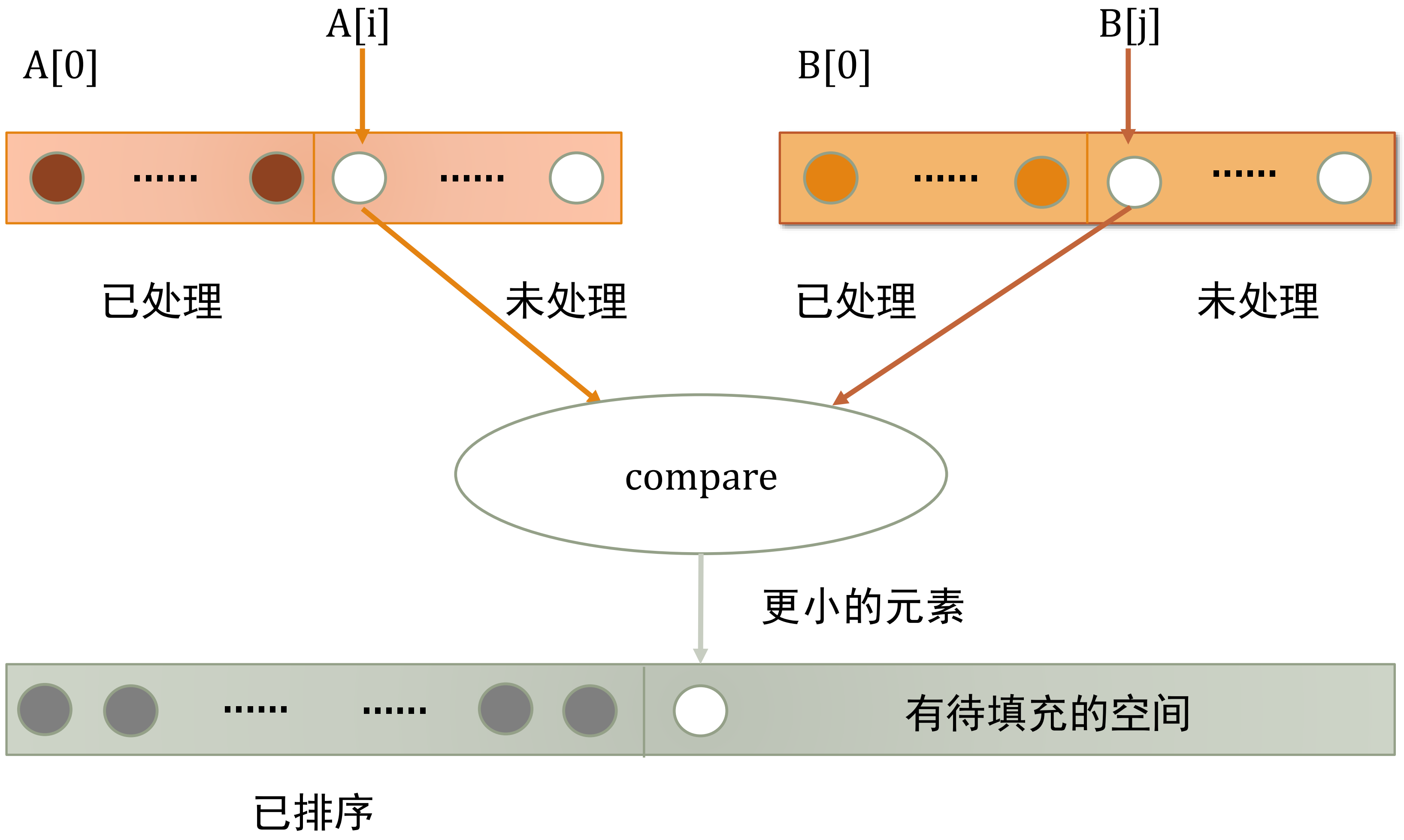
Outline

Lecture 3

合并排序

- 合并排序
  - 最坏情况分析
- 排序下界
  - 最坏情况
  - 平均情况

合并排序



最坏情况时间复杂度

- 最坏情况是最后一次比较元素是 A[k-1] 和 B[m-1]
  - 每次比较，至少一个元素可以放到已排序序列。
  - 之后，这个元素不再比较。
  - 完成最后一次比较，最少需要 n-1 次比较。

- 最坏情况下，需要 n-1 次比较

- 例：A[0]<B[0]<A[1]<B[1]<...<A[i]<B[i]<A[i+1]...<A[k-1]<B[m-1]

合并排序分析

- 递归方程：$W(n) = 2W\left(\frac{n}{2}\right) + n - 1$
- 根据 Master 定理：$W(n) \in \Theta(n \log n)$

合并排序应用

- 逆序对计数
  - 蛮力算法：$O(n^2)$
  - 可以用分治策略吗？
    - $\mathrm{O}(n\log n)$ => combination in $O(n)$

剑指 Offer 51. 数组中的逆序对

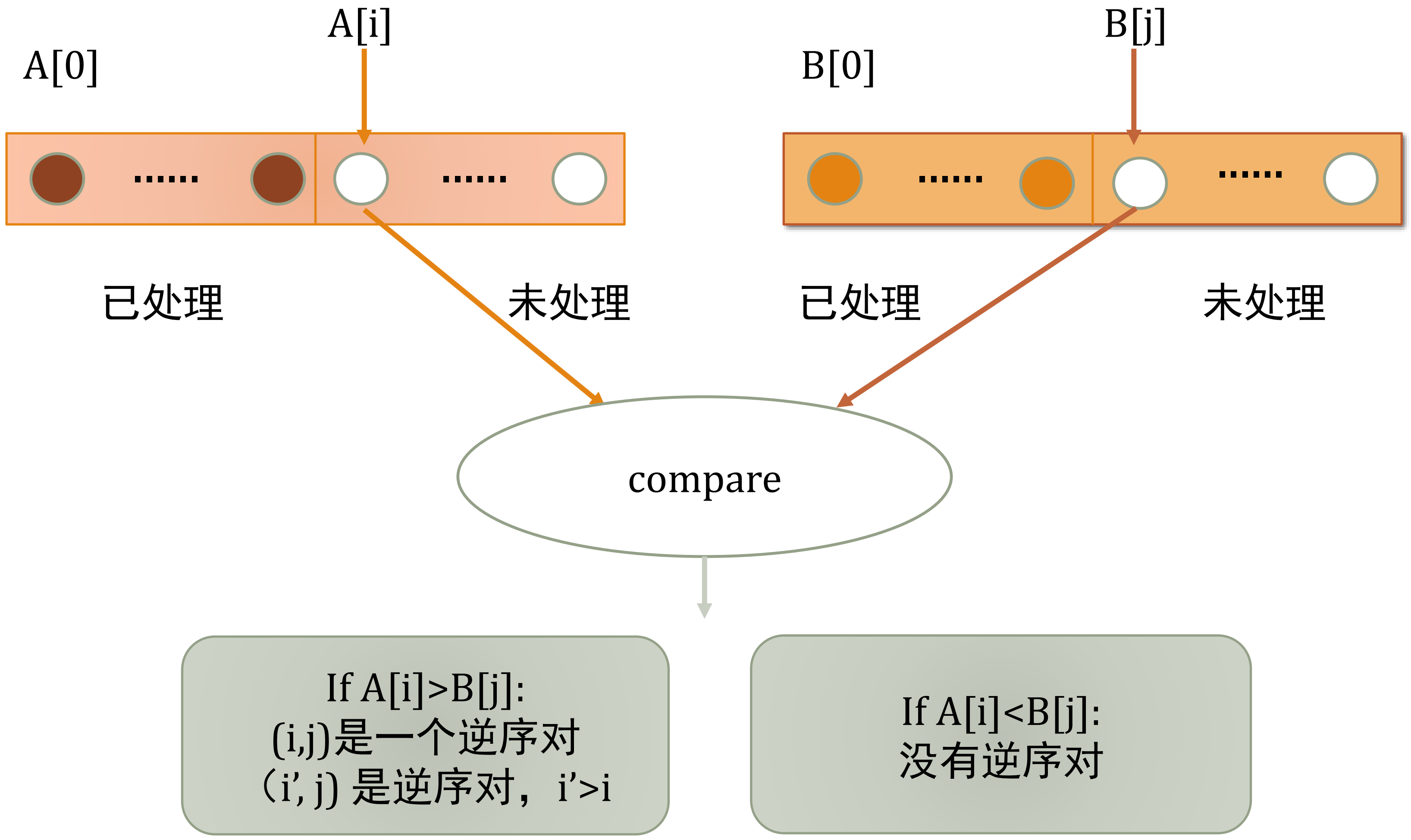
难度 困难 530 收藏 分享 切换为英文 接收动态 反馈

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

```
输入: [7,5,6,4]
输出: 5
```

合并排序应用


A[i]
A[0]
B[j]
B[0]
……
……
……
……
已处理
未处理
已处理
未处理
compare
If A[i]>B[j]:
(i,j)是一个逆序对
（i', j) 是逆序对，i'>i
If A[i]<B[j]:
没有逆序对

合并排序应用

- 最大子序列
- 找频繁元素
- 整数相乘

划分子问题

线性时间合并

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = O(n \log n)$$

Lecture 3

堆排序

Outline

- 堆
- 堆排序

- 修堆
- 堆构建

- 堆排序复杂性

堆

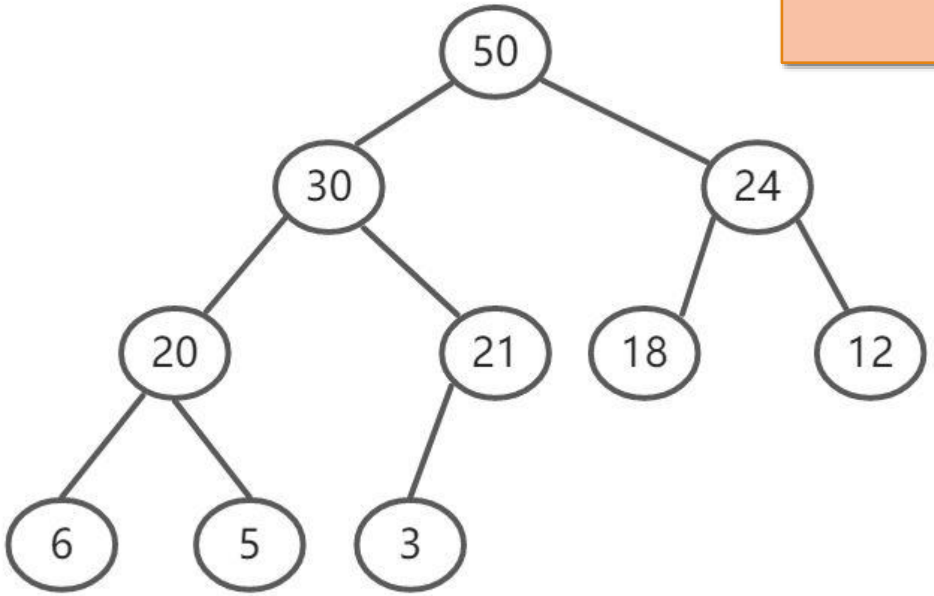
堆定义

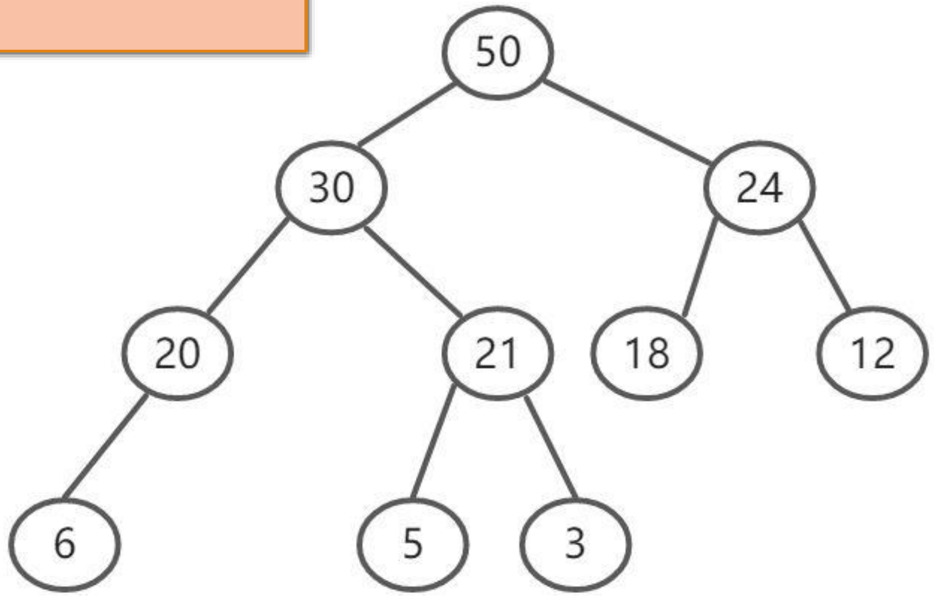
- 堆结构特性：
  - 要么是完美二叉树
  - 要么比完美二叉树在底层（深度最大的层）少若干节点
  - 底层节点从左向右紧挨着依次排列

- 堆偏序特性：
  - 父节点的值大于所有子节点的值
  - 子节点之间的大小关系无要求

堆：示例

最大值在根节点



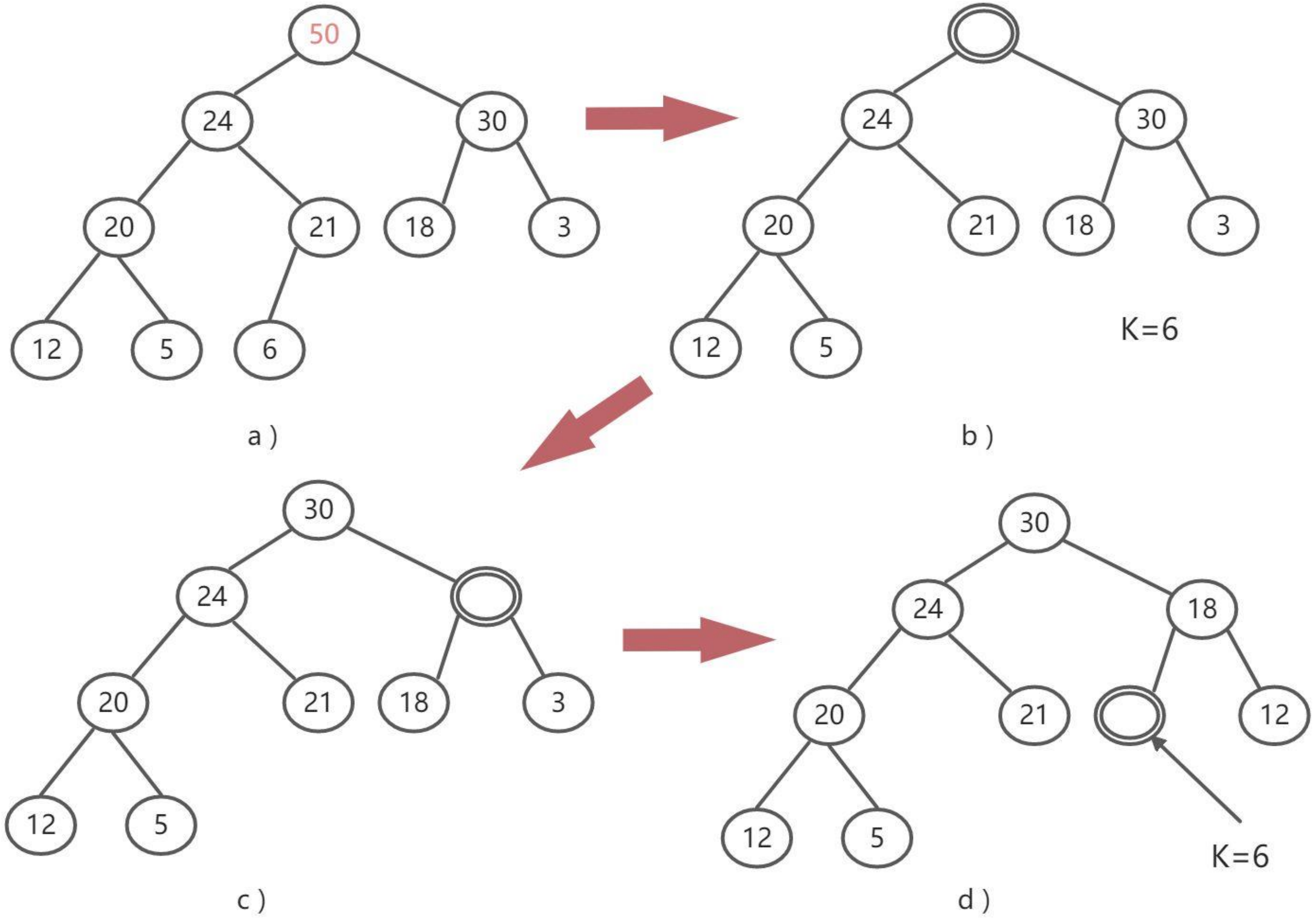


堆排序

```
heapSort(E,n)
    Construct H from E, the set of n elements to be sorted;
    for (i=n;i≥1;i--)
        curMax = getMax(H);
        deleteMax(H);
        E[i] = curMax
```

deleteMax(H):
    fixHeap(H,K)

堆修复 FixHeap


50
24
30
20
21
18
3
12
5
6
a)
24
30
20
21
18
3
12
5
K=6
b)
30
24
20
21
18
3
12
5
c)
30
24
18
20
21
12
12
5
K=6
d)

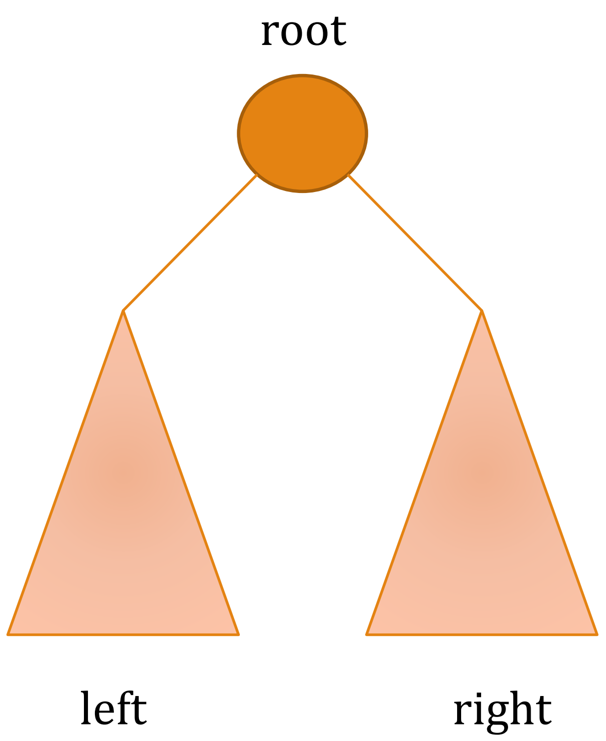
堆修复—最坏情况分析

- 一次修复最多两次比较
- 最多修复次数不超过堆的高度
- 堆修复代价—$O(\log n)$

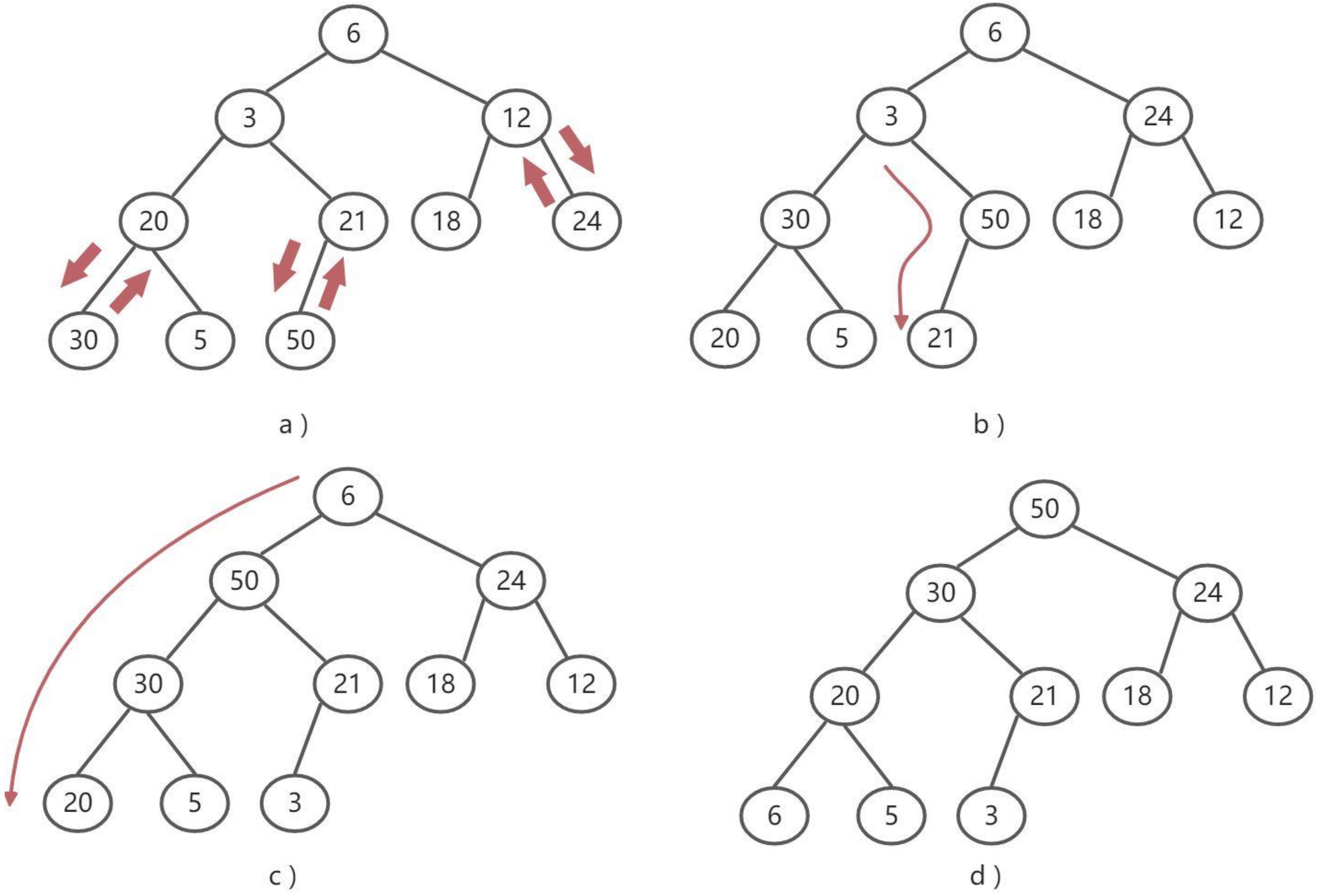
堆构建

- 堆结构 （递归）
- 堆偏序 （fixHeap(H, root(H))）

```
void constructHeap(H)
  if (H is not a leaf)
    constructHeap(left subtree of H);
    constructHeap(right subtree of H);
    Element K=root(H);
    fixHeap(H,K)
  return
```



堆构建



6
3
12
20
21
18
24
30
5
50
a)
6
3
24
30
50
18
12
20
5
21
b)
6
50
24
30
21
18
12
20
5
3
c)
50
30
24
20
21
18
12
6
5
3
d)

堆构建—复杂度分析

- 递归方程：

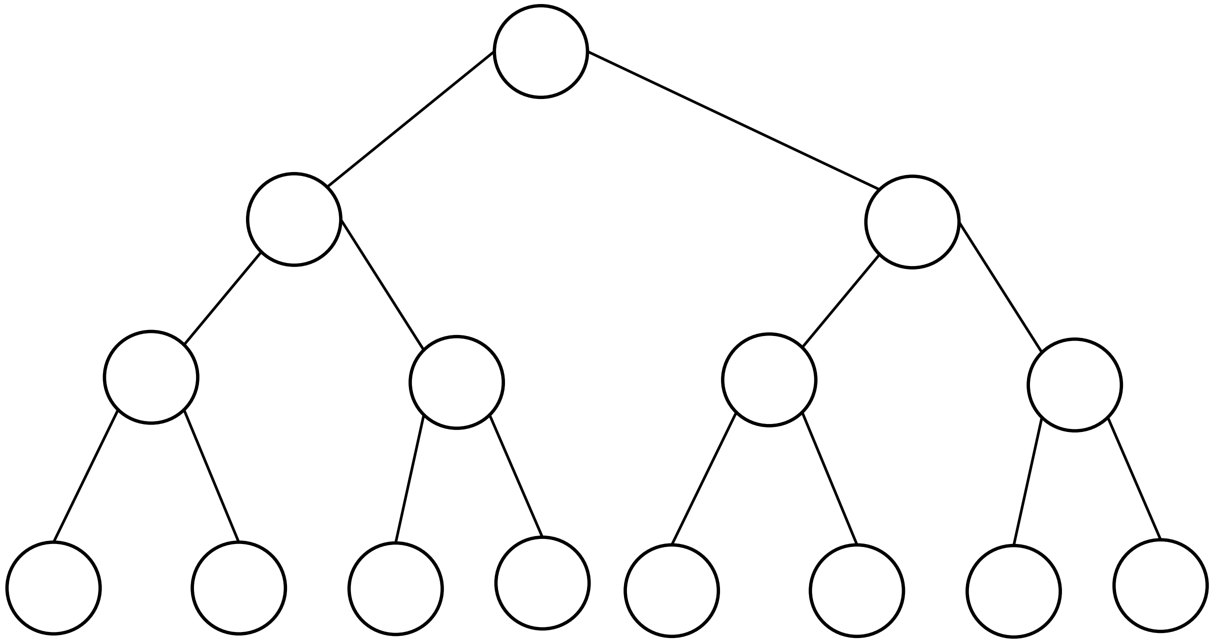
$$W(n) = \underbrace{2W(\frac{n}{2})}_{\text{左右子树递归构建}} + \underbrace{2\log n}_{\text{根节点的堆修复}}$$

- 根据 Master 定理：
  - a=b=2, E=1, $\log n = O(n^{1-\epsilon})$
  - $W(n) = O(n)$

堆构建—复杂度分析

- 堆构建

Cost=$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} O(h) = O(n)$



C=log n fix; h=log n; # = 1

C=2 fix; h=2; # = n/8

C=1 fix; h=1; # = n/4

C=0 fix; h=0; # = n/2

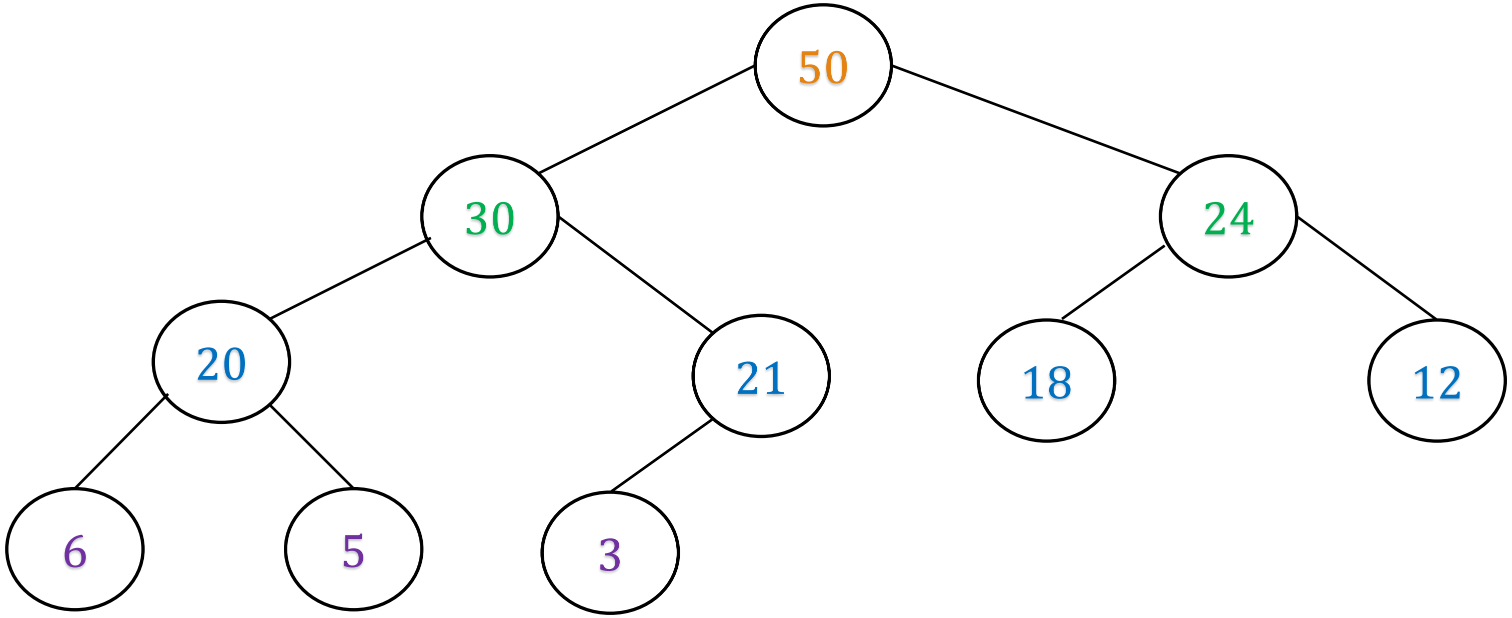
堆—思考？

- 堆中第k大的元素？
  - 1^{st}? 2^{nd} ? 3^{rd} ?
  - k^{th} ? 代价？
  - k << n，代价要求是 k 的函数？

- 证明一个有n个节点的堆，所有节点的高度之和最多为 n-1.

堆的实现

- 对于一个大小为 n 的堆，需要一个大小为 n 的数组，核心是确定父子节点的下标之间的运算关系



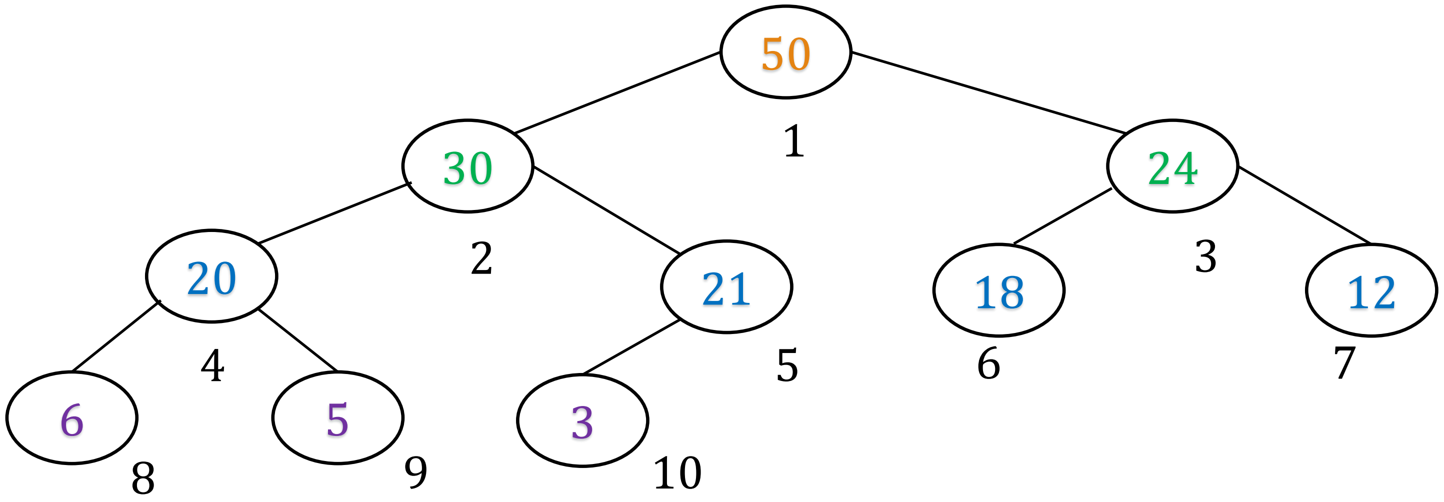
50	30	24	20	21	18	12	6	5	3

堆的实现

- PARENT(i)=$\left\lfloor \frac{i}{2} \right\rfloor$
- LEFT(i) = 2i
- RIGHT(i)=2i+1

$$i + (2^{j-1} - k) + 2(k-1) + 1 = 2i$$

i 是第 j 层的第 k 个元素



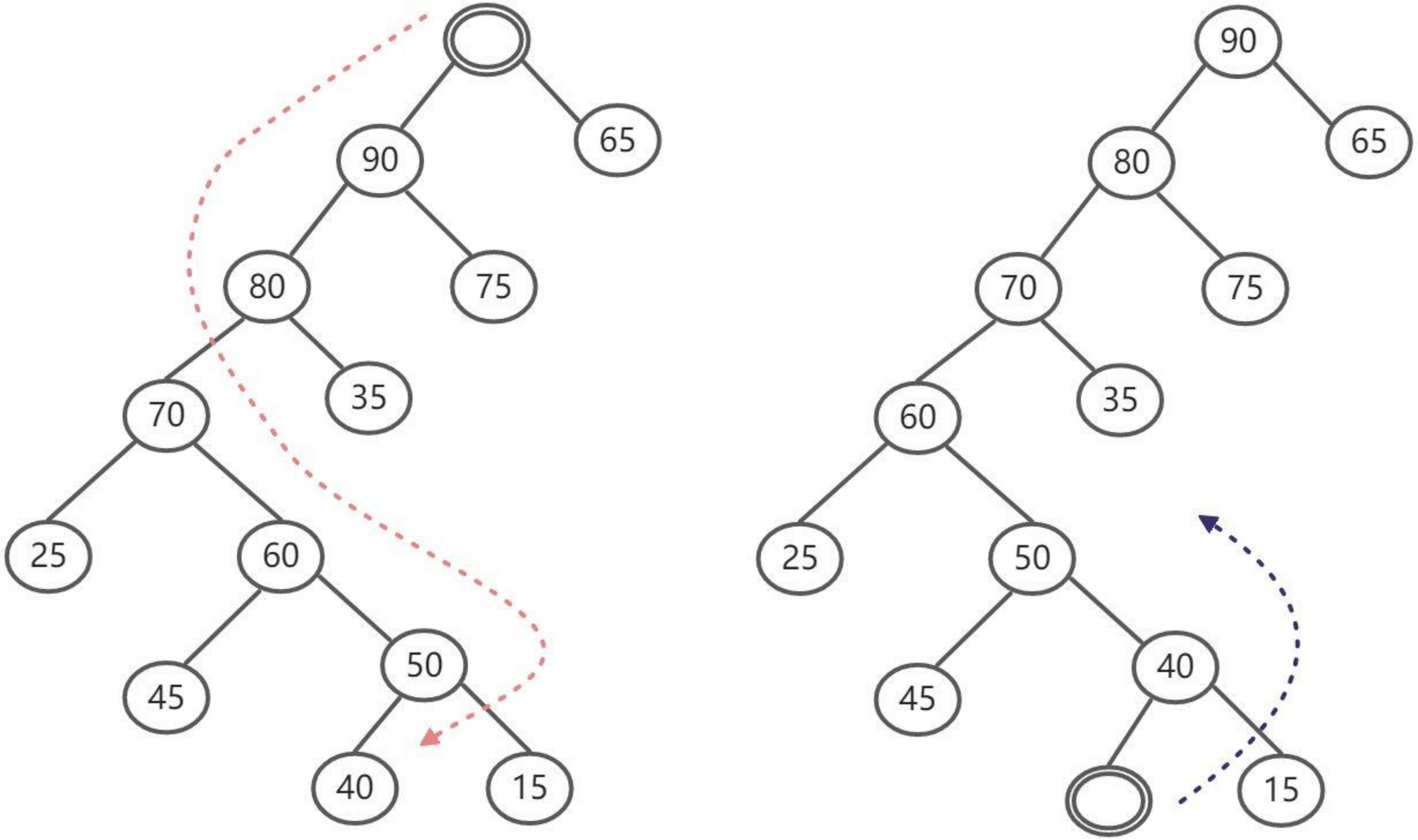
50	30	24	20	21	18	12	6	5	3
1	2	3	4	5	6	7	8	9	10

堆排序—最坏情况分析

- $W(n) = W_{构建}(n) + \sum_{k=1}^{n-1} W_{修复}(k)$
- $W_{构建}(n) \in \Theta(n), W_{修复}(k) < 2\log k$

- $W(n) \leq 2n\log n + \Theta(n)$，即 $W(n) \in \Theta(n\log n)$

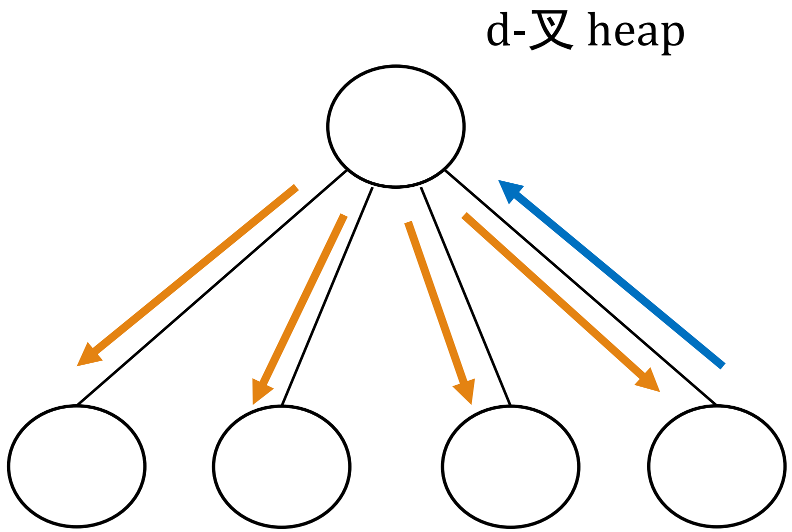
堆排序—优化



堆—泛化

- d-叉 heap
  - 结构/偏序性质

- 如何选择 d？
  - Top-down: 修复父节点
    - 代价：d 次比较
  - Bottom-up: 修复子节点
    - 代价：1



思考

- 找第 k 大的元素
  - 代价为 k 的函数 f(k)
- 找前 k 大的元素
- 合并 k 个有序列表
- 数据流中的中位数
- ……

剑指 Offer 41. 数据流中的中位数

难度 困难 190 收藏 分享 切换为英文 接收动态 反馈

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例 1：

```
输入：
["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]
[[],[1],[2],[],[3],[]]
输出：[null,null,null,1.50000,null,2.00000]
```

Lecture 3

排序下界

Outline

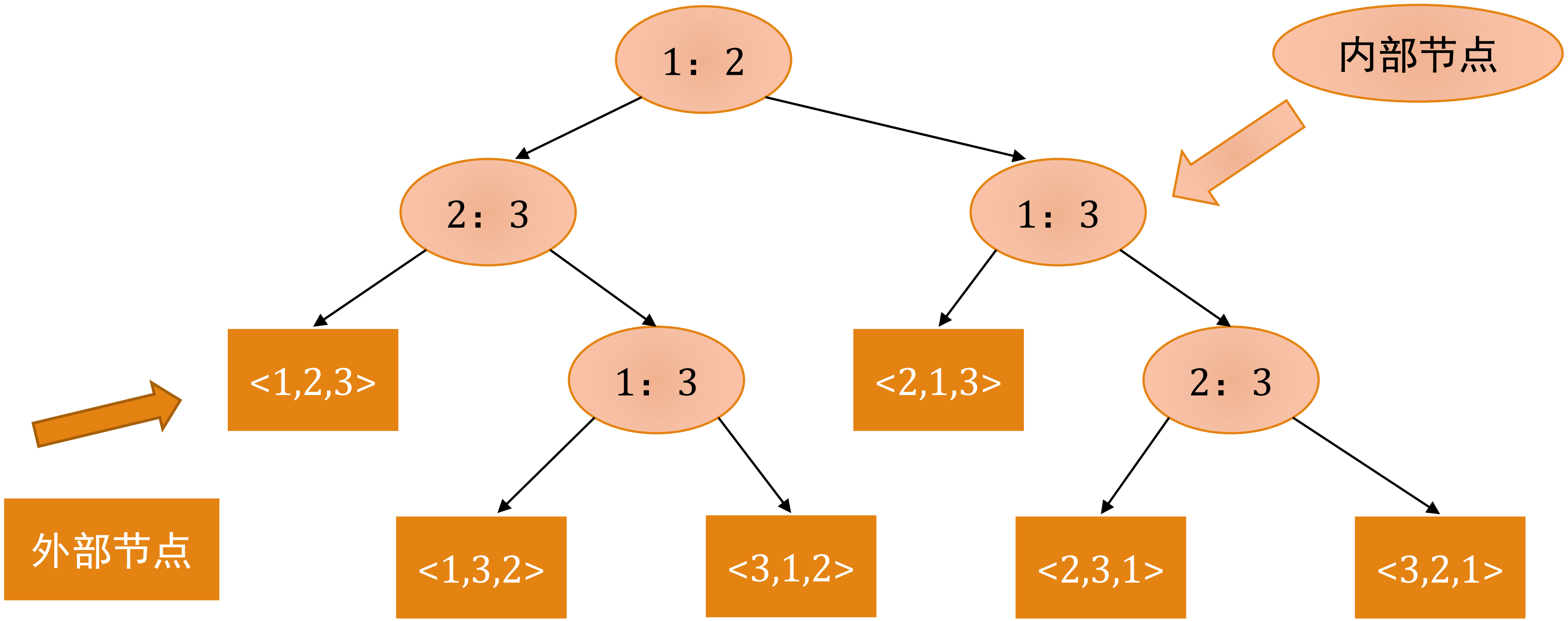
- 排序下界分析
  - 决策树
  - 最坏情况—树的高度
  - 平均情况—EPL

下界分析

- 上界：最坏情况代价
  - 对于任意可能输入，算法的代价不高于**上界**。

- 算法下界（排序为例）：
  - 对任意可能的排序算法，最坏代价不低于**下界。**

决策树的引入



三个元素比较排序的决策树

决策树分析

- 对于任意一个n元素序列，有n!不同的排列
  - 因此，决策树刚好有 n! 叶子节点。
  - 为了分析下界，我们使用刚好有 n! 叶子节点的树。

- 最坏情况的比较次数为**树的高度**

- 平均情况的比较次数为所有根到叶子节点的**路径长度的平均值**

最坏情况下界分析

- 引理：假设一棵二叉树的高度为 h，叶节点个数为 L，它们之间有如下关系：$L \le 2^h$.

- $h \ge \log L = \log n! = \Omega(n \log n)$

- 比较排序算法的最坏情况时间复杂度的下界为 $\Omega(n \log n)$

外部路径长度

- 外部路径长度：定义一颗二叉树 T 的外部路径长度 EPL(T)为根节点到所有叶子节点路径长度之和。等价地，递归定义为：
  - 只含一个节点的树，它的 EPL 为 0
  - 一棵树 T 的左右子树分别为 T_L 和 T_R , 则

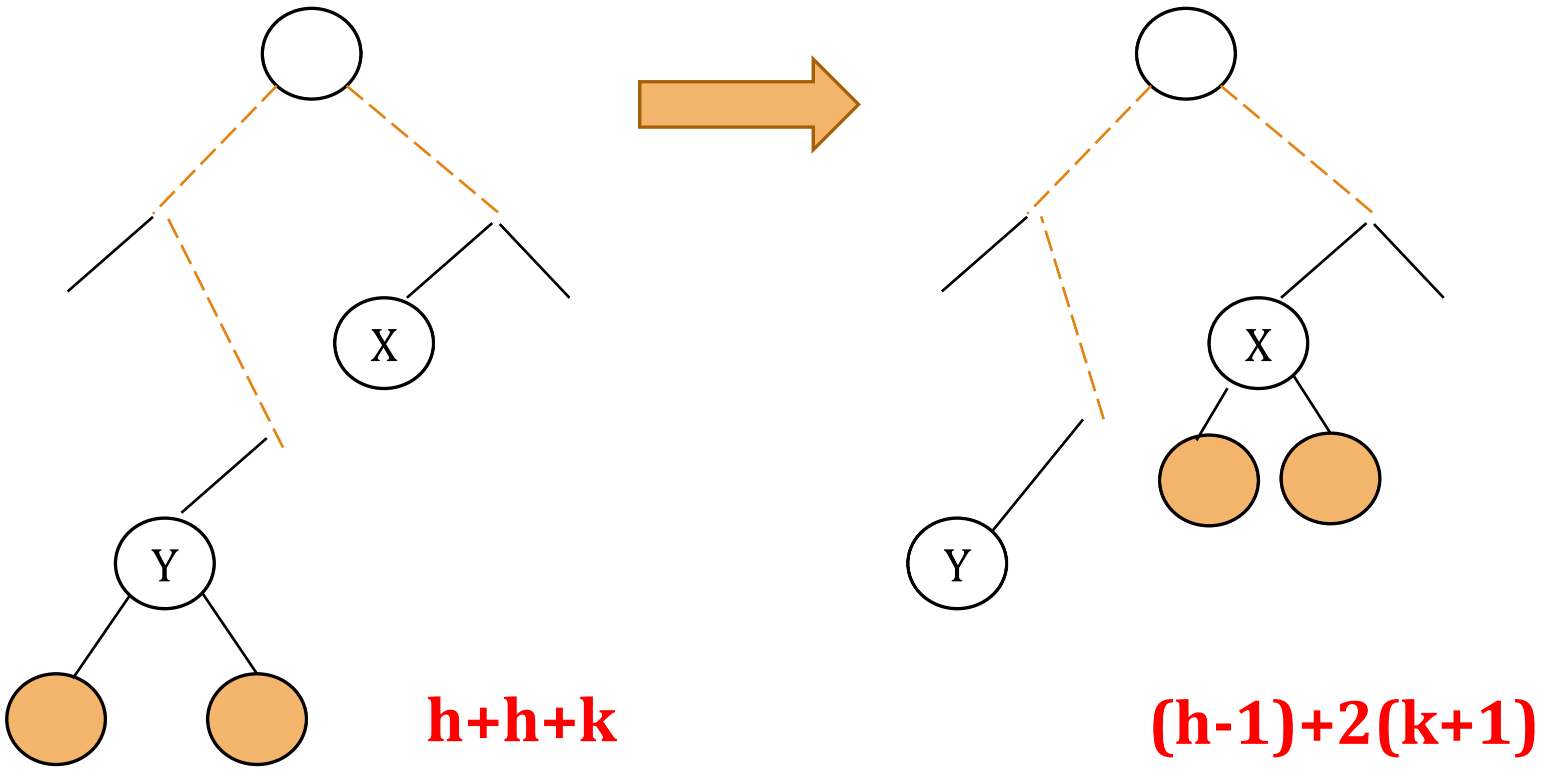
$$EPL(T) = EPL(T_L) + N_L + EPL(T_R) + N_R$$

N_L 和 N_R 分别表示左、右子树的叶节点个数。

- 平均情况复杂度：EPL/L

外部路径长度

- 引理：越平衡的 2-tree 具有越小的 EPL。



平均情况下界分析

- 对于一颗尽量平衡的 2-tree，它的叶子节点树为 L, 树高为 log L
- EPL 为 LlogL.
- A(n)=EPL/L

$$\begin{aligned} A(n) &= \frac{EPL}{L} \\ &= \Omega(\frac{L\log L}{L}) \\ &= \Omega(\log n!) \\ &= \Omega(n\log n) \end{aligned}$$

Thanks