Part II

Sorting and Order Statistics

songyou@buaa.edu.cn

Sorting and Order Statistics

Chapter 6 Heapsort

- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- Priority queues

Chapter 7 Quicksort

- Description and Performance
- A randomized version of quicksort
- Analysis of quicksort

Chapter 8 Sorting in Linear Time*

Chapter 9 Medians and Order Statistics

- Minimum and maximum
- Selection in expected linear time

6 Heapsort

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

调整二叉树的元素位置，
使其为最大堆

```
BUILD-MAX-HEAP(A)
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

调整二叉树的元素位置，
使其为最大堆

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

调整二叉树的元素位置，
使其为最大堆

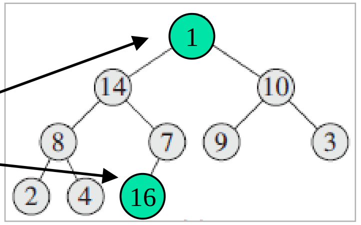
堆排序算法：

1. 建堆（得到最大堆）
2. 交换元素（最大元素定位到最后一个位置）
3. 维护最大堆（序列操作中，最大堆性质被破坏了【上一步中最大元素被改变】），回到第2步

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

step1

step2



step1. Build: 建一个最大堆

step2. 交换堆中的首尾元素

（最大元素放最后，并把堆减小1）

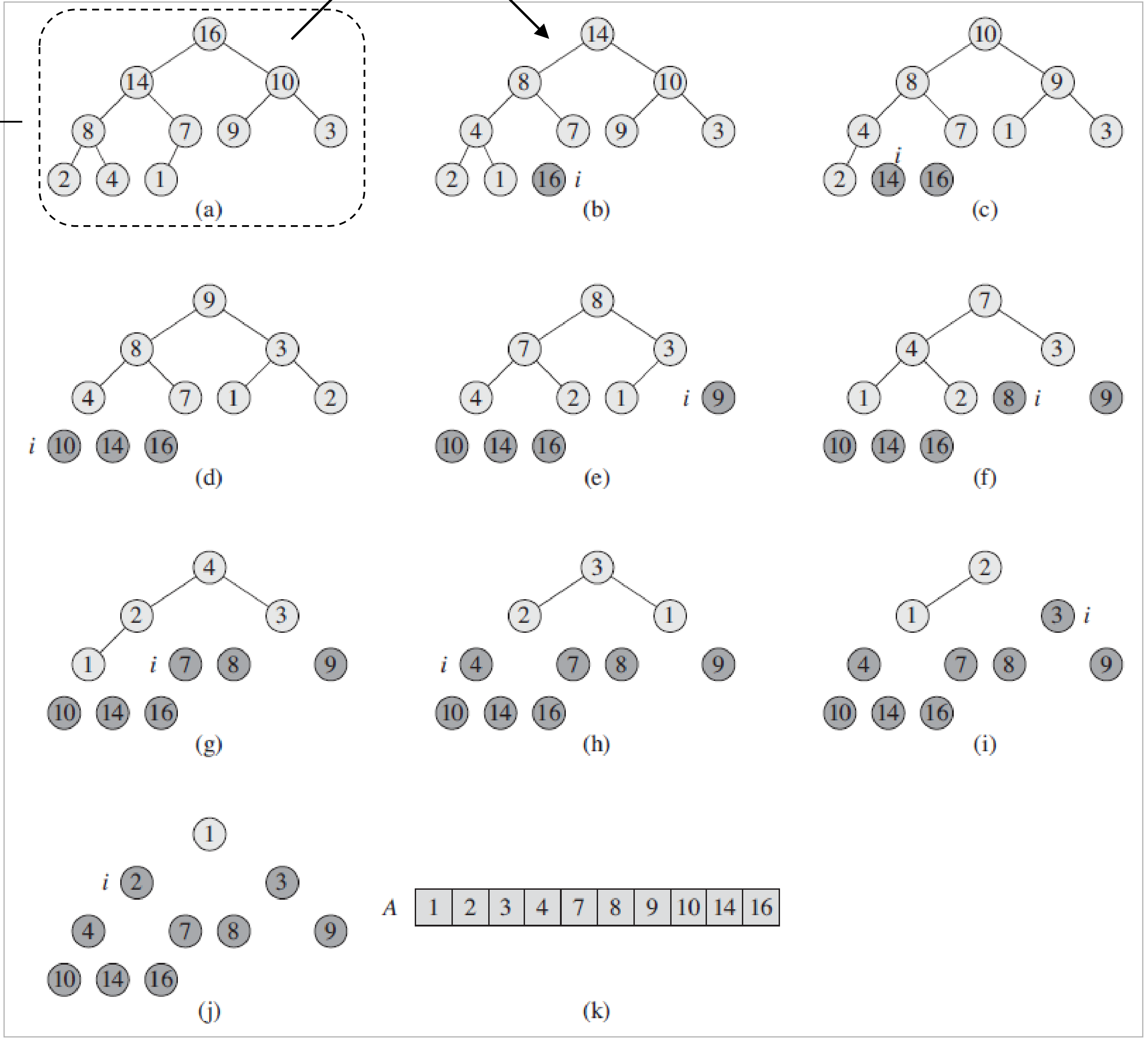
step3. 维护最大堆

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

建好最大堆

2

3




```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

6 Heapsort

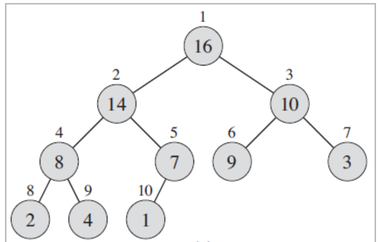
- Running time: $\Theta(n\lg n)$
- Using a data structure “heap” to manage information
- Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue

Applications:

We use min-heaps to implement min-priority queues in Chapters 16 (Greedy Algorithms),
Chapters 23 (Minimum Spanning Trees),
Chapters 24 (Single-Source Shortest Paths).

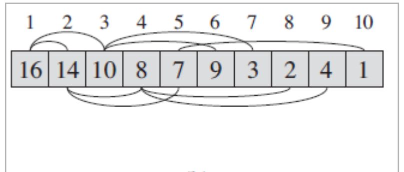
6.1 Heaps

工程模型 →



(a)

物理模型 →



(b)

数学模型
or 计算模型

- (binary) **heap : complete binary tree** (priority queue)

  (二叉树的深度为 h，除第 h 层外，其它各层 (1 ~ h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边)
- A: An array can represent a heap
- $A.length$: the number of elements in the array;

  $A.heap\text{-}size$: the number of elements in the heap

  $(0 \le A.heap\text{-}size \le A.length)$

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

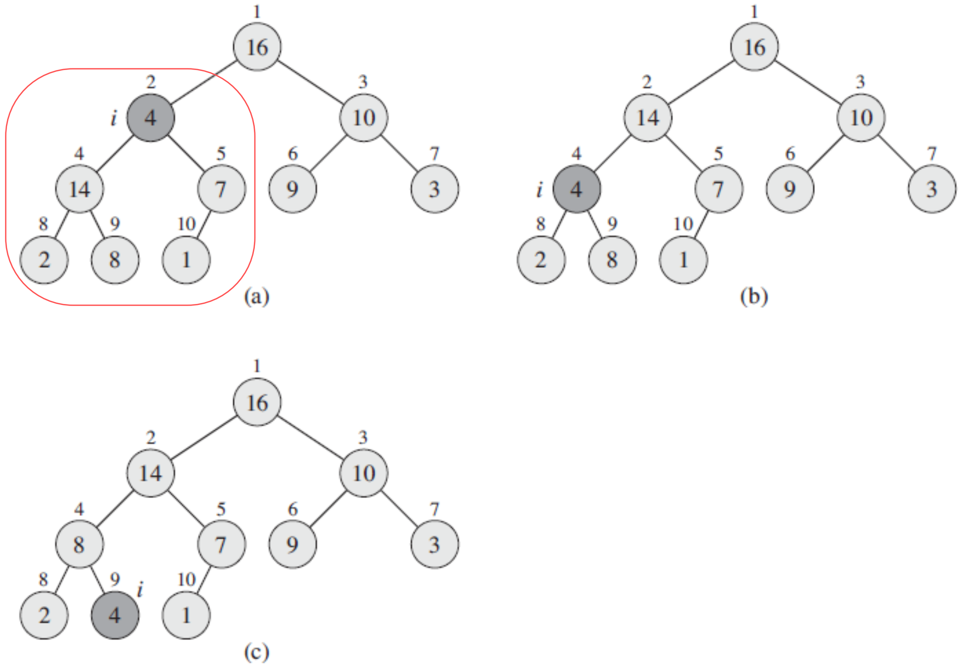
RIGHT(i)
return $2i + 1$

- max-heap : $A[\text{PARENT}(i)] \ge A[i]$, for every node i other than the root.
- min-heap : ?

6.2 Maintaining the heap property

- **MAX-HEAPIFY** assumes that the trees rooted at **LEFT**(i) and **RIGHT**(i) are max-heaps, but that $A(i)$ might be smaller than its children, thus violating the max-heap property.
  （维护最大堆：二叉树以节点 i 为根，记为二叉树 i，假设以 **LEFT**(i) 和 **RIGHT**(i) 为根的子树都为最大堆，但二叉树 i 不是最大堆）
- **MAX-HEAPIFY** lets the value at $A(i)$ “float down” in the max-heap.

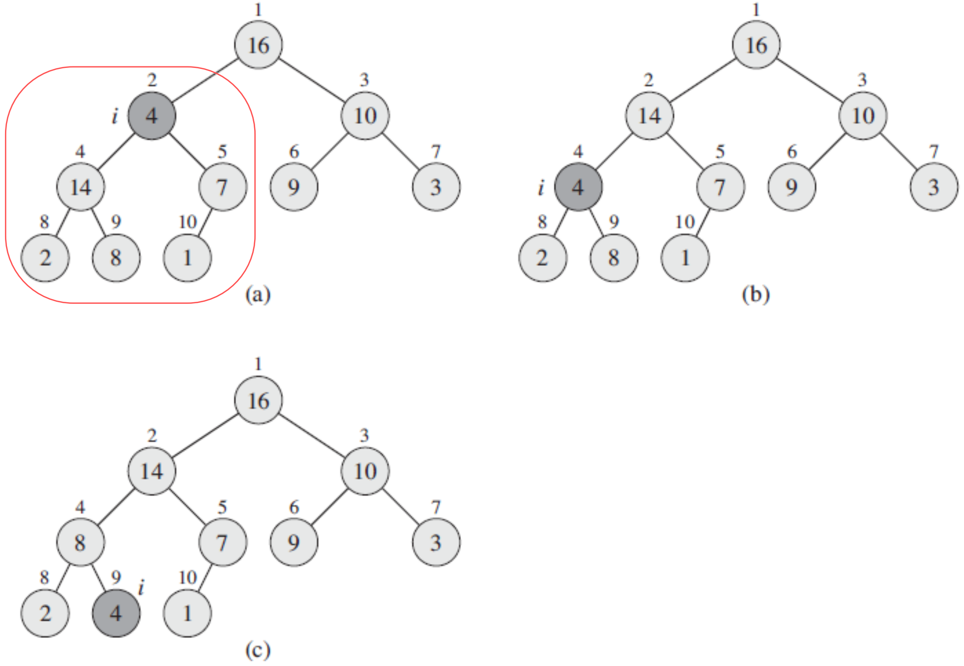
```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```



- The running time?

6.2 Maintaining the heap property

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```



- The running time?

  For node i , the children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full.

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

  Answer ? Master method.

- In fact, the running time of **MAX-HEAPIFY** on a node of height h is $O(h)$.

6.3 Building a Heaps

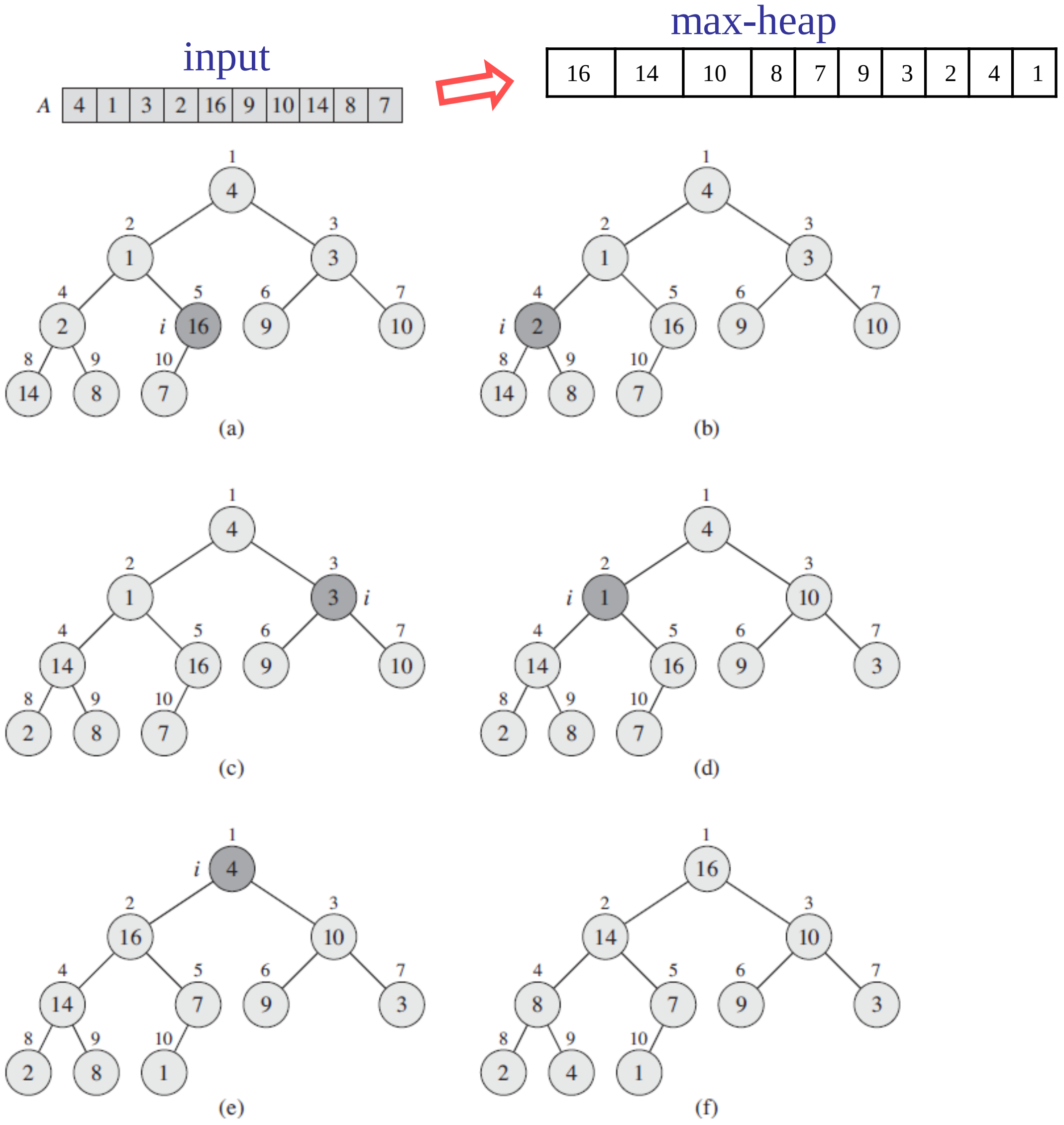
- We use the procedure **MAX-HEAPIFY** in a bottom-up manner to convert an array $A[1 \,..\, n]$ into a max-heap.
- The elements in the subarray $A[(\text{floor}(n/2) + 1) \,..\, n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. (Exercise 6.1-7)

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

- Correct ?

  Loop invariant.

- The running time?



6.3 Building a Heaps

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

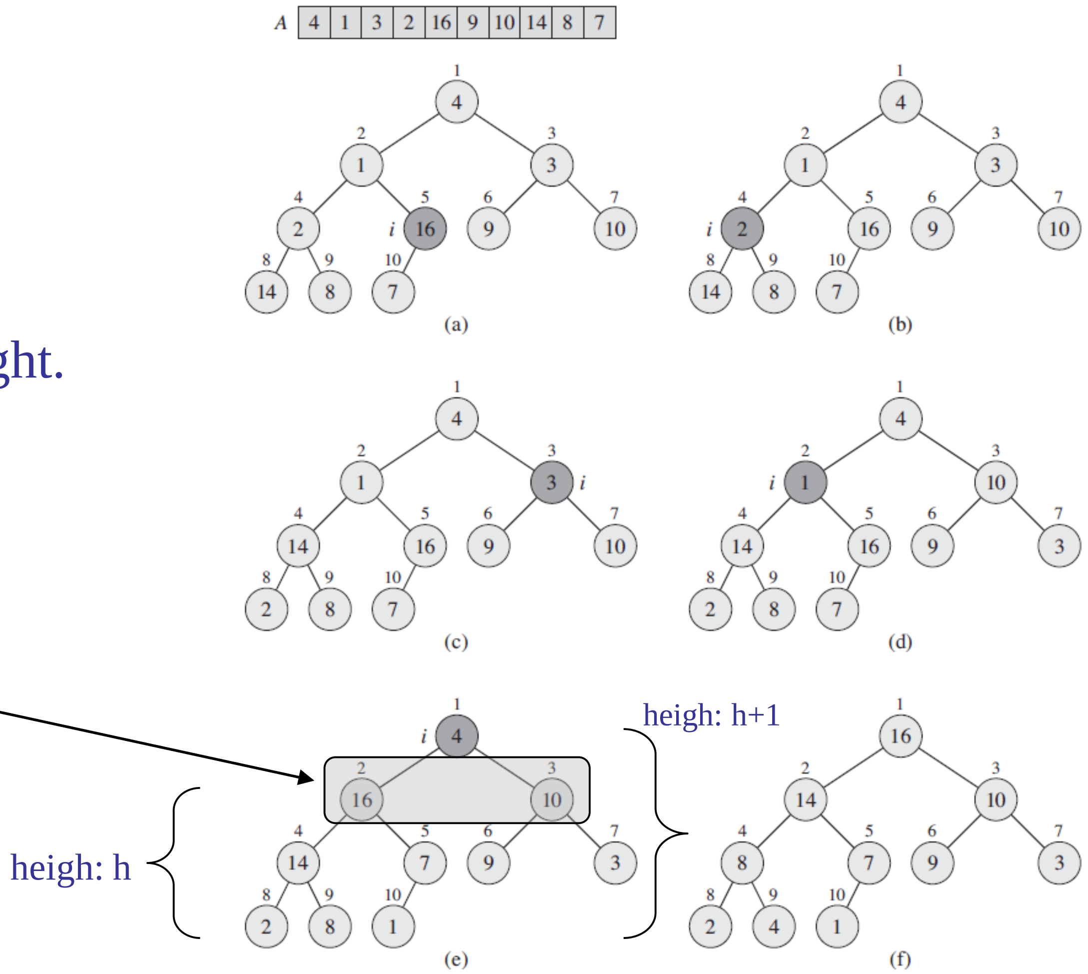
- The running time?

  O($n$lg$n$), correct, but not asymptotically tight.

- at most ceil($n/2^{h+1}$) nodes of any height h (Exercise 6.3-3)

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

$$O\left(n\sum_{h=0}^{\lfloor \lg n\rfloor}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right) = O(n)\,.$$



$O(n)$ 在高度 h 的地方，最多有 $n/(2^{(h+1)})$ 个节点，每个节点调用MAX-HEAPIFY时花的时间为 $O(h)$，高度从0到 lg$n$.

直观地说，$n/2$ 个节点中，高度为 lg$n$ 的节点1个（根节点），高度为 (lg$n$)-1 的节点2个，高度为 (lg$n$)-2 的节点4个，……，也就是说，大部分节点的高度很小，并不是所有的 $n/2$ 个节点的高度都是 lg$n$

6.4 The heapsort algorithm

The running time of Heapsort ?

$O(n\lg n)$

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

n

$O(n)$

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

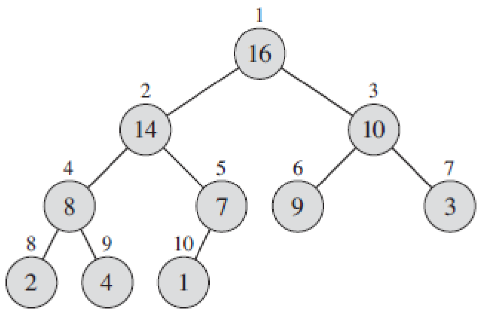
$O(h)$

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

6.5 Priority queues

- One of the most popular applications of a heap:
    - An efficient priority queue (max-priority, min-priority)
- Applications:

6.5 Priority queues

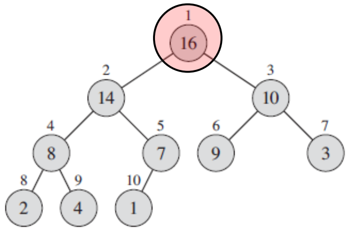


- **A max-priority queue supports the following operations:**
  - **MAXIMUM(**S**)**: returns the element of S with the largest key.
  - **EXTRACT-MAX**(S): removes and returns the element of S with the largest key. （移走并返回最大元素）
  - **INCREASE-KEY**(S, i, k): increases the value of element i's key to the new value k.
  - **INSERT(**S, x): inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.
- **Applications:**
  - A max-priority queue: schedule jobs on a shared computer.
  - A min-priority queue: an event-driven simulator.

Operations of a max-priority queue

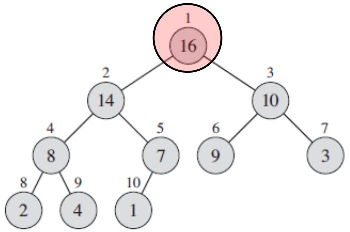
(1) 读最大元素 $\Theta(1)$

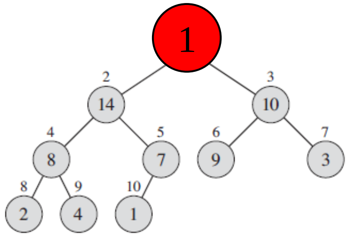
```
HEAP-MAXIMUM(A)
    return A[1]
```

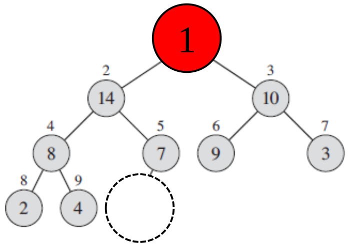


(2) 移走最大元素 $O(\lg n)$

```
HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1
        error “heap underflow”
    max = A[1]
    A[1] = A[A.heapsize]
    A.heapsize--
    MAX-HEAPIFY(A, 1)
    return max
```





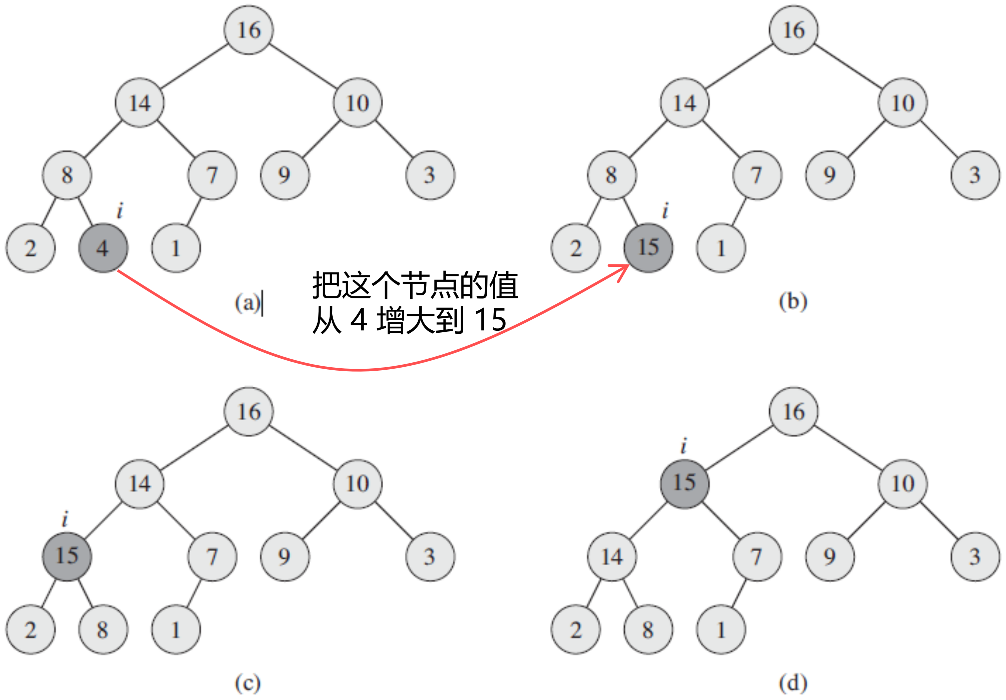


Operations of a max-priority queue

(3) 把某个节点的值增加（增大）

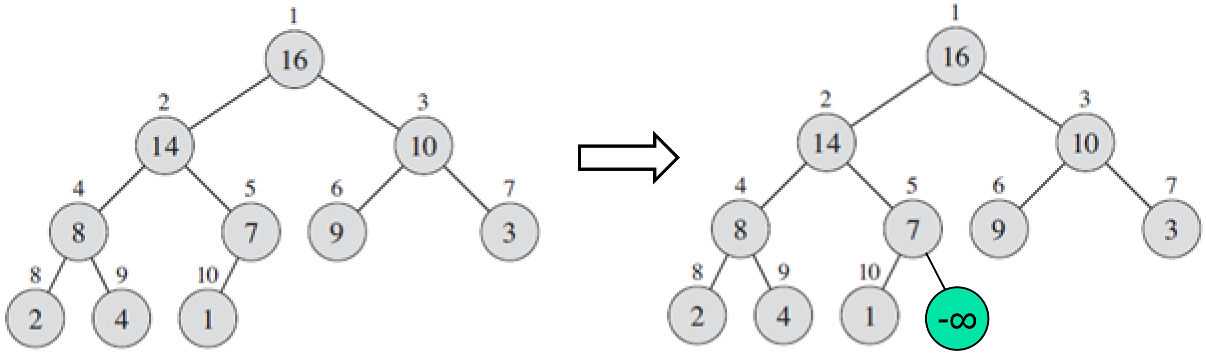
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

$O(\lg n)$



Operations of a max-priority queue

```
HEAP-INCREASE-KEY(A, i, key)                O(lgn)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



(4) 插入一个新节点

```
MAX-HEAP-INSERT(A, key)                     O(lgn)
A.heapsize++
A[A.heapsize] = -∞
HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Exercise for chapter 6

- 把课本上最大堆、堆排序、最大优先队列的所有算法程序实现
- 用最小堆重复 chapter6
- 堆排序是否是稳定的?

7 Quicksort

- Worst-case running time: $\Theta(n^2)$
- Expected running time: $\Theta(n\lg n)$
- Quicksort is often the best practical choice for sorting because it is remarkably efficient on the average. The constant factors hidden in the $\Theta(n\lg n)$ are quite small.

7.1 Description of quicksort

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

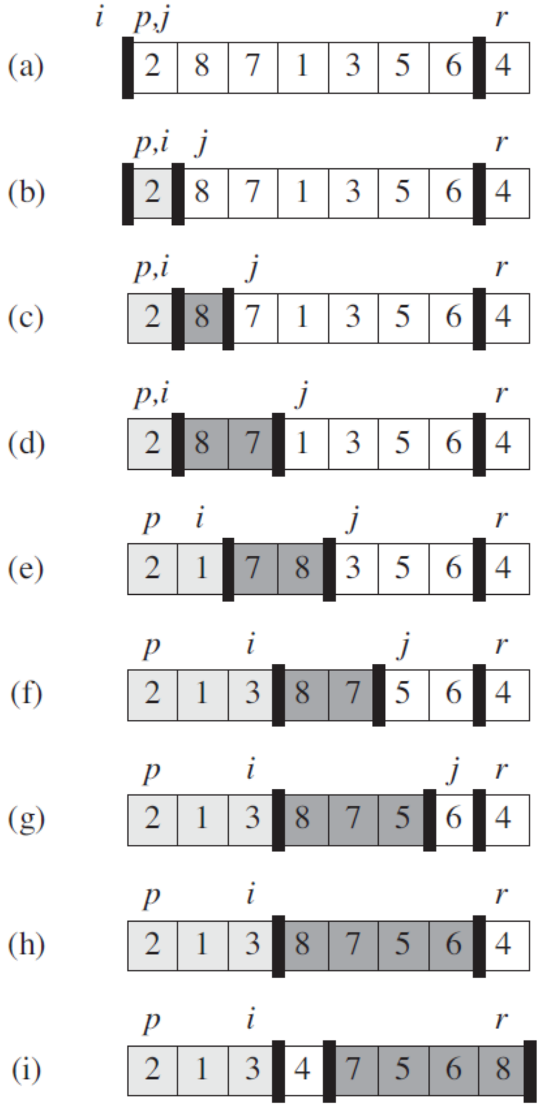
```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

分区算法 PARTITION 执行过程中，i 之前（含）的元素小于等于哨兵（标记） $A[r]$， i+1 及其之后的元素大于 $A[r]$。
即 $p \sim i$ 的元素小于等于 $A[r]$，其后元素 $A[i+1] \sim A[j-1]$ 比 $A[r]$大。
最后，交换$A[i+1]$与$A[r]$，以保证$A[r]$ 之前的元素都比它小，之后的元素比它大。

$A[p \sim i]$, $\leq A[r]$

$A[i+1 \sim r-1]$, $> A[r]$

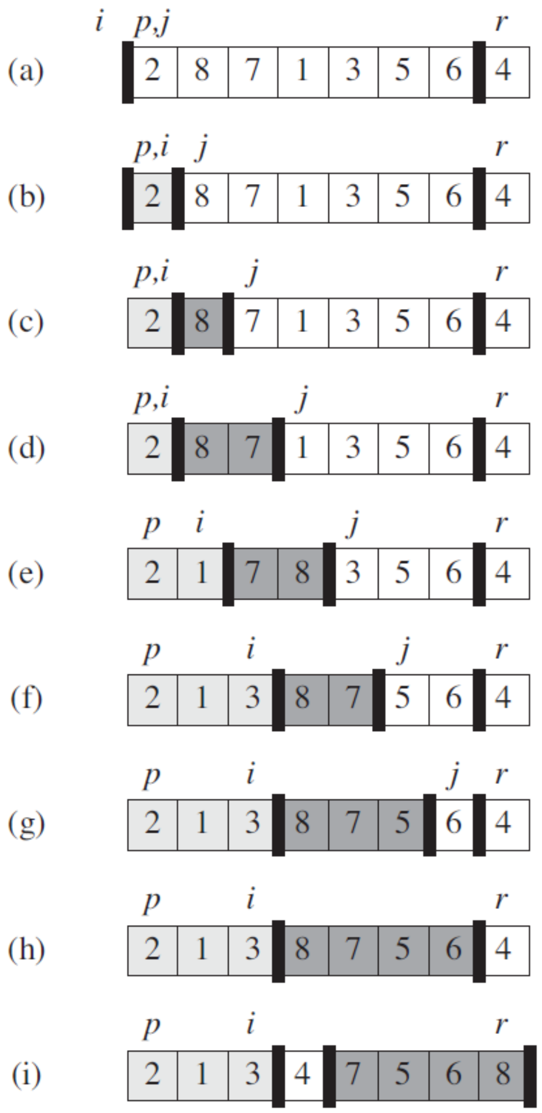
swap($A[i+1]$, $A[r]$), when termination $0 \leq i \leq r-1$



7.2 Performance of quicksort

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



- Worst-case partitioning ($\in$ Unbalanced)

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

- Best-case partitioning ($\in$ Balanced)

  $T(n) = 2T(n/2) + \Theta(n)$? ← strong-balanced

- Balanced partitioning (e.g.)

  $T(n) = T(9n/10) + T(n/10) + \Theta(n)$?

  $T(n) = T(99n/100) + T(n/100) + \Theta(n)$? ← weak-balanced

- Running time for the average case?

7.2 Performance of quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Worst-case partitioning

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

  $= n + n\text{-}1 + T(n\text{-}2) = \ldots = n + n\text{-}1 + \ldots + 1 = \Theta(n^2)$

什么情况下出现最坏分区？

- Best-case partitioning

  $T(n) = 2T(n/2) + \Theta(n) \Rightarrow$ Master method: $\Theta(n\lg n)$

什么情况下出现最好分区？

7.2 Performance of quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Worst-case partitioning

$$T(n) = T(n\text{-}1) + T(0) + \Theta(n) = \Theta(n^2)$$

- Unbalanced partitioning

$$T(n) = T(n\text{-c-}1) + T(\text{c}) + \Theta(n)\ ?$$

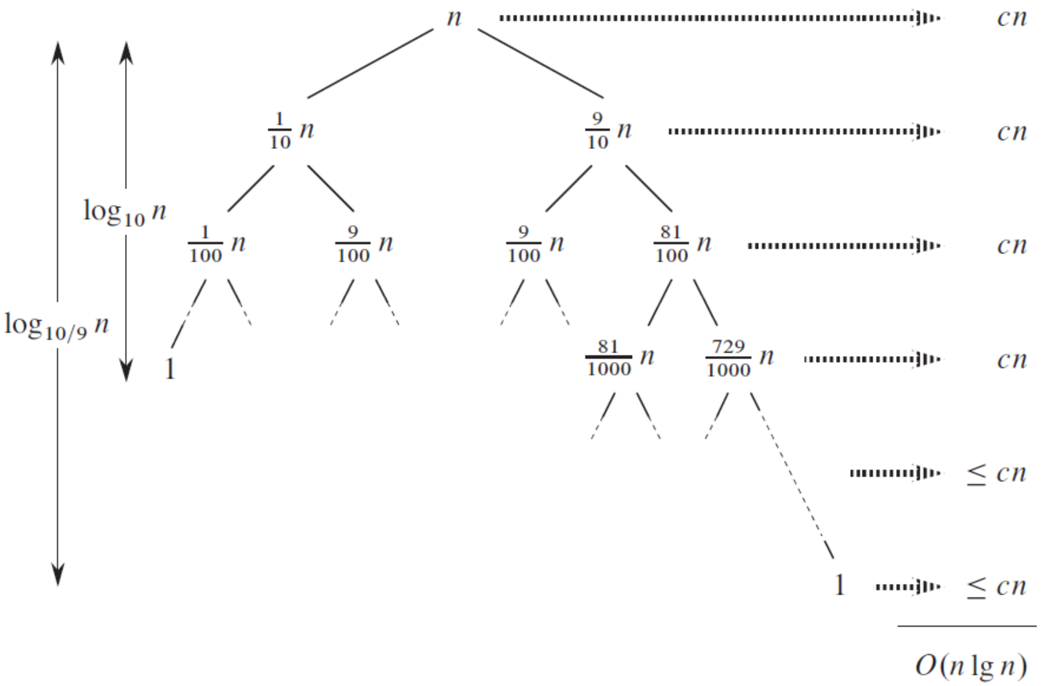
7.2 Performance of quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Balanced partitioning (e.g.)

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$?



树的最小高度： $(n/10)^L = 1$ 时， => $L = \lg n/\lg 10$

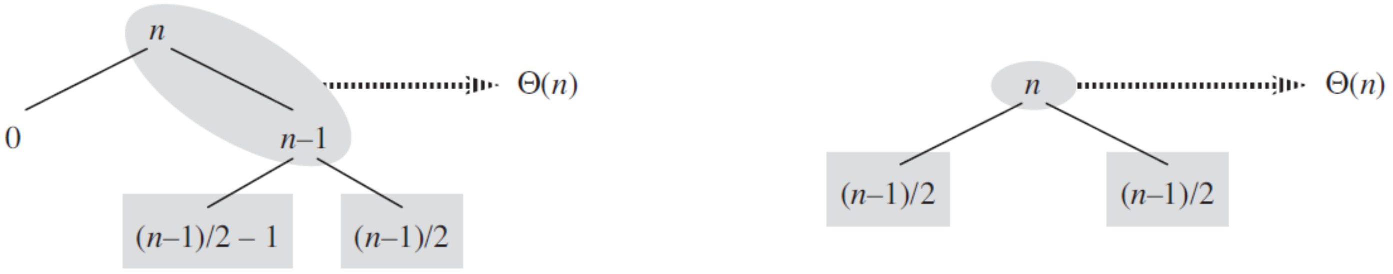
树的最大高度： $(9n/10)^H = 1$ 时， => $H = \lg n/\lg(10/9)$

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Average case, $T(n) = ?$

Intuitively, the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits.



(a) (b)

假设最好和最坏分区交叉出现，则“分区树”的高度为 $2*\lg n$，每一层的时间为 n，则得 $O(n\lg n)$

7.3 Randomized-quicksort

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

RANDOMIZED-PARTITION，随机分隔：每次分隔（分区）从数组里随机选一个数，把它定位到它在数组里的顺序位置。

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

7.4 Analysis of quicksort

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Worst-case analysis

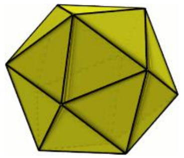
  $$T(n) = \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1) \right) + \Theta(n)$$

  Substitution method, $\Theta(n^2)$

- Expected running time?
  - Indicator random variables
  - Intuitively…

7.4 Analysis of quicksort (1)

```
Times-QS            // 运行QS若干次
for i = 1 to m      // (m = k·n)
    RANDOMIZED-QUICKSORT(A, p, r)
```



```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Intuitively…

Run RANDOMIZED-QUICKSORT m times (Roll a dice with n points m times , each point has k times, $m = n\,k$)

$$
n \left\{
\begin{array}{ll}
T(n) = T(n-1) + T(0) + \Theta(n) & \text{--------}\ k \text{ times} \\
T(n) = T(n-2) + T(1) + \Theta(n) & \text{--------}\ k \text{ times} \\
T(n) = T(n-3) + T(2) + \Theta(n) & \text{--------}\ k \text{ times} \\
\ldots & \\
T(n) = T(n-n/2) + T(n/2-1) + \Theta(n) & \text{--}\ k \text{ times} \\
\ldots & \\
T(n) = T(2) + T(n-3) + \Theta(n) & \text{--------}\ k \text{ times} \\
T(n) = T(1) + T(n-2) + \Theta(n) & \text{--------}\ k \text{ times} \\
T(n) = T(0) + T(n-1) + \Theta(n) & \text{--------}\ k \text{ times}
\end{array}
\right.
$$

Idea of proof:
不妨令 $k = 1$，设有 x 个 un-Balanced partitioning，则有 n-x 个 Balanced partitioning ，则the running time of Times-QS is ($x << n$?):

$$
\begin{aligned}
&\frac{xn^2 + (n-x)n\lg n}{n} \\
&= \frac{xn^2 + n^2\lg n - xn\lg n}{n} \\
&= xn + n\lg n - x\lg n \\
&\le xn + n\lg n \\
&\le n\lg n + n\lg n \quad \cdots (\text{if } x \le \lg n) \\
&= 2n\lg n
\end{aligned}
$$

7.4 Analysis of quicksort (2)

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, 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
```

- Expected running time? (use Indicator random variables)
- running time X: the number of comparisons performed in line 4 of PARTITION. (平均的元素比较次数)
- For ease of analysis, we rename the elements of the array A as $z_1, z_2, \dots, z_n$
- $Z_{ij} = \{ z_i, z_{i+1}, \dots, z_j \}$: set of elements between z_i and z_j
- Indicator random variables:

  $X_{ij} = \mathrm{I}\{ z_i \text{ is compared to } z_j \}$

  $E[X_{ij}]$：任意两个元素 z_i 和 z_j 的平均比较次数

- The total number of comparisons (running time of quicksort)

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

7.4 Analysis of quicksort (2) - **Indicator random variables**

```
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)
```

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

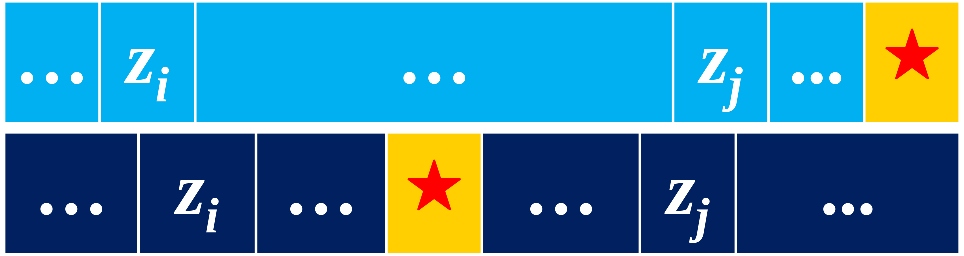
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- X : the number of comparisons performed in line 4 of PARTITION.
- $Z_{ij} = \{ z_i , z_{i+1} , \dots , z_j \}$
- Indicator random variables:

  $X_{ij} = \mathrm{I}\{ z_i \text{ is compared to } z_j \}$
- The total number of comparisons

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

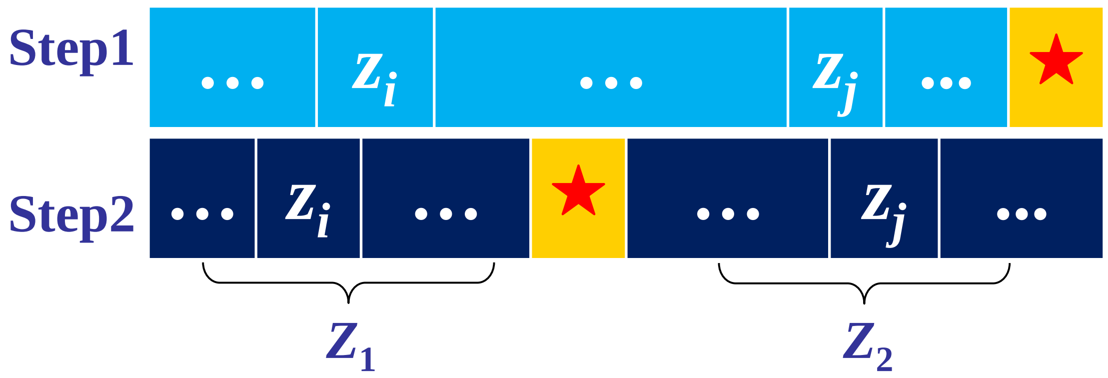
- 整个快排过程中，两个数 z_i and z_j 最多比较一次， when?



7.4 Analysis of quicksort (2) - Indicator random variables

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

整个快排过程中，两个数 z_i and z_j 最多比较一次，出现在分区时产生了子序列 Z_{ij}



Step1: 快排过程中，出现序列 Z，哨兵（标记）元素为 ★

Step2: 对序列 Z 分区后，产生两个子序列 Z_1 和 Z_2

Z 中的其他元素仅与 ★ 比较一次

(1) 如果 z_i 或 z_j 为 ★ ，则两者比较一次，此后不再相遇（不会比较）

(2) 如果 z_i 与 z_j 分别被分区到 Z_1 与 Z_2，则两者无比较，此后也不会相遇（过去没有，此时没有，将来也没有）

(3) 如果 z_i 与 z_j 被分区到同一 Z_1 或 Z_2，重复Step1和Step2的逻辑

7.4 Analysis of quicksort (2)

- Indicator random variables:

  $X_{ij} = \mathrm{I}\{\ z_i \text{ is compared to } z_j\ \}$

- The total number of comparisons

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}
\end{aligned}
$$

$$
\begin{aligned}
&\Pr\{z_i \text{ is compared to } z_j\} \\
&= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}
$$

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

7.4 Analysis of quicksort (3) - why is quicksork quick?

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

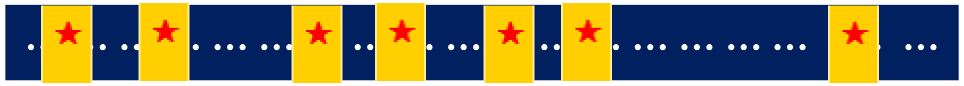
```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Intuitively…

第 1 次分区，定位好 1 个元素

第 2 次分区，又定位好 2 个元素（已定位 2^2-1 个）



第 3 次分区，又定位好 4 个元素（已定位 2^3-1 个）

……

第 k 次分区，又定位好 2^{k-1} 个元素（共定位 2^k-1 个）

$2^k - 1 = n \Rightarrow k = \lg(n+1)$

每次分区有最多 n-1 次比较 ⇨ $O(n\lg n)$

Exercise for chapter 7

随机产生一组数据（如1M），多次运行 quicksort 算法，观察 partition 算法中第4行（元素比较）执行的次数，并对你的观察结果进行思考。

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, 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
```

Sorting in Linear Time

- ✓ counting sort （计数排序）
- ✓ *radix sort （基数排序）
- ✓ *bucket sort（基数排序）

These algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

Lower bounds for comparison sort

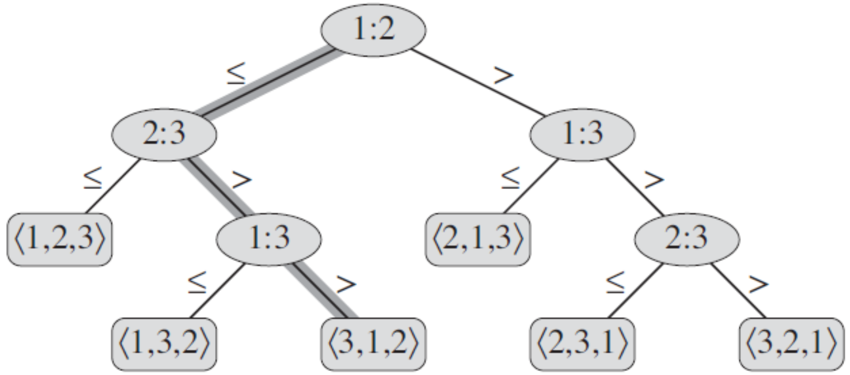
- Algorithms
  - ✓ bubble, select, insert, merge, heap, quick, …
- Comparison sort
  - ✓ The sorted order they determine is based only on comparisons between the input elements.
  - ✓ We use only comparisons between elements to gain order information about an input sequence $<a_1, a_2, \ldots, a_n>$. That is, given two elements a_i and a_j, without loss of generality, we perform only comparison $a_i \le a_j$.
- Running time

  $\Omega(n\lg n)$?　（计算时间至少为 $n\lg n$，最好情况除外）

The decision-tree model

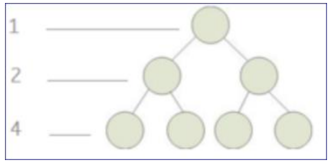
- We can view comparison sorts abstractly in terms of decision trees.
- Decision tree: is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

（给定某个输入，某种算法执行时，由元素之间的比较而产生的一个满二叉树）

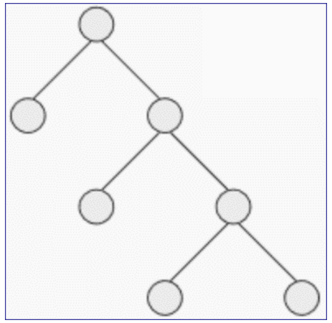


Example: The decision tree for insertion sort operating on three elements

“中国版”的满二叉树

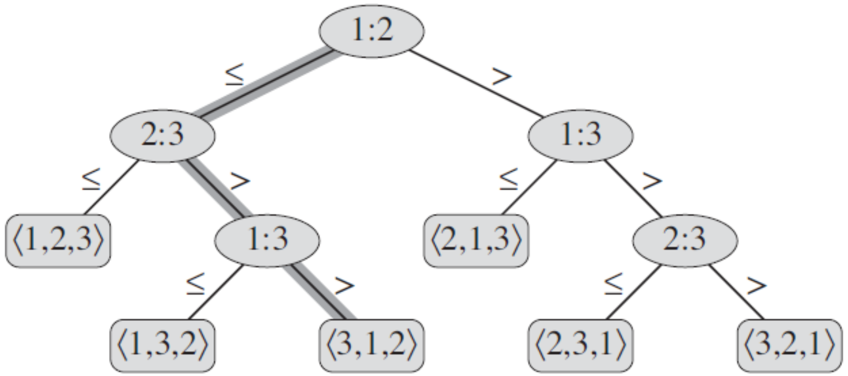


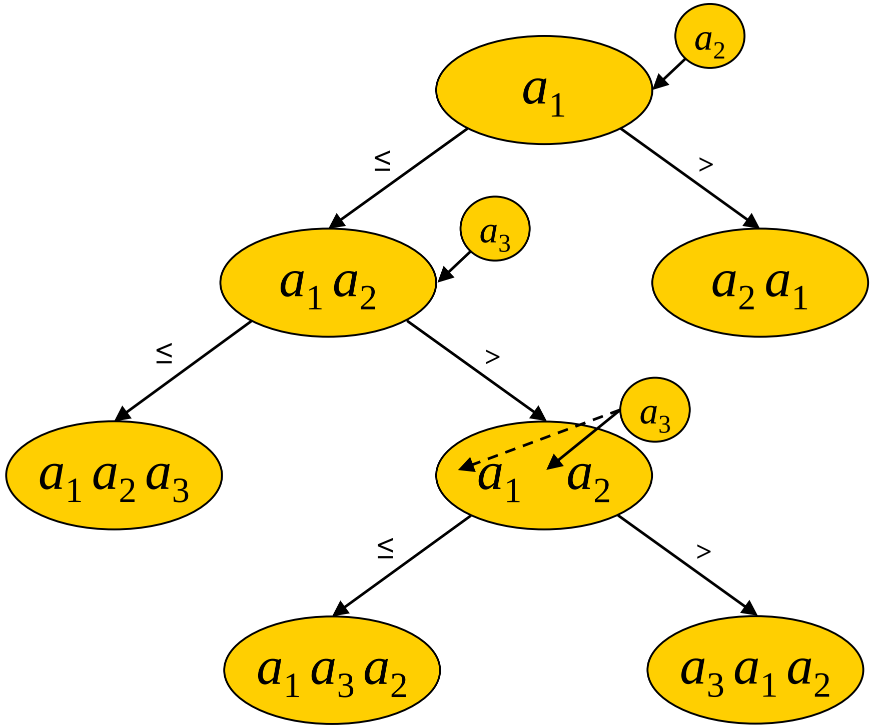
“外国版”的满二叉树（结点要么叶子，要么有两个孩子）



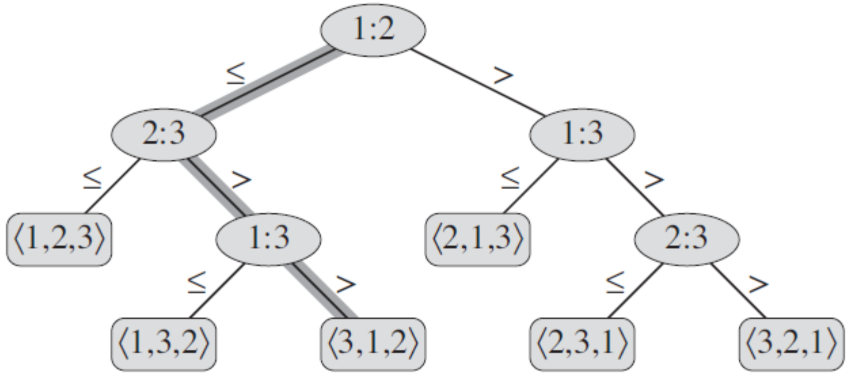
The decision-tree model

Example: The decision tree for insertion sort operating on three elements





The decision-tree model



- In a decision tree, each leaf is a permutation (a solution of sort) $<\pi(1), \pi(2), \ldots, \pi(n)>$ of $<1, 2, \ldots, n>$

  决策树中，输出是某一个叶节点，表示输入的一个置换（或排列）（排序算法的解）

- There have $n!$ permutations of $<1, 2, \ldots, n>$

  n 个数的置换有种 $n!$

- A correct sorting algorithm must be able to produce a permutation(leaf) that establish the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

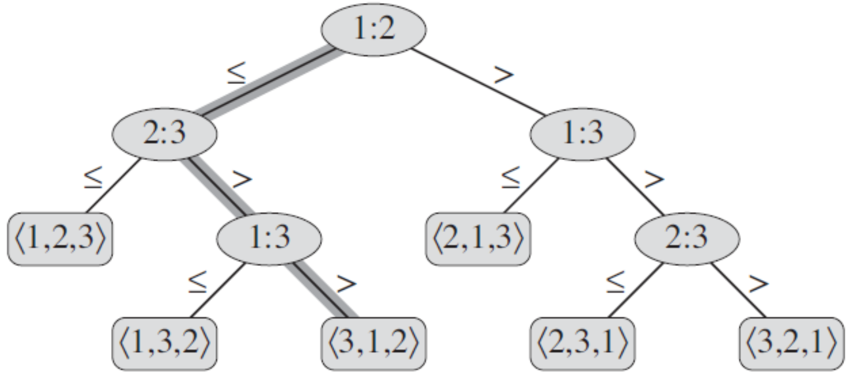
  正确的排序算法能产生一个置换（叶节点），使得该叶节点的数的顺序满足排序输出要求

- An actual execution of the comparison sort: A path from the root by a downward to the leaf. What's the height h?

  比较排序算法的执行过程：从树根到叶节点的路径。该路径的高度（长度）h 就是比较时间（计算时间）。

The decision-tree model

- What's the height h for a decision tree corresponding to a comparison sort?
  一种比较排序算法的决策树的高度 h 是多少?



- A comparison sort on n elements: $n!$ permutations.
  一种比较排序算法对应一个决策树，n 个元素的置换有 n 种（排序输出是其中的一种）。
- For a decision tree（对应某种比较排序算法的一个决策树）
  排序算法的输出是某一个叶节点，计算时间是从树根到该叶节点的长度（数的高度）
  - leaves（叶节点个数）: l
  - height （决策树高度）: h

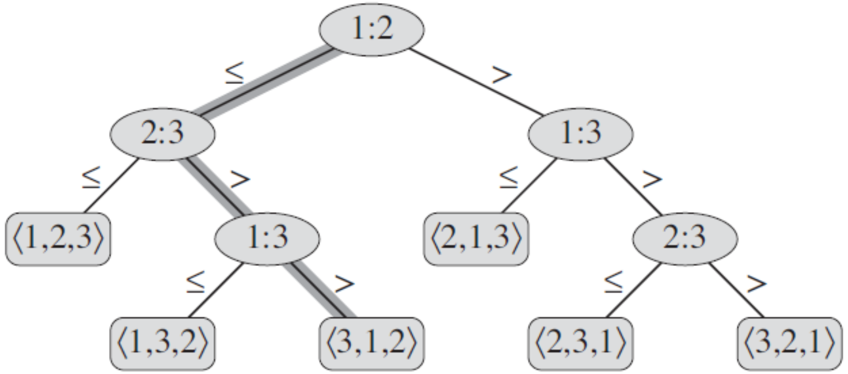
$n! \le l$：叶子数 $\ge n$ 排列数，保证叶子包括了所有可能的解。一个解是一个叶子。

$l \le 2^h$：对高度为 h 的二叉树，叶子数最多为 2^h

$$n! \le l \le 2^h$$

$$h \ge \lg(n!) = \Omega(n\lg n) \ ?$$

The decision-tree model



$$n! \le l \le 2^h$$

$$h \ge \lg(n!) = \Omega(n\lg n)$$

A tournament problem

LR Ford Jr, SM Johnson - The American Mathematical Monthly, 1959 - Taylor & Francis

… In his book, t Steinhaus discusses the problem of ranking n objects according to some … In this paper we shall adopt the terminology of a tennis tournament by n players. The problem may …

☆ 保存 ⁹⁹ 引用 被引用次数：203 相关文章 所有 4 个版本

A tournament problem

在引用文章中搜索

Applications of network optimization

RK Ahuja, TL Magnanti, JB Orlin, MR Reddy - Handbooks in Operations …, 1995 - Elsevier

Publisher Summary This chapter discusses several applications of the following network optimization problems: shortest paths, maximum flows, minimum cost flows, assignment and …

☆ 保存 ⁹⁹ 引用 被引用次数：187 相关文章 所有 13 个版本

[图书] Introduction to algorithms

TH Cormen, CE Leiserson, RL Rivest, C Stein - 2022 - books.google.com

A comprehensive update of the leading algorithms text, with new material on matchings in bipartite graphs, online algorithms, machine learning, and other topics. Some books on …

☆ 保存 ⁹⁹ 引用 被引用次数：66600 相关文章 所有 91 个版本 »

9 Medians and Order Statistics

The ith **order statistic** of a set of n elements is the ith smallest element.

- ✓ the **minimum** of a set of elements is the first order statistic ($i = 1$).
- ✓ the **maximum** is the nth order statistic ($i = n$).
- ✓ A **median**, informally, is the "halfway point" of the set.

6	12	5	9	2	10	8	7

Minimum: 2 （1阶顺序统计量）

Maximum: 12 （n阶顺序统计量）

9 Medians and Order Statistics

- For convenience, consider the problem of selecting the ***i*th order statistic** from a set of n distinct numbers. （从 n 个不同值的数里找到 i 阶顺序统计量）
- We can solve the selection problem in O(***nlgn***) time, since we can sort the numbers and then simply index the ith element in the output array. Can we do it better?

6	12	5	9	2	10	8	7

2	5	6	7	8	9	10	12

The 5th **order statistic** is 8

9.1 Minimum and maximum

```
MINIMUM(A)
    min = A[1]
    for i = 2 to A.length
        if min > A[i]
            min = A[i]
    return min
```

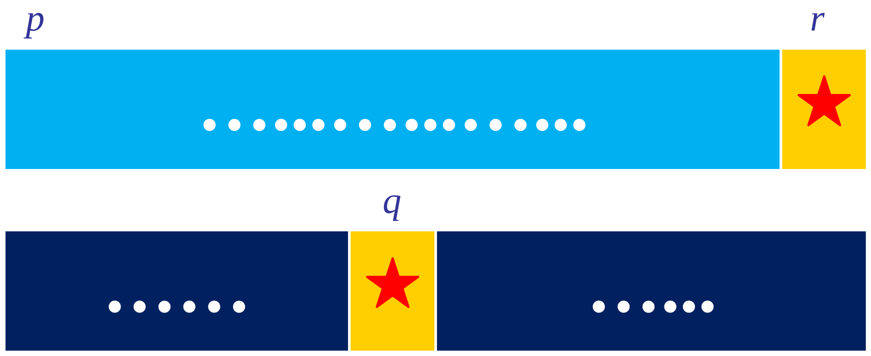
n-1 comparisons

9.2 Selection in expected linear time

- The general selection problem appears more difficult than the simple problem of finding a minimum.
- Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$.

求第 i 小的元素　　Firstly, p is 1, r is n

```
RANDOMIZED-SELECT(A, p, r, i)
    if p == r
        return A[p]
    q = RANDOMIZED-PARTITION(A, p, r)
    k = q - p + 1
    if i == k   // the pivot value is the answer
        return A[q]
    elseif i < k
        return RANDOMIZED-SELECT(A, p, q-1, i)
    else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```



// 随机分区，找到 A 中的第 k 小的元素 $A[q]$

// 左边的 k-1 个元素比 $A[q]$ 小，第 k 大的元素是 $A[q]$

9.2 Selection in expected linear time

$T(n) = T(\max(k-1, n-k)) + O(n)$

- **Worst-case running time**

  $T(n) = T(n-1) + O(n)$,

  $\Theta(n^2)$

- **A special case**

  $q = (r-p)/2$, then

  $T(n) = T(n/2) + O(n)$,

  $\Theta(n)$

- **Expected running time ?**

  Indicator random variables, $\Theta(n)$? *

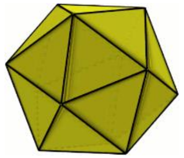
```
RANDOMIZED-SELECT(A, p, r, i)
    if p == r
        return A[p]
    q = RANDOMIZED-PARTITION(A, p, r)
    k = q - p + 1
    if i == k   // the pivot value is the answer
        return A[q]
    elseif i < k
        return RANDOMIZED-SELECT(A, p, q-1, i)
    else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

9.2 Selection in expected linear time

$T(n) = T(\max(k-1, n-k)) + O(n)$

- **Expected running time ?**

  Indicator random variables, $\Theta(n)$?



- **Intuitively,**

  Run RANDOMIZED-SELECT *m* times

  ($m = x \cdot n$: Roll a dice with *n* points *m* times , each point has *x* times.)

$$m \cdot T(n) = x \sum_{k=1}^{n} (T(\max(k-1,\ n-k)) + O(n))$$

$$T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(\max(k-1,\ n-k)) + O(n))$$

$$\leq \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) \ + \ O(n) \qquad ?$$

```
RANDOMIZED-SELECT(A, p, r, i)
    if p == r
        return A[p]
    q = RANDOMIZED-PARTITION(A, p, r)
    k = q - p + 1
    if i == k   // the pivot value is the answer
        return A[q]
    elseif i < k
        return RANDOMIZED-SELECT(A, p, q-1, i)
    else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

Using substitution,

$T(k) \leq ck \ \Rightarrow T(n) \leq cn$?

作业

- **6~9章所有的课后习题**
- **Running time?**

$$T(n) = T(n\text{-}3) + T(2) + \Theta(n)$$

$$T(n) = T(2n/7) + T(5n/7) + \Theta(n)$$

$$T(n) = T(n/a) + O(n) \quad \ldots \; (\, a > 1 \,)$$

作业

```
RANDOMIZE-IN-PLACE(A, n)
    for(i=1; i<=n; i++)
        swap(A[i], A[RANDOM(i, n)])
```

产生 1, 2, 3, ... , n (如 $n = 10^6$) 的随机置换 A （如上算法），

从 A 中取部分数据 B，如前 80/100，

在 B 中找第 k 小的数，

分别用：

排序法，$O(n\lg n)$；

chapter9.2 的随机分区法，$O(n)$。

比较两种方法分别多少次能找到（设置一个计数器来统计）。