中国科学技术大学
University of Science and Technology of China

算法基础

第四讲：线性时间排序和次第计算

主　讲:　顾 乃 杰　 教授

单　位:　计算机科学技术学院

学　期:　2015-2016 (秋)

8. Sorting in Linear Time

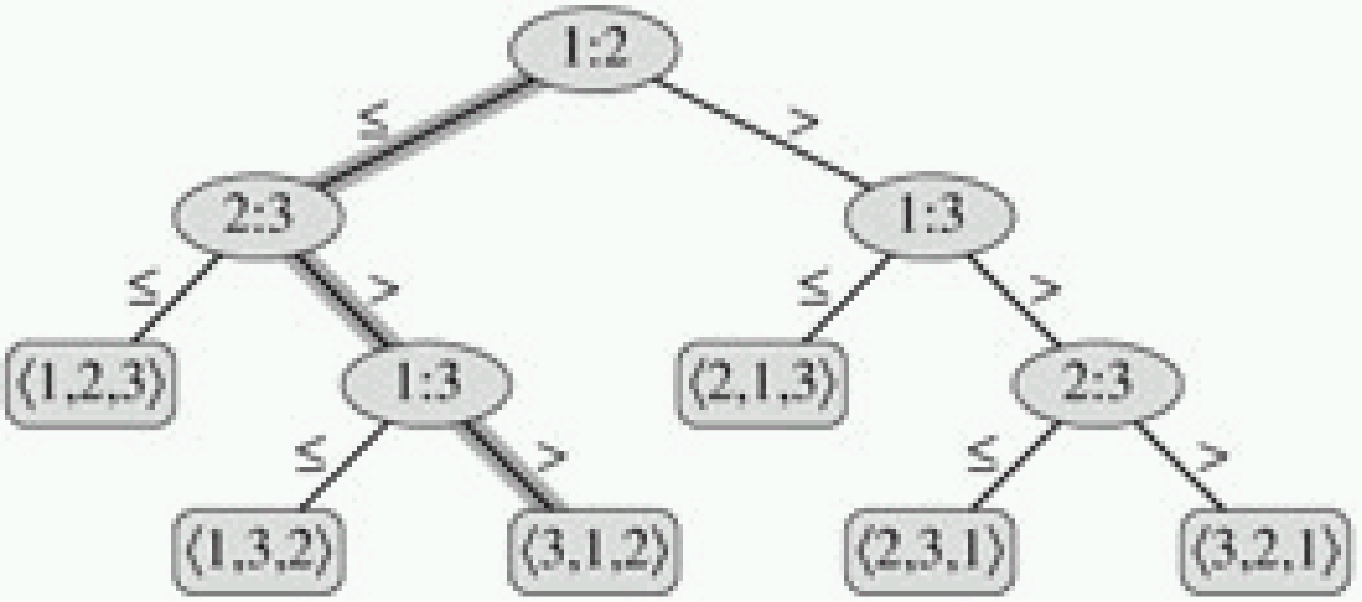
- Several algorithms introduced can sort n numbers in $O(n\log n)$ time.
- The sorted order is based only on comparisons between the input elements, so such sorting algorithms are called ***comparison sorts;***
- Any comparison sort must make $\Omega(n\log n)$ comparisons in the worst case to sort n elements。
- Three sorting algorithms—counting sort, radix sort, and bucket sort — run in linear time 。

8.1 Lower bounds for sorting

- Assume that all of the input elements are distinct，so all comparisons have the form $a_i < a_j$。
- Comparison sorts can be viewed abstractly in terms of ***decision trees*** （判定树）
- A decision tree is a full binary tree （严格二叉树） that represents the comparisons between elements;
- Control, data movement, and all other aspects of the algorithm are ignored 。
- 基于比较的排序算法，其元素比较的过程可以用一棵判定树来表示。

The decision tree

The decision tree for insertion sort operating on three elements：



The decision tree

- The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree;
- A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is a lower bound on the running time of any comparison sort algorithm;
- 对于排序问题开言，由于输入的n个元素，可以有$n!$种不同的排序结果，因此表示n个元素的排序过程的判定树至少有$n!$个叶结点，分别表示$n!$种可能的排序结果。

The decision tree

- ***Theorem 8.1：***

  Any comparison sort algorithm requires $\Omega(n\log n)$ comparisons in the worst case。

- ***Proof***
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements;
  - Each of the $n!$ permutations of the input appears as some leaf, so $n! \le l$
  - A binary tree of height h has no more than 2^h leaves, so

$$\boldsymbol{n! \le l \le 2^h},$$

$$\boldsymbol{h \ge \log(n!) = \Omega(n\log n)}$$

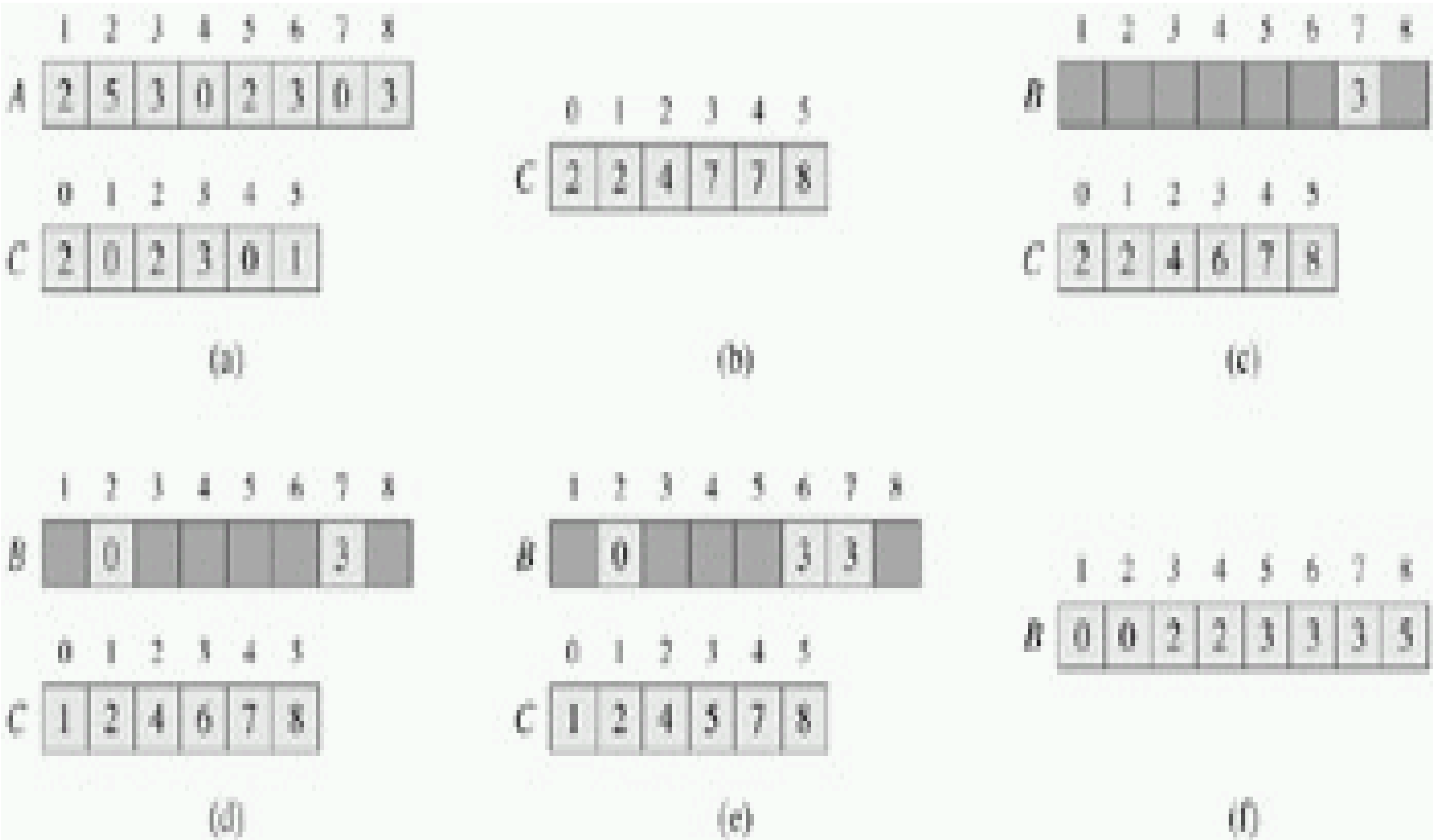
8.2 Counting Sort

- ***Counting sort*** assumes that each of the n input elements is an integer in the range 0 to k , for some integer k 。
- The basic idea is to determine, for each input element x, the number of elements less than or equal to x;
- This information can be used to place element x directly into its position in the output array;
- In the counting sort, the input is an array $A[1 \cdot\cdot n]$ and two other arrays are required: the array $B[1 \cdot\cdot n]$ holds the sorted output, and the array $C[0 \cdot\cdot k]$ provides temporary working storage

Counting Sort 算法

```
COUNTING-SORT (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[t]表示等于 t 的元素个数 //
For i←1 to k
    do C[i]←C[i]+C[i-1]    // 现在C[t]表示小于等于 t 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
```

Example: Counting Sort



Counting Sort 性能分析

- Time complexity
  - The **for** loop of lines 1–2 takes time $\Theta(k)$
  - The **for** loop of lines 3–4 takes time $\Theta(n)$
  - The **for** loop of lines 6–7 takes time $\Theta(k)$
  - The **for** loop of lines 9–11 takes time $\Theta(n)$
  - Thus, the overall time is $\Theta(k+n)$
  - Counting sort is not a comparison sort, so it beats the lower bound of $\Omega(n \log n)$
  - Counting sort is ***stable***: numbers with the same value appear in the output array in the same order as they do in the input array.

Homework 8.2

- Page 100: 8.2-1, 8.2-4

8.3 Radix sort

- ***Radix sort*** : The algorithm used by the card-sorting machines. It sorts *n* cards on a *d*-digit number;
- Radix sort sorts on the *least significant* digit first and are then combined into a single deck, then the entire deck is sorted again on the second-least significant digit and recombined in a like manner;
- The process continues until the cards have been sorted on all *d* digits .

Example: Radix sort

- Following figure shows how radix sort operates on a "deck" of seven 3-digit numbers.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort 算法

- The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

- **RADIX-SORT**(A, d)

  1 **for** $i \leftarrow 1$ **to** d

  2 **do** use a stable sort to sort array A on digit i

Radix sort 算法性能分析

- **Lemma 8.3:**

  Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d\,(n + k))$ time。

- ***Proof***
  - Each pass over n d-digit numbers takes time $\Theta(n + k)$
  - There are d passes, so the total time for radix sort is $\Theta(d\,(n + k))$。

Radix sort 算法性能分析(续)

- **Lemma 8.4:**

  Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time.

- ***Proof***
  - For a value $r \le b$, each key was viewed as having $d = \lceil b/r \rceil$ digits of r bits each;
  - Each digit is an integer in the range 0 to 2^r-1 , so that we can use counting sort with $k = 2^r - 1$;
  - Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$, and there are d passes.
  - So the total running time is $\Theta(d\,(n + 2^r)) = \Theta((b/r)(n+2^r))$.

Radix sort 算法性能分析(续)

- For given values of n and b, how to choose the value of r, with $r \le b$, that minimizes the expression $(b/r)(n+2^r)$.
  - If $b < \lfloor \log n \rfloor$, choosing $r = b$ yields a running time of
    $$\Theta((b/b)(n+2^b)) = \Theta(n)$$
  - If $b \ge \lfloor \log n \rfloor$, then choosing $r = \lfloor \log n \rfloor$, so the running time is $\Theta(bn/\log n)$.
  - Increasing r above $\lfloor \log n \rfloor$ yields a running time of $\Omega(bn/\log n)$
  - If decreasing r below $\lfloor \log n \rfloor$, then the b/r term increases and the $(n+2^r)$ term remains at $\Theta(n)$.

排序算法的选择

- Is radix sort preferable to a comparison-based sorting algorithm, such as quick-sort?
  - If $b = O(\log n)$ and $r \approx \log n$, then radix sort's running time is $O(n)$, which is better than quicksort's average-case running time of $\Theta(n \log n)$.
  - Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take longer time.

排序算法的选择（续）

- Which sorting algorithm is preferable depends on the characteristics of the implementations, of the underlying machine and of the input data.

- Radix sort that uses counting sort does not sort in place, which many comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort may be preferable.

Homework 8.3

- Page 102: 8.3-2, 8.3-4

8.4 Bucket sort

- ***Bucket sort*** runs in linear time when the input is drawn from a uniform distribution。
- It assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1)。
- The idea of bucket sort is to divide the interval [0, 1) into *n* equal-sized subintervals, or ***buckets***, and then distribute the *n* input numbers into the buckets。
- Sort the numbers in each bucket and then go through the buckets in order, listing the elements in each 。

Bucket sort 算法

- **BUCKET-SORT**(A)

$n \leftarrow Length[A]$

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n\,A[i]\rfloor]$

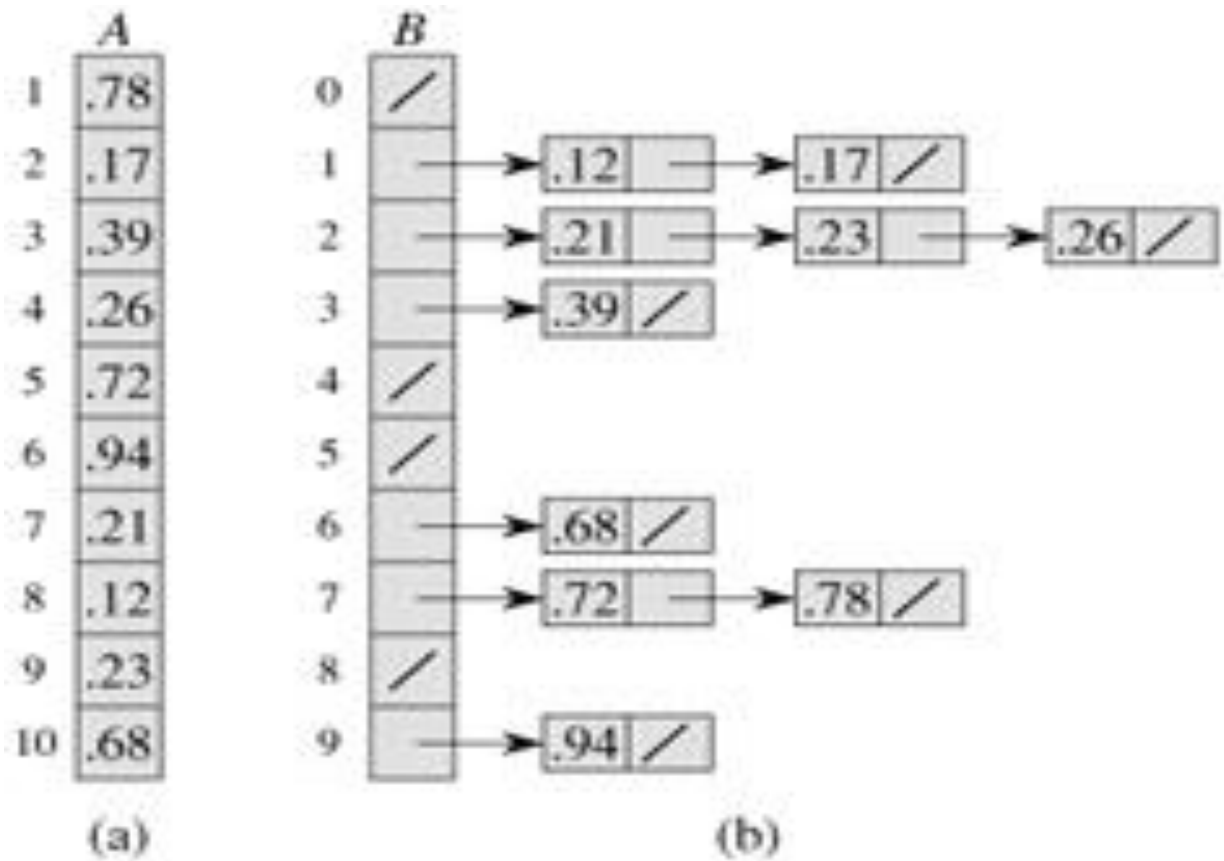
for $i \leftarrow 0$ **to** $n - 1$

  do sort list $B[i]$ with insertion sort

Concatenate the lists $B[0], B[1], \ldots, B[n - 1]$ together in order

Eexample: Bucket sort

- The figure shows the operation of bucket sort on an input array of 10 numbers



Bucket sort 算法分析

- Running time
  - 设 n_i 为一个随机变量表示 $B[i]$桶中的元素个数，由于插入排序时间复杂度为$O(n^2)$，则桶排序的运行时间为：

$$T(n) \le Cn + \sum_{i=0}^{n-1} O(n_i^2)$$

  - 上式两边取期望值可得平均时间复杂度为：

$$E(T(n)) = E(Cn + \sum_{i=0}^{n-1} O(n_i^2)) = Cn + \sum_{i=0}^{n-1} O(E(n_i^2))$$

Bucket sort 算法分析(续)

- 下面证明: $E(n_i^2) = 2 - 1/n$

  对下标 $i = 0, 1, \ldots, n - 1$ and $j = 1, 2, \ldots, n$， 定义随机变量:

  $X_{ij} = \{1 \mid A[j] \in B(i)\}$ 则: $n_i = \sum_{j=1}^{n} X_{ij}$

$$E(n_i^2) = E((\sum_{j=1}^{n} X_{ij})^2) = E(\sum_{j}^{n}\sum_{k}^{n} X_{ij} X_{ik})$$

$$= E(\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n} \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij} X_{ik}) = \sum_{j=1}^{n} E(X_{ij}^2) + \sum_{1\le j\le n} \sum_{\substack{1\le k\le n \\ k\ne j}} E(X_{ij} X_{ik})$$

Bucket sort 算法分析 (续)

由于随机变量 X_{ij}为1的概率为1/n，为 0 的概率为 1-1/n

$$E(X_{ij}^2) = 1 \cdot 1/n + 0 \cdot (1 - 1/n) = 1/n$$

当 $k \neq j$ 时, X_{ij} 和X_{ik} 是独立的随机变量，因此

$$E(X_{ij} X_{ik}) = E(X_{ij}) E(X_{ik}) = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

$$E(n_i^2) = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \neq j}} \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n}$$

Using this expected value, we conclude that the expected time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$

Homework 8.4

- Page 104: 8.4-1, 8.4-4.

9.Medians and Order Statistics

- The i th ***order statistic*** of a set of n elements is the i th smallest element
- A ***median*** is the "halfway point" of the set. Medians occur at $i = \lfloor(n + 1)/2\rfloor$ (the ***lower median***) and $i = \lceil(n + 1)/2\rceil$ (the ***upper median***). In this course they refer to the lower median.
- The ***selection problem*** is specified as follows:
  - **Input:** a set A of n (distinct) numbers and a number i, with $1 \le i \le n$
  - **Output:** The element $x \in A$ that is larger than exactly i - 1 other elements of A .

9.1 Minimum and maximum

- To determine the minimum of a set of n elements, a lower bound of comparisons is $n - 1$.
- The following procedure selects the minimum from the array A , where $Length[A] = n$.

MINIMUM(A)

1 $min \leftarrow A[1]$
2 **for** $i \leftarrow 2$ **to** $Length[A]$
3   **do if** $min > A[i]$
4   **then** $min \leftarrow A[i]$
5 **return** min

同时找最大、最小值

- 在有些应用中，需要将 n 个元素中的最大和最小都找到，为了解决同时找出最大、最小值的问题;
- 可以先求最大值，再求最小值，这样需要 2n-3次比较;
- 同时找最大、最小值问题需要的元素间的比较次数至少为:

$$\left\lceil \frac{3n}{2} \right\rceil - 2$$

- 我们可以将所有元素两两比较，从大的元素中找出最大值，从小的元素中找出最小值，这样所需的比较次数恰好与下限相同。

同时找最大、最小值算法

MAX-MIN (A)

1. If $A[1]>A[2]$ then Min$\leftarrow A[2]$, Max$\leftarrow A[1]$;
2. else Min$\leftarrow A[1]$, Max$\leftarrow A[2]$;
3. $m\leftarrow n/2$ // 整除 //
4. For $i\leftarrow 2$ to m
5. do if $A[2i-1]>A[2i]$
6. then if $A[2i]<$Min then Min $\leftarrow A[2i]$
7. if $A[2i-1]>$Max then Max $\leftarrow A[2i-1]$
8. else if $A[2i-1]<$Min then Min $\leftarrow A[2i-1]$
9. if $A[2i]>$Max then Max $\leftarrow A[2i]$
10. if $n\neq 2m$ then if $A[n]<$Min then Min $\leftarrow A[n]$
11. if $A[n]>$Max then Max $\leftarrow A[n]$
12. Return (Min, Max)

找最大、最小值算法分析

1. 当 n 是偶数时，有整数个数对，先将第一对元素比较一次得到最大值和最小值，然后将剩余的元素两两比较。总共进行的比较次数是 $3n/2-2$。

2. 当 n 是奇数时，前n-1个元素中找出最大、最小值共需 $3(n-1)/2-2$次比较，再与第n个元素各比较一次，总共进行的比较次数是 $3(n-1)/2$。

3. 总之，在任何一种情况下，总的比较次数可以表示为 $\lceil 3n/2 \rceil-2$ 。

Homework 9.1

- Page 109: 9.1-1.

9.2 Selection-Expected Linear Time

- A divide-and-conquer algorithm for the selection problem: RANDOMIZED-SELECT
- The idea is to partition the input array recursively (as in quicksort)
- The difference is that quicksort recursively processes both sides of the partition, but RANDOMIZED-SELECT only works on one side of the partition
- Quicksort has an expected running time of $\Theta(n \log n)$, but the expected time of RANDOMIZED-SELECT is $\Theta(n)$

简单选择算法

RANDOMIZED-SELECT(A,p,r,i)

1. If $p=r$ then return $A[p]$
2. $q \leftarrow$RANDOMIZED-PARTITION(A,p,r)
3. $k \leftarrow q-p+1$
4. If $i=k$ then return $A[q]$
5. else if i<k
6. then return RANDOMIZED-SELECT($A,p,q-1,i$)
7. else return RANDOMIZED-SELECT($A,q+1,r,i-k$)

简单选择算法性能

- The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, because we probably always partition around the largest remaining element, and partitioning takes $\Theta(n)$ time
- The expected running time for RANDOMIZED-SELECT is O(n)；

简单选择算法性能分析

- The time required by RANDOMIZED-SELECT on an input array $A[p \mathinner{..} r]$ of n elements is denoted by $T(n)$。
- We define indicator random variables X_k where

X_k = I {the subarray $A[p \mathinner{..} q]$ has exactly k elements；

- So we have: $E[X_k] = 1/n$
- X_k *has* the value 1 for exactly one value of k, and it is 0 for all other k
- When $X_k = 1$, the two subarrays on which we might recurse have sizes $k - 1$ and $n - k$。

简单选择算法性能分析（续）

$$
\begin{aligned}
T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\
&= \sum_{k=1}^{n} (X_k \cdot T(\max(k-1, n-k)) + O(n)) .
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{E}[T(n)] \\
&\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(by linearity of expectation)} \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (C.23))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (9.1))} .
\end{aligned}
$$

简单选择算法性能分析（续）

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil , \\ n-k & \text{if } k \le \lceil n/2 \rceil . \end{cases}$$

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) .$$

Assume that $T(n) \le cn$ for some constant c that satisfies the initial conditions of the recurrence。 Pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded from above by an for all $n > 0$

Using the inductive hypothesis above, we have：

$$
\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2\rfloor-1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\lfloor n/2\rfloor}{2}\right) + an \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4-3n/2+2}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) .
\end{aligned}
$$

简单选择算法性能分析（续）

- For sufficiently large n, we have $cn/4 - c/2 - an \geq 0$

  $$n(c/4 - a) \geq c/2$$

- As long as we choose the constant c so that $c/4 - a > 0$, i.e., $c > 4a$, we can divide both sides by $c/4 - a$, giving

  $$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

- If we assume that $T(n) = O(1)$ for $n < 2c/(c - 4a)$, we have $T(n) = O(n)$
- So any order statistic, and in particular the median, can be determined on average in linear time

Homework 9.2

- Page 111: 9.2-4.

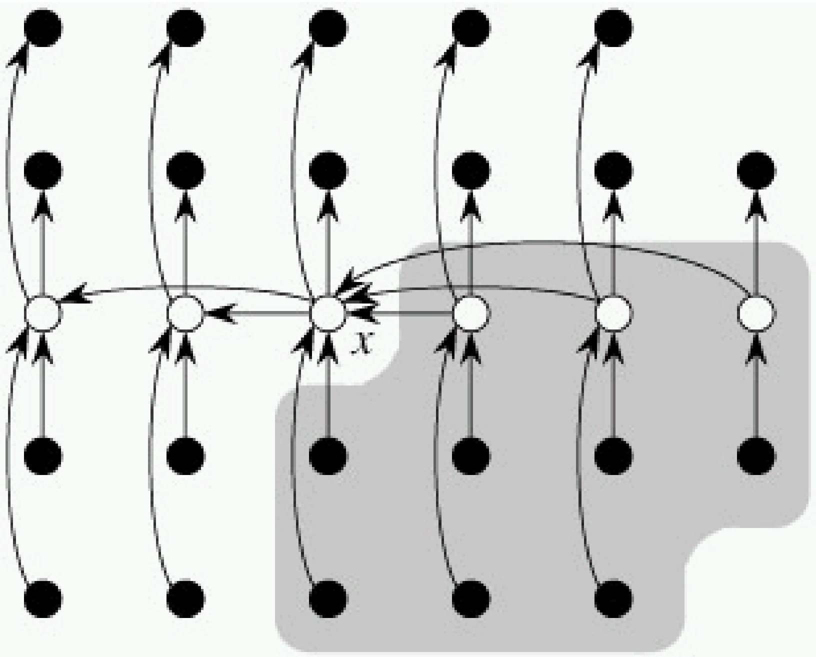
9.3 线性时间选择算法

- 在本节中介绍一种最坏情况时间为$O(n)$的选择算法;
- 算法主要针对上节算法中划分不均匀的情况进行改进，使得每次划分时均比较均衡。
- The SELECT algorithm determines the *i*th smallest of an input array of $n > 1$ elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the *i*th smallest。)

选择算法思想

- 将输入数组的n 个元素分成每组5个元素，共$\lfloor n/5 \rfloor$ 组，最后剩下一组少于5个元素；
- 在有5个元素的每一组中采用插入排序求出每组的中值，然后从排好序的组元素列表中将中值摘出；
- 递归调用 SELECT找出第2步所求的$\lfloor n/5 \rfloor$个中值的中值x（若有偶数个中值，x 是指较低的中值）；
- 使用PARTITION的调整版本，根据中值的中值x来划分输入数组。使得划分后低端元素数少于k，x 是第 k 小的元素，高端有 $n-k$ 个元素；
- 如果 $i=k$，就返回 x；否则，如果 $i<k$，递归调用SELECT在低端部分寻找第 i 小的元素；如果 $i>k$，在高端部分寻找第 $(i-k)$的元素；

- Figure 9.1 shows analysis of the algorithm SELECT
- The *n* elements are represented by small circles
- The medians of the groups are whitened, and the median-of-medians *x* is labeled. Arrows are drawn from larger elements to smaller.



选择算法性能分析

- To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element *x*.
- At least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than *x*, except for the one group that has fewer than 5 elements if 5 does not divide *n* exactly, and the one group containing *x* itself.
- So the number of elements greater than *x* is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

- So in the worst case, SELECT is called recursively on at most $7n/10 + 6$ elements in step 5.

选择算法性能分析(续)

- Steps 1, 2, and 4 take $O(n)$ time.
- Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10+6)$, assuming that T is monotonically increasing
- Assume that any input of 140 or fewer elements requires $O(1)$ time.
- So we have the recurrence

$$T(n) \le \begin{cases} \Theta(1) & \text{if } n \le 140\,, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{if } n > 140\,. \end{cases}$$

选择算法性能分析（续）

- Assuming that $T(n) \leq cn$ for some suitably large constant c and all $n \leq 140$
- Pick a constant a such that the function described by the $O(n)$ term above is bounded above by an for all $n > 0$
- So we have

$$
\begin{aligned}
T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\
&\leq cn/5 + c + 7cn/10 + 6c + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an)
\end{aligned}
$$

选择算法性能分析（续）

- Thus T(n) is at most *cn* if $-cn/10 + 7c + an \le 0$. (9.2)

  $c \ge 10a(n/(n - 70))$ when $n > 70$.

- Because $n \ge 140$ $n/(n - 70) \le 2$

- So choosing $c \ge 20a$ will satisfy inequality (9.2)
- The worst-case running time of SELECT is therefore linear
- The algorithm is still correct if each group has *r* elements where *r* is odd and is not less than 5。

选择算法性能分析（续）

- Sorting requires $\Omega(n \log n)$ time in the comparison model, even on average, and the linear-time sorting algorithms in Chapter 8 make assumptions about the input。
- But the linear-time selection algorithms in this chapter do not require any assumptions about the input
- The running time is linear because these algorithms do not sort

Homework 9.3

- Page 113: 9.3-3， 9.3-6.