算法设计与分析

主讲人：吴庭芳

Email: *tfwu@suda.edu.cn*

苏州大学 计算机学院







第五讲 快速排序算法

内容提要：

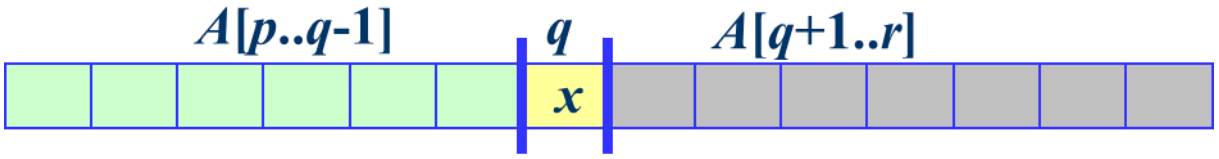
- 快速排序算法的描述
- 快速排序算法的性能
- 快速排序算法的随机化版本
- 排序算法的下界

快速排序算法的描述

- 快速排序是 C.R.A. Hoare 于 1962 年提出的一种**原址排序算法**
- 对于包含 n 个数的输入数组，快速排序算法在最坏情况下运行时间为 $\Theta(n^2)$
- 但快速排序通常是实际排序应用中最好的选择，因为其平均性能非常好：**期望运行时间为 $\Theta(n\lg n)$，且其常数因子较小**

快速排序算法的描述

- **快速排序算法基本思想：采用分治策略，将未排序数组分为两部分，然后分别递归调用自身进行排序**
  - **分解：**数组 $A[p\ldots r]$ 被划分为两个（某个可能为空）子数组 $A[p\ldots q\text{-}1]$ 和 $A[q\text{+}1..r]$，使得 $A[p\ldots q\text{-}1]$ 中每个元素都小于或等于 $A[q]$，$A[q\text{+}1..r]$ 中的每个元素大于 $A[q]$。下标 q 在这个划分过程中进行计算



  - **解决：**递归调用快速排序算法，对子数组 $A[p...q\text{-}1]$ 和 $A[q\text{+}1..r]$ 排序
  - **合并：**因为子数组都是原址排序的，不需要进行合并操作

快速排序算法的描述

- 快速排序伪代码：

```
QUICKSORT(A, p, r)
1   if   p < r
2        then   q = PARTITION(A, p, r)   //划分过程
3               QUICKSORT(A, p, q-1)    //对子数组 A[p...q-1] 递归调用
4               QUICKSORT(A, q+1, r)    //对子数组 A[q+1...r] 递归调用
```

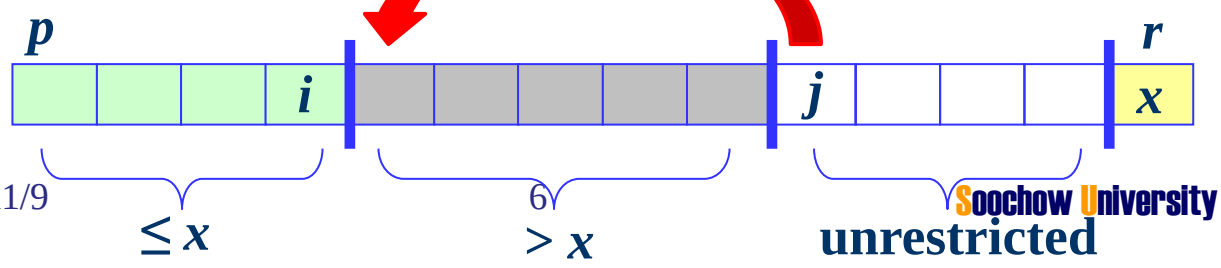
- 数组划分过程 PARTITION 是 QUICKSORT 算法的关键

快速排序算法的描述

- 数组划分过程 PARTITION：

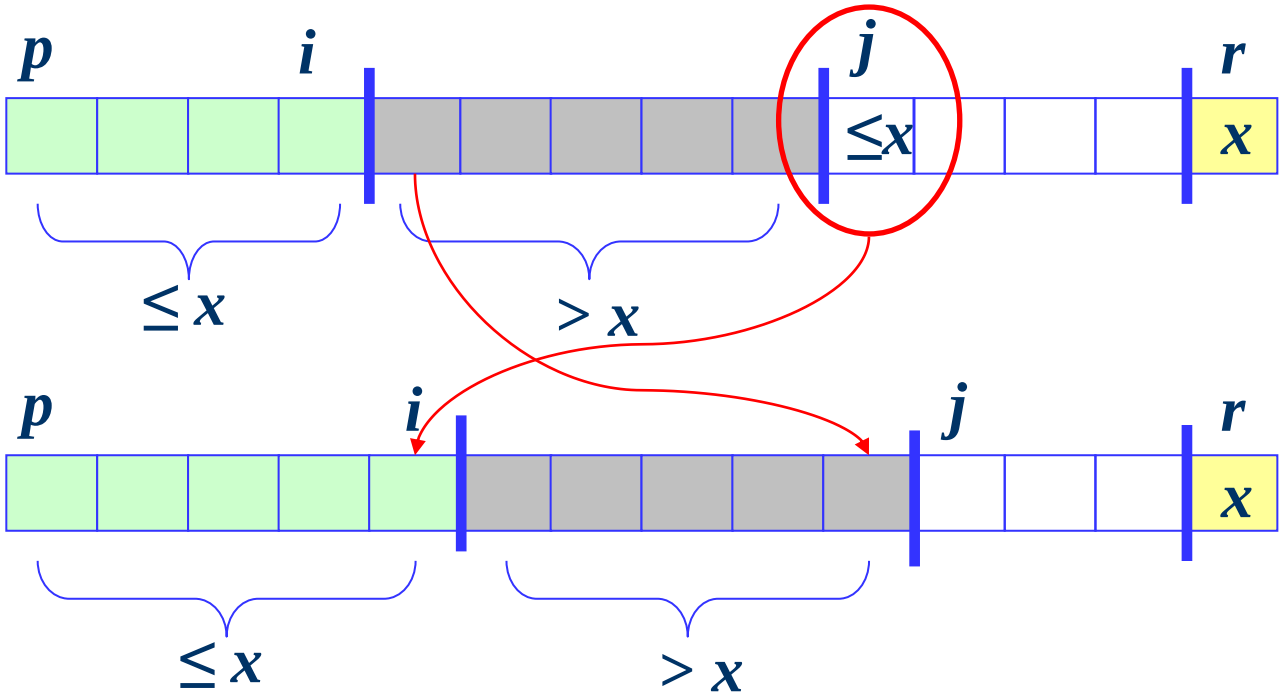
```
PARTITION(A, p, r)
1  x = A[r]                          // x 为主元，围绕 x 划分子数组
2  i = p - 1                         // i 下标左边的数据表示比主元 x 小的元素
3  for j = p to r – 1                // j 用来寻找比主元 x 小的元素
4      if A[j] ≤ x
5          i = i + 1                 // i 往后移动一位，给要换过来的 A[j] 空出个位置
6          exchange A[i] with A[j]   // 把A[j]换到新位置，原位置比 x大的元素换到 j 位置
7  exchange A[i + 1] with A[r]       // 将 x 与大于 x 部分的第一个元素进行交换
8  return i + 1                      // 返回主元 x 的新下标
```

- PARTITION 在子数组 $A[p…r]$ 上的执行时间为 $\Theta(n)$，$n=r-p$



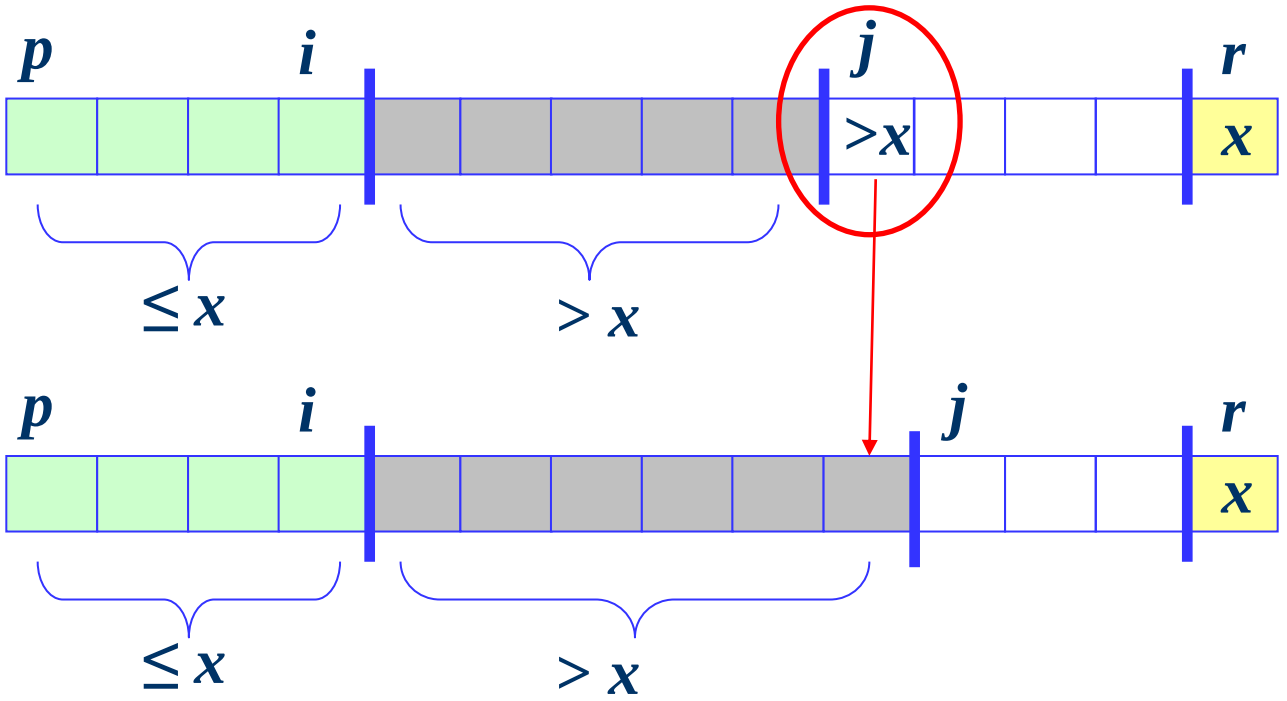
快速排序算法的描述

- i 和 j 如何改变：



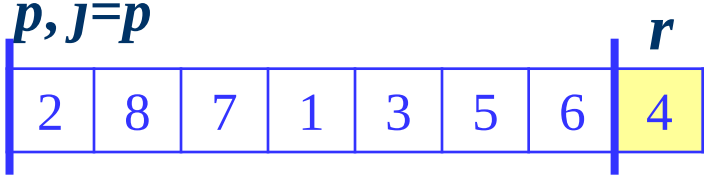
快速排序算法的描述

- i 和 j 如何改变：

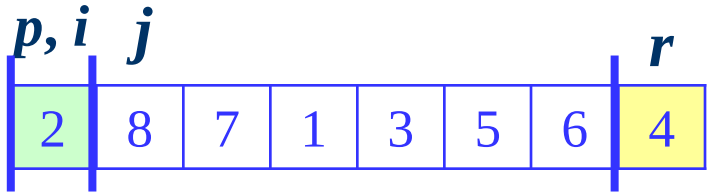


快速排序算法的描述

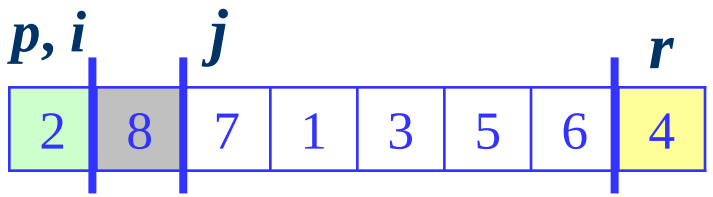
(a) $i=p-1$ $p, j=p$



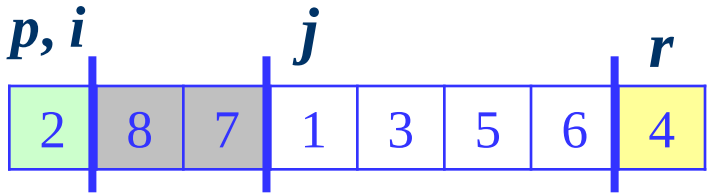
(b) p, i j



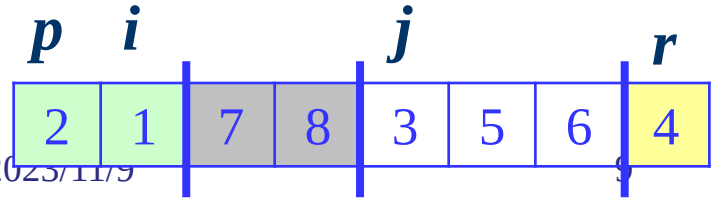
(c) p, i j



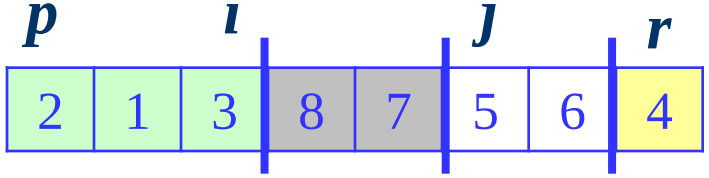
(d) p, i j



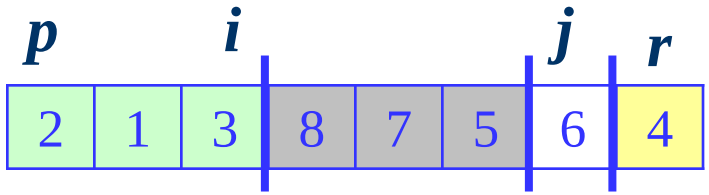
(e) p i j



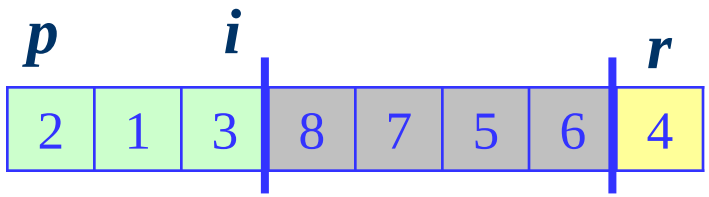
(f) p i j



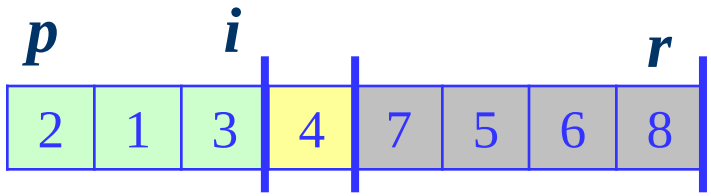
(g) p i j



(h) p i



(i) p i









第五讲 快速排序算法

内容提要：

- 快速排序算法的描述
- 快速排序算法的性能
- 快速排序算法的随机化版本
- 排序算法的下界

快速排序算法的性能

- 快速排序算法最坏情况下时间分析
  - 当划分产生得到的两个子问题分别包含 n-1 个元素和 0 个元素时，比如数组已被排序好，此时快速排序算法的最坏情况发生
  - 假设算法的每一次递归调用都出现这种极度不平衡的划分，快速排序算法的递归式可以表示为：

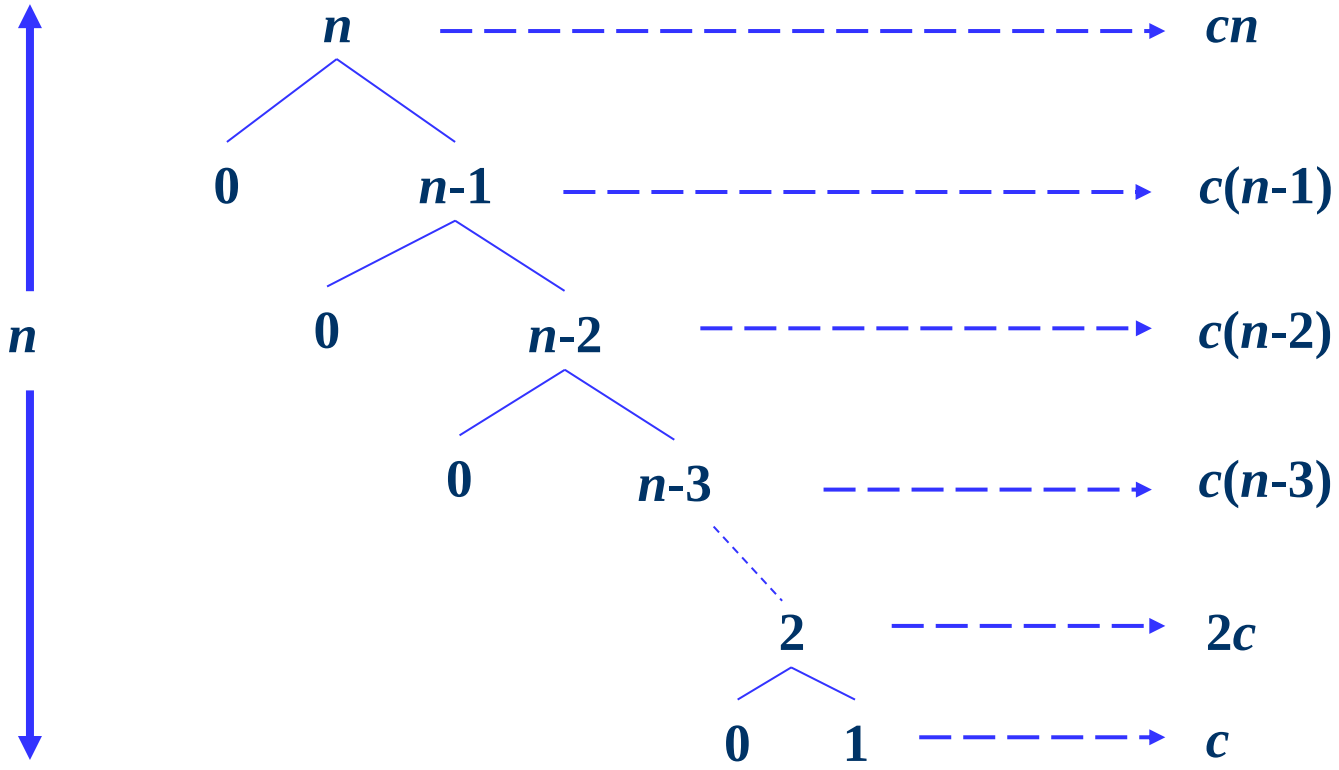
$$\mathrm{T}(n) = \mathrm{T}(n-1) + \mathrm{T}(0) + \Theta(n) = \mathrm{T}(n-1) + \Theta(n)$$

    利用代入法求得解为：$\mathrm{T}(n) = \Theta(n^2)$

  - 因此，如果算法的每一层递归上，划分都是最大程度的不平衡，那么算法的时间复杂度就为：$\Theta(n^2)$

快速排序算法的性能

$$T(n) = T(n-1) + \Theta(n)$$



快速排序算法的最坏情况分析

- 快速排序算法最坏情况时间的理论分析
  - 前面从直觉上可以判断出最坏情况发生在每次划分过程中产生的两个子数组分别包含 n-1 个元素和 0 个元素的时候
  - 下面将从理论证明这种划分过程产生的两个子数组分别包含 n-1 个元素和 0 个元素的情况就是最坏情况

快速排序算法的最坏情况分析

- 设 T(n) 是快速排序算法作用于规模为 n 的输入上的最坏情况下的运行时间：

$$\mathrm{T}(n) = \max_{0 \leqslant q \leqslant n-1} (\mathrm{T}(q) + \mathrm{T}(n-q-1)) + \Theta(n)$$

因为 PARTITION 过程生成的两个子问题的规模之和总为 n-1，所以参数 q 的变化范围是 0 到 n-1

- 猜测快速排序的最坏情况运行时间是：$\mathrm{T}(n)=\Theta(n^2)$

快速排序算法的最坏情况分析

$$\mathrm{T}(n)=\max_{0\leqslant q\leqslant n-1}(\mathrm{T}(q)+\mathrm{T}(n-q-1))+\Theta(n)$$

- 首先证明 $\mathrm{T}(n)\leqslant cn^2$，其中 c 为常数，将归纳带入上式，得到：

$$\begin{aligned}\mathrm{T}(n)&\leqslant\max_{0\leqslant q\leqslant n-1}(cq^2+c(n-q-1)^2)+\Theta(n)\\&\leqslant c\cdot\max_{0\leqslant q\leqslant n-1}(q^2+(n-q-1)^2)+\Theta(n)\\&\leqslant c(n-1)^2+\Theta(n)\\&\leqslant cn^2-c(2n-1)+\Theta(n)\\&\leqslant cn^2\end{aligned}$$

- 因为在 [0, n-1] 上 $q^2+(n-q-1)^2$ 关于 q 递减，所以当 q=0 时 $q^2+(n-q-1)^2$ 有最大值 $(n-1)^2$
- 挑选足够大的 c，使得 $c(2n-1)$ 项能够显著大于 $\Theta(n)$ 项，有 $\mathrm{T}(n)=\mathrm{O}(n^2)$

快速排序算法的性能

- 快速排序算法最佳情况的时间分析
  - 在可能的**最平衡的划分**中，PARTITION 过程得到的**两个子问题的规模都不大于** $n/2$，其中一个子问题的规模为 $\lfloor n/2 \rfloor$，另一个子问题的规模为 $\lfloor n/2 \rfloor - 1$
  - 在这种情况下，通过忽略向下取整和向上取整以及减 1 的操作，快速排序算法运行时间的递归表达式为：

$$T(n) = 2T(n/2) + \Theta(n)$$

根据主方法的情况 2，递归式的解为：

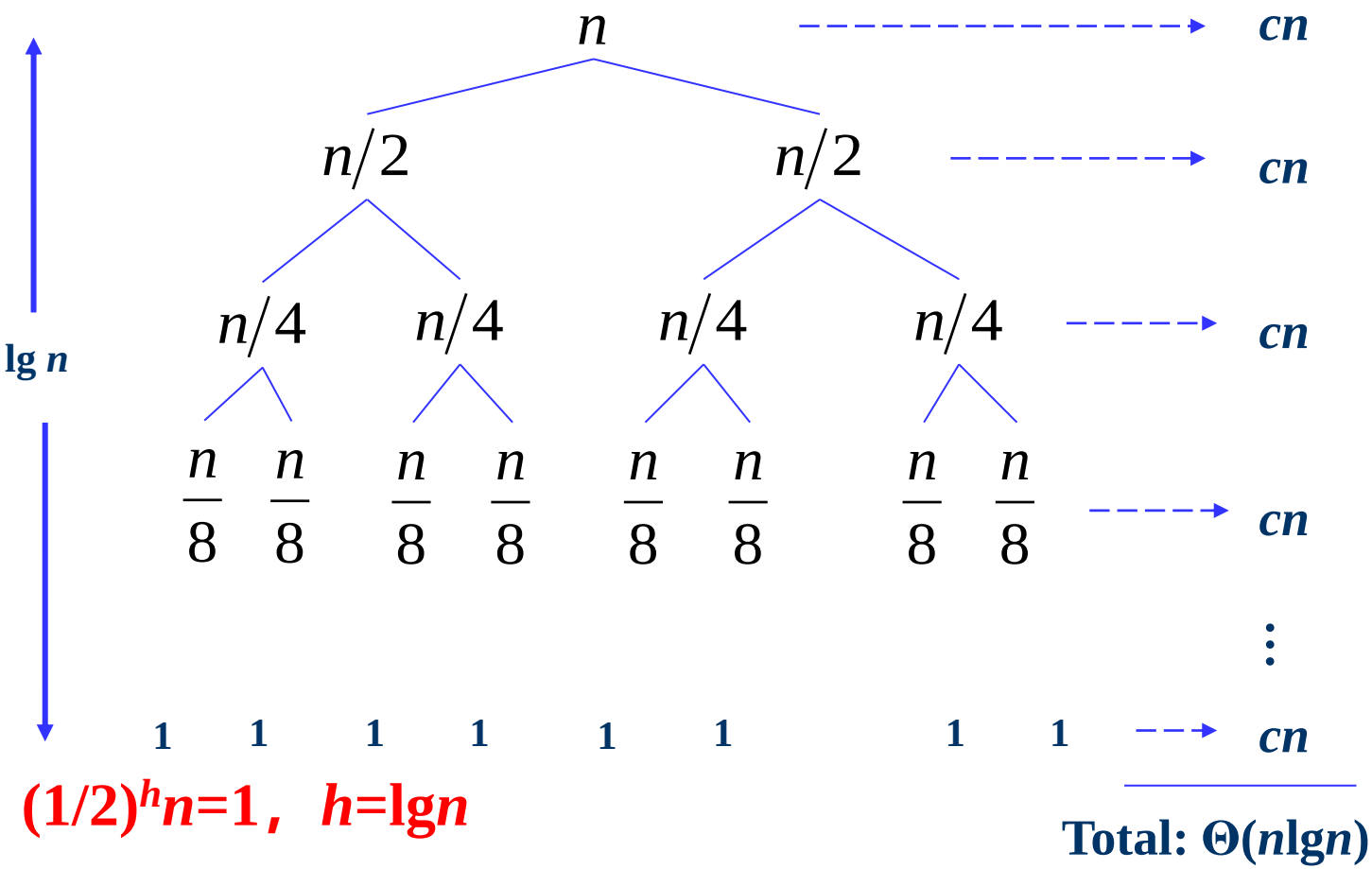
$$T(n) = \Theta(n \lg n)$$

快速排序算法的性能









快速排序算法的性能

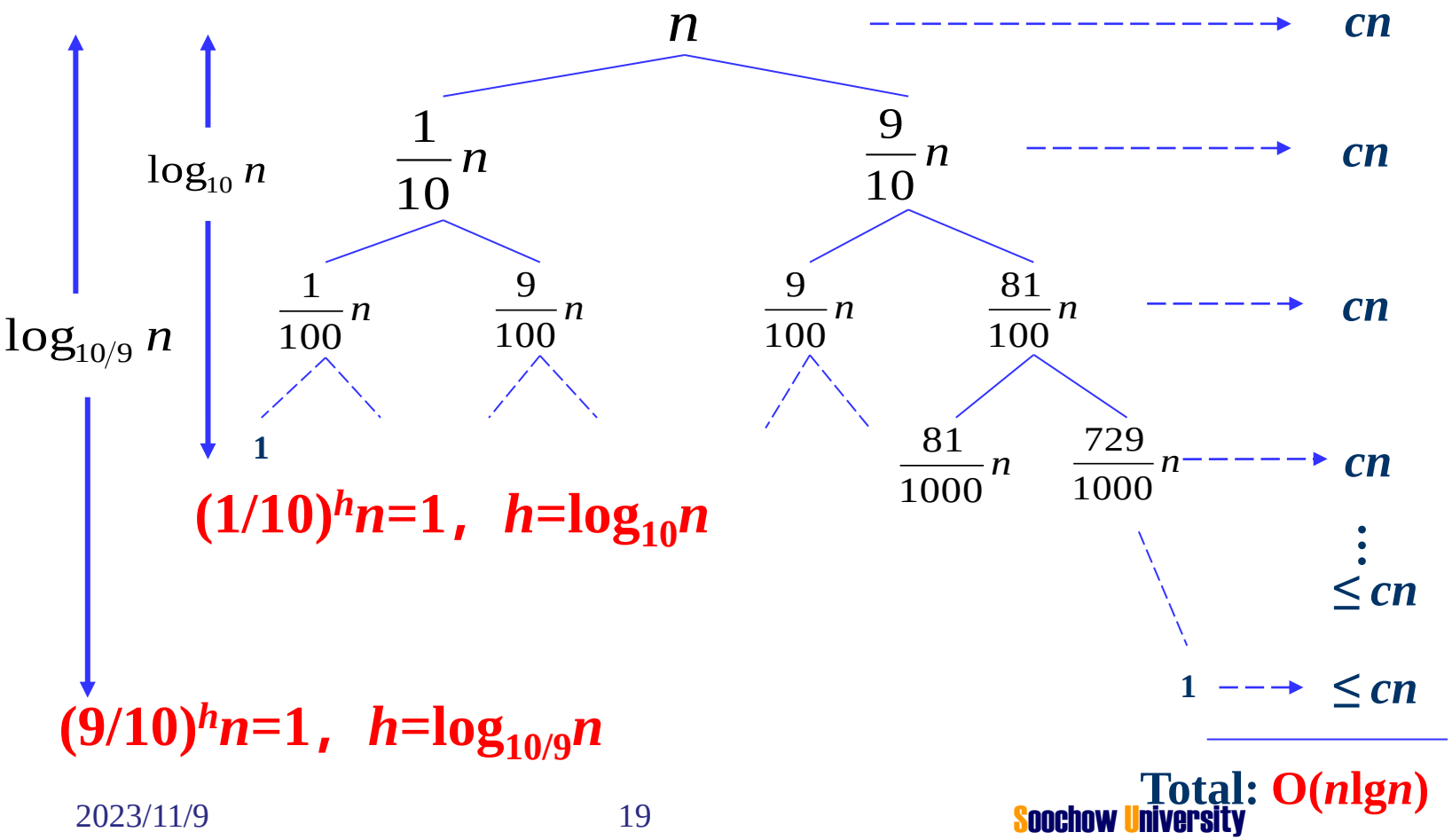
- 平衡的划分
  - 如果以**常数比例进行划分**，即使**该比例很不平衡**（如9:1），快速排序算法运行时间的递归表达式为：

$$\mathrm{T}(n) = \mathrm{T}(n/10) + \mathrm{T}(9n/10) + \Theta(n)$$

快速排序算法的运行时间仍然为 $\mathrm{T}(n)=\mathrm{O}(n\lg n)$

快速排序算法的性能

- **求解递归式：** $\mathrm{T}(n) = \mathrm{T}(n/10) + \mathrm{T}(9n/10) + \Theta(n)$



快速排序算法的性能

- 平衡的划分
  - 事实上，即使是 99:1 的划分，快速排序的运行时间代价仍然为 $O(n\lg n)$
  - 任何一种按常数比例进行划分**所产生的递归树的深度都为 $\Theta(\lg n)$**，其中每一层的代价为 $O(n)$，因而不管常数比例是什么，总的运行时间都为 $O(n\lg n)$，只是其中隐含的常数因子不同而已

快速排序算法的性能

- **概率分析：在问题分析中应用概率的理念**
  - 概率分析用来分析一个算法的运行时间，也可用于其他量的分析，例如利用概率分析技术来分析雇用费用
  - 概率分析的本质：需要使用或假定关于输入的分布
- **概率分析：首先使用关于输入分布的知识或者对其做的假设，然后分析算法，计算出一个平均情况下的运行时间。实际上，对所有可能的输入产生的运行时间取平均**
- **当报告此种类型的运行时间时，称其为平均情况运行时间**

快速排序算法的性能

随机算法

- 为了利用概率分析技术，就需要了解关于输入分布的一些信息。但在许多情况下，我们对输入分布了解很少。而且即使知道输入分布的某些信息，也无法从计算上对该分布知识建立模型
- **那么如何让输入变得可控?**

  通过对算法中的某部分的行为进行随机化，从而为输入强加一种分布，则可利用概率和随机性作为算法设计与分析的工具进行处理

快速排序算法的性能

- **随机算法：如果一个算法的行为不仅由输入决定，而且也由一个随机数生成器产生的数值决定，则称这个算法是随机的**
- **当报告此种类型的运行时间时，称其为期望运行时间**
- **随机数生成器 RANDOM：**
  - 调用 RANDOM(a, b) 将返回一个介于 a 和 b 之间（包含 a 和 b）的整数，并且每个整数以等概率出现

例： RANDOM(0, 1)：返回 0 和 1，每个出现的概率都为 1/2

RANDOM(3, 7)：返回 3，4，5，6，7，每个出现的概率为 1/5

  - 每次 RANDOM 返回的整数**都独立于**前面调用的返回值
  - 大多数编程环境中的 RANDOM 实际上由一个确定的算法模拟产生的（伪随机产生器），其结果表面上看上去像是随机数

快速排序算法的性能

一般而言：

- 当概率分布是在算法的输入上时，我们讨论算法的“平均情况运行时间”
- 当算法本身做出随机选择时，我们讨论算法的“期望运行时间”

快速排序算法的性能

- **快速排序算法平均情况的直觉分析**
  - ✓ 为了对快速排序的各种随机情况有一个清楚的认识，需要对遇到各种输入的出现频率作出假设，这里假设输入数据的所有排列都是等概率的（即均匀随机排列）
  - ✓ 当我们对一个随机的输入数组运行快速排序算法时，要想在每一层上都有同样的划分是不太可能的。我们预期某些划分比较平衡，而另一些则会很不平衡
  - ✓ 事实上，可以证明 PARTITION 过程所产生的划分中 80% 以上都比 9:1 更平衡，而另 20% 的划分比 9:1 更不平衡

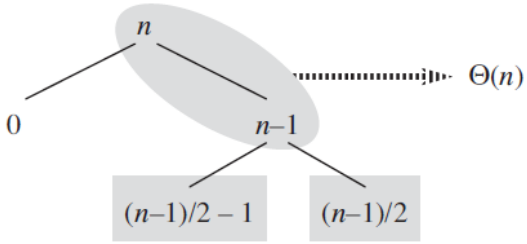
快速排序算法的性能

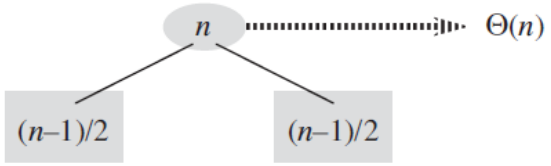
- **快速排序算法平均情况的直觉分析**
  - ✓ 在平均情况下，PARTITION 过程所产生的划分同时混合有“好”和“差”的划分
  - ✓ 与 PARTITION 平均情况执行过程所对应的递归树中，好和差的划分随机分布在树的各层上的
  - ✓ 为与我们的直觉相一致，假设好和差的划分交替出现在树的各层上，并且好的划分是最好的划分，而差的划分是最坏的划分

快速排序算法的性能

快速排序算法平均情况的直觉分析

- 左图（对应于平均情况下的划分）表示递归树的连续两层上的划分情况：在根节点处，划分的代价为 n，划分出来的两个子数组大小为 0 和 n-1，即最坏情况
- 在下一层上，大小为 n-1 的子数组按最好情况划分为大小分别为 (n-1)/2-1 和 (n-1)/2 的两个子数组





- 在一个差的划分后面接着一个好的划分，产生三个子数组，大小分别是 0、(n-1)/2-1 和 (n-1)/2，划分代价为 $\Theta(n)+\Theta(n-1)=\Theta(n)$
- 在右图（对于最佳情况下的划分），一层划分就产生出大小为 (n-1)/2 的两个子数组，划分代价为 $\Theta(n)$，这种划分是平衡的

快速排序算法的性能

- **快速排序算法平均情况的直觉分析**
  - ✓ 从直觉上看，差划分的代价 $\Theta(n\text{-}1)$ 可以被吸收到好的划分的代价 $\Theta(n)$ 中去，而结果也是一个好的划分
  - ✓ 因此，当好、差的划分交替出现时（对应于平均情况下的划分），快速排序的时间复杂度与全是好的划分时一样（对应于最佳情况下的划分），仍然是 $O(n\lg n)$，区别是 O 符号中隐藏的常数因子要略大一些







第五讲 快速排序算法

内容提要：

- 快速排序算法的描述
- 快速排序算法的性能
- 快速排序算法的随机化版本
- 排序算法的下界

快速排序算法的随机化版本

- 前面讨论快速排序的平均情况性能的分析中，我们的前提假设是：**输入数据的所有排列都是等概率的**。但在实际应用中，这个假设不会总是成立
- 解决方法：利用随机化策略，能够克服分布的等可能性假设所带来的问题
- 随机算法不是假设输入符合某个分布，而是通过对输入**设定一个分布**

快速排序算法的随机化版本

- **随机算法的性质：**
  - ✓ 随机算法中，由于首先需要对输入进行重排列，因此，对于任何一个给定的输入，都无法说出具体的输出，因为在每次运行随机算法时输出都不相同
  - ✓ 因此，对于同一个输入，每次运行随机算法时，每次输出的值（得到的代价）可能不一样，依赖于随机选择
  - ✓ 对于随机算法，没有特别的输入会引起它的最坏情况行为，因为随机化处理使得输入次序不再相关。只有随机数生成器产生一个”不走运“的排列时，随机算法才会运行得很差

快速排序算法的随机化版本

- **快速排序的随机化版本有一个和其他随机化算法一样的有趣性质：没有一个特别的输入会导致最坏情况出现**
- **随机化快速排序算法的最坏情况出现是由随机数产生器决定的**
  - 即使有意给出一个坏的输入也没用，因为随机化排列会使得输入数据次序对算法不产生影响
  - 只有在随机数产生器给出一个很不巧的排列时，随机算法的最坏情况才会出现
  - 事实上可以证明几乎所有的排列都可使快速排序接近平均情况，只有非常少的几个排列才会导致算法的最坏情况出现

快速排序算法的随机化版本

- 根据前面对快速排序算法的平均情况的直觉分析，一组好坏相杂的划分仍能产生很好的运行时间，因此我们可以认为快速排序算法的随机化版本也能具有较好的计算性能
- 可以向算法中加入随机性，以便对于所有的输入，算法均能获得较好的期望性能
- 采用**随机采样 (random sampling)** 的随机化技术
  - 做法：从子数组 $A[p\ldots r]$ 中随机选择一个元素作为主元，即将 $A[r]$ 与从 $A[p\ldots r]$ 中随机选出的一个元素交换，保证主元元素 $x = A[r]$ 等概率地从 $r\text{-}p+1$ 个元素中选取的，从而达到对输入数组的划分比较平衡

快速排序算法的随机化版本

- 采用随机采样 (random sampling)

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with 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      q = RANDOMIZED-PARTITION(A, p, r )
3      RANDOMIZED-QUICKSORT(A, p, q-1)
4      RANDOMIZED-QUICKSORT(A, q+1, r)
```

快速排序算法的期望运行时间







- 引入指示器随机变量的目的：为了建立**概率** (probabilities) 和**期望** (expectations) 之间的联系，用于实现概率与期望之间的转换
- 指示器随机变量的定义：给定一个样本空间 S (sample space) 和其中一个事件 A (event)，那么事件 A 对应的**指示器随机变量 $\mathrm{I}\{A\}$** 定义为：

$$\mathrm{I}\{A\} = \begin{cases} 1, & \text{如果} A \text{发生} \\ 0, & \text{如果} A \text{不发生} \end{cases}$$

快速排序算法的期望运行时间







口 例：抛掷一枚硬币，求**正面朝上**的期望次数

解：首先，样本空间 $S=\{H, T\}$，其中 $\Pr(H)=\Pr(T)=1/2$

接下来定义一个指示器随机变量 $X_H=\mathrm{I}\{H\}$，对应于硬币正面朝上的事件 H：

$$X_H=\mathrm{I}\{H\}=\begin{cases}1, & \text{如果}H\text{发生}\\ 0, & \text{如果}T\text{发生}\end{cases}$$

快速排序算法的期望运行时间







口 例：抛掷一枚硬币，求**正面朝上**的期望次数。

解：则一次抛掷硬币正面朝上的期望次数，即指示器随机变量 $\mathrm{I}\{H\}$ 的期望值：

$$\begin{aligned}\mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2\end{aligned}$$

注：一个事件对应的指示器随机变量的期望值等于该事件发生的概率

快速排序算法的期望运行时间







- **引理 1. 给定一个样本空间 S 和 S 中的一个事件 A，设事件 A 对应的指示器随机变量为 $X_A=\mathrm{I}\{A\}$，那么 $\mathrm{E}[X_A]=\Pr\{A\}$**

证明：由指示器随机变量的定义以及期望值的定义，有：

$$\begin{aligned}\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1\cdot\Pr\{A\}+0\cdot\Pr\left\{\overline{A}\right\} \\ &= \Pr\{A\}\end{aligned}$$

其中，$\overline{A}$ 表示 S-A，即 A 的补

快速排序算法的期望运行时间

- **RANDOMIZED-QUICKSORT 快速排序的期望运行时间分析**
  - QUICKSORT 和 RANDOMIZED-QUICKSORT 除了选择主元元素有差异外，其他方面完全相同
  - 因此，可以讨论 QUICKSORT 和 PARTITION 的基础上分析 RANDOMIZED-QUICKSORT
  - RANDOMIZED-QUICKSORT 随机地从子数组中选择元素作为主元

快速排序算法的期望运行时间

- **RANDOMIZED-QUICKSORT 快速排序的期望运行时间分析**
  - QUICKSORT 的运行时间是由花在 PARTITION 过程上的时间所决定的
  - 每当 PARTITION 过程被调用时，需要选择一个主元元素，而且该元素不会被包含在后续对 QUICKSORT 和 PARTITION 的递归调用中，于是最多对 PARTITION 调用 n 次
  - 调用一次 PARTITION 的时间为 O(1)+一段循环时间，这段时间与 3~6 行 **for** 循环的迭代次数成正比，**for** 循环每一轮迭代都要在第 4 行中进行一次比较：比较主元元素与数组 A 中的另一个元素
  - 只需要知道第 4 行被执行的总次数，就可以知道 QUICKSORT 中 **for** 循环所花时间的界

快速排序算法的期望运行时间

- 数组划分过程 PARTITION：

```
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)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for  j = p to r – 1
4       if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

快速排序算法的期望运行时间

- **引理 7.1：假设所有元素都不相同，当在一个包含 n 个元素的数组上运行 QUICKSORT 时，假设在 PARTITION 的第 4 行中所执行的总的比较次数为 X，那么 QUICKSORT 的运行时间为 $O(n+X)$**
  - **证明：**根据上面的讨论，对 PARTITION 过程的调用共有 n 次，每一次调用 PARTITION 的时间为 O(1) 和执行若干次 for 循环。在每一次 for 循环中，都需要执行第 4 行
  - **关键点：**目标是计算出 X，即所有对 PARTITON 过程的调用中，所执行的总的比较次数。因此，必须了解算法在什么时候对数组中的两个元素进行比较，什么时候不进行比较

快速排序算法的期望运行时间

- **定义指示器随机变量：** $X_{ij} = \mathrm{I}\{z_i 与 z_j 进行比较\}$
  - 将数组 A 的元素重新命名为 z_1，z_2，…，z_n，其中 z_i 是数组 A 中第 i 小的元素
  - 定义 $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ 为 z_i 与 z_j 之间的元素集合（包含 i 和 j）
  - 首先，每一对元素至多比较一次：因为每个元素只与主元元素进行比较，并且在某一次 PARTITION 过程调用结束之后，该次调用中所用到的主元元素就再也不会与其他元素进行比较了
  - 因此，算法的总比较次数：$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

快速排序算法的期望运行时间

- **对上式两边取期望，有：**

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{与} z_j \text{进行比较}\}$$

- **考虑两个元素 z_i 与 z_j 何时不会进行比较?**

范例: 令 A = {1, 2, 3, …, 10}， 假设选择的第一个主元是 7，A 被划分为两个子集合 A_1={1, 2, …, 6}，A_2={8, 9, 10}。 在这一过程中，主元 7 与所有这些元素进行比较。但第一个集合 A_1 中任何一个元素不会与第二个集合 A_2 中的任何元素进行比较

快速排序算法的期望运行时间

- 假设每个元素是互异的，因此，一旦一个满足 $z_i < x < z_j$ 的主元 x 被选择后，那么 z_i 与 z_j 就再也不可能进行比较了
- 如果 z_i 在 $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ 中的所有其他元素之前被选为主元，那么 z_i 将与 Z_{ij} 中除了它自身之外的所有元素进行比较
- 类似地，如果 z_j 在 Z_{ij} 中的所有其他元素之前被选为主元，那么 z_j 将与 Z_{ij} 中除了它自身之外的所有元素进行比较
- 因此，两个元素 z_i 与 z_j 进行比较，当且仅当 Z_{ij} 中将被选为主元的第一个元素是 z_i 或者 z_j

快速排序算法的期望运行时间

- 假设 RANDOMIZED-PARTITION **随机且独立地选择主元**
- 在 Z_{ij} 中的某一元素被选为主元之前，整个集合 Z_{ij} 的元素都属于某一划分的同一分区。因此，Z_{ij} 中的任何元素都会被等概率地首先选为主元。集合 Z_{ij} 中有 j-i+1 个元素，并且主元的选择是随机且独立的，所以任何元素被首先选为主元的概率是 1/j-i+1

$$\begin{aligned}\Pr\{z_i\text{与}z_j\text{进行比较}\} &= \Pr\{z_i\text{或}z_j\text{是集合}Z_{ij}\text{中选出的第一个主元}\}\\ &= \Pr\{z_i\text{是集合}Z_{ij}\text{中选出的第一个主元}\}\\ &\quad + \Pr\{z_j\text{是集合}Z_{ij}\text{中选出的第一个主元}\}\\ &= \frac{1}{j-i+1}+\frac{1}{j-i+1}=\frac{2}{j-i+1}\end{aligned}$$

快速排序算法的期望运行时间

- 根据上述分析，E[X] 有：

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} \mathrm{O}(\lg n) = \mathrm{O}(n \lg n)
\end{aligned}$$

- 综合起来，使用 RANDOMIZED-PARTITION，在输入元素互异情况下，快速排序算法的期望运行时间为 O($n$lg$n$)







第五讲 快速排序算法

内容提要：

- 快速排序算法的描述
- 快速排序算法的性能
- 快速排序算法的随机化版本
- 排序算法的下界

排序算法时间的下界

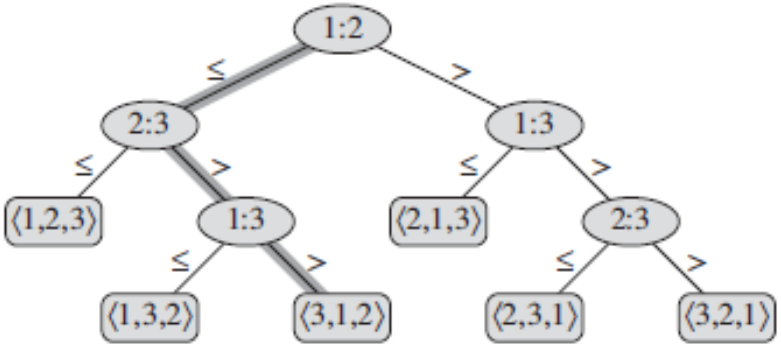






- 基于**比较运算**的排序算法的时间下界
  - 以比较为基础的排序：只使用比较运算来决定元素之间的大小关系并调整其位置，不做改变元素值大小等其它操作的排序算法
  - 冒泡排序，插入排序，归并排序，快速排序都是比较排序算法

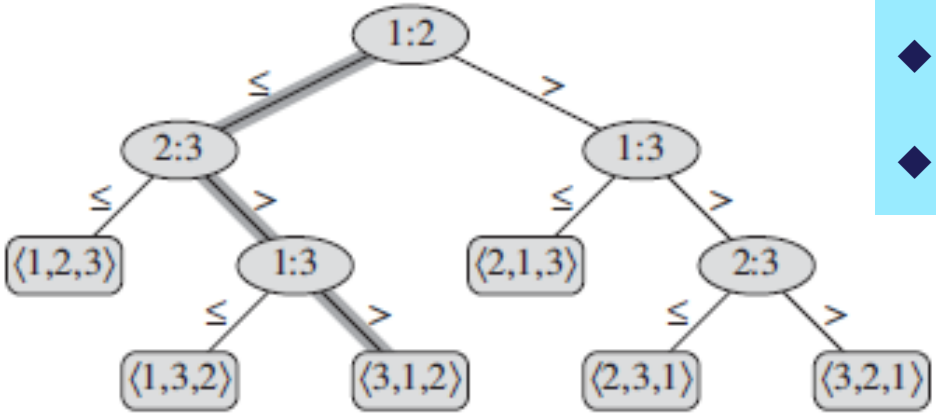
排序算法时间的下界



- 若 $a_i \le a_j$，进入下一级的左分支
- 若 $a_i > a_j$，进入下一级的右分支

- **一个以比较元素为基础的排序算法可以按照比较的顺序构建出一棵决策树（Decision-Tree）**
  - 决策树是一棵满二叉树，表示在给定输入规模情况下，某一特定排序算法对所有元素的比较操作
  - 每个内部结点标记为 $i:j\ (1 \le i, j \le n)$，表示一对元素 a_i 与 a_j 的比较
  - 每个叶结点上标注一个排序序列 $\langle \pi(1), \pi(2), \cdots, \pi(n) \rangle$，$n$ 个元素的 $n!$ 种可能的排列都应该出现在决策树的叶结点上

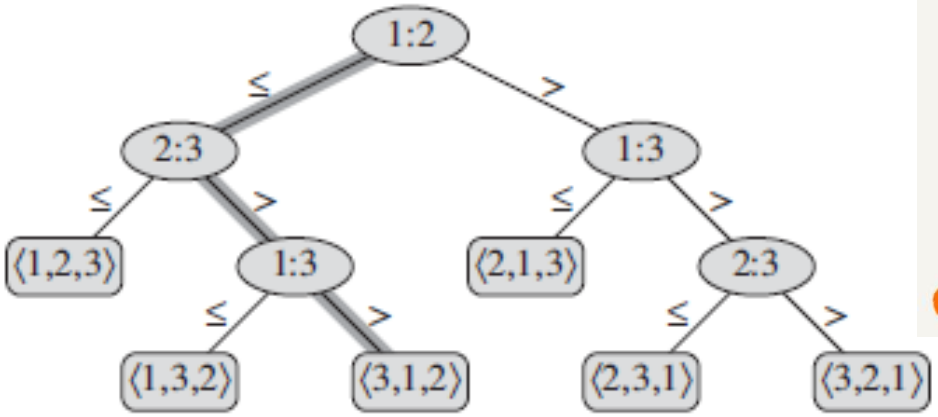
排序算法时间的下界

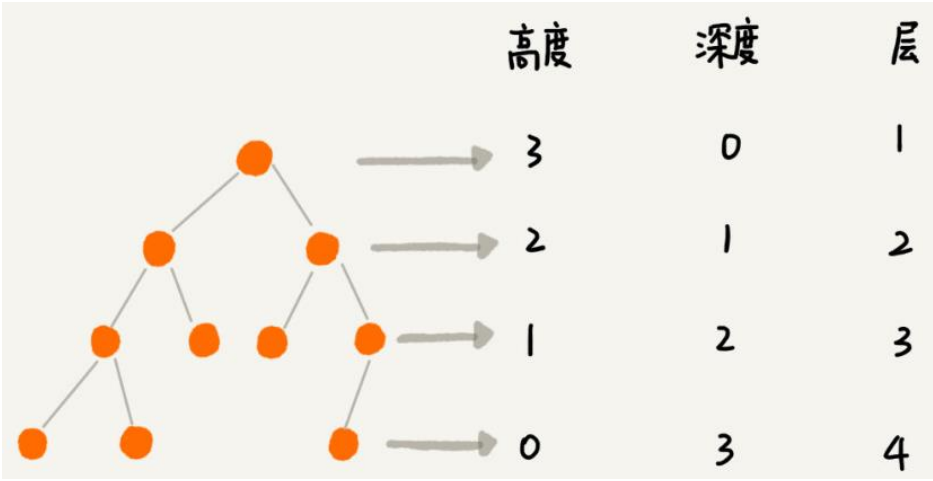


- 若$a_i \le a_j$，进入下一级的左分支
- 若$a_i > a_j$，进入下一级的右分支

- 算法在叶结点终止：排序算法的执行对应于一条从树的根结点到叶结点的路径。当到达一个叶结点时，表示排序算法已经确定一个排序序列 $a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant \cdots \leqslant a_{\pi(n)}$
- 路径长度表示生成该排序序列所需要的比较次数

排序算法时间的下界





- 在决策树中，从根结点到任意一个可达叶结点之间**最长简单路径的长度**表示对应的排序算法在**最坏情况下所作的比较次数，也就是决策树的高度**
- 因此，**决策树的高度下界**也就是比较排序算法**在最坏情况下运行时间的下界**

排序算法时间的下界

- **决策树的性质有：**
  - ✓ 由于 n 个关键字有 $n!$ 种可能的排列，所以决策树中将有 $n!$ 个叶结点
  - ✓ 一棵高为 h 的二叉决策树中，最多有 2^h 个叶结点（完全二叉树），因此有 $n! \leq 2^h \rightarrow h \geq \lg(n!) = \Omega(n\lg n)$
- **因此，任何以比较操作为基础的排序算法的最坏情况下运行时间的下界为：$\Omega(n\lg n)$**

排序算法时间的下界

- 堆排序和归并排序是**渐近最优的排序算法**，因为它们在最坏情况下运行时间上界 $O(n\lg n)$ 与上述给出的最坏情况下界 $\Omega(n\lg n)$ 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归并排序更好







谢谢！

Q & A

作业：7.1-3，7.4-1　7.4-2