算法设计与分析
Algorithms Design & Analysis

第四讲：线性时间的排序法

线性时间的排序法(Sorting in Linear Time)

- Lower bound for comparison-based sorting(基于比较的排序方法的开销下限)
- Counting sort(记数排序)
- Radix sort(基数排序法)
- Bucket sort(桶排序)

排序法回顾(Sorting So Far)

- Insertion sort(插入排序):
  - Easy to code(易于实现)
  - Fast on small inputs (less than ~50 elements)(适合小规模输入)
  - Fast on nearly-sorted inputs(对近似排序好的输入效果好)
  - $O(n^2)$ worst case(最坏情况)

排序法回顾(Sorting So Far)

- Merge sort(合并排序):
  - Divide-and-conquer(分治):
    - Split array in half(分割)
    - Recursively sort subarrays(递归处理)
    - Linear-time merge step(线性时间合并)
  - O(n lg n) worst case(最坏情况)

How Fast Can We Sort?(我们能否排得更快?)

- Insertion sort $\Theta(n^2)$
- Merge sort $\Theta(n\lg n)$
- What is common to all these algorithms? (上述算法共同点?)
  - These algorithms sort by making comparisons between the input elements (基于元素之间的两两比较)

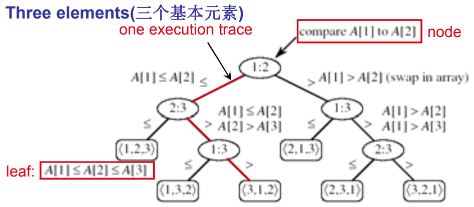
比较排序(Comparison Sorting)

- Comparison sorts use comparisons between elements to gain information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ (通过比较获得输入元素之间的相对顺序，比较的操作有：)
- $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$

- To sort n elements, comparison sorts must make $\Omega(n\lg n)$ comparisons in the worst case (在最坏情况,基于比较思想排序的必须要做$\Omega(n\lg n)$次比较)

判定树模型(Decision Tree Model)

- **Represents the comparisons made by a sorting algorithm on an input of a given size: models all possible execution traces(表示基于比较排序算法的所有可能比较情况)**
- **Control, data movement, other operations are ignored(忽略了其他操作,比如控制,数据转移等操作)**
- **Count only the comparisons(只考虑比较)**
- **Three elements(三个基本元素)**



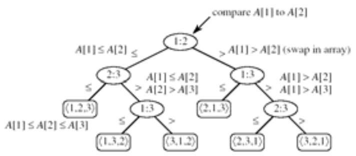
7

Decision Tree Model

- All permutations on n elements must appear as one of the leaves in the decision tree n! permutations

n!种排列对应同数目的叶节点

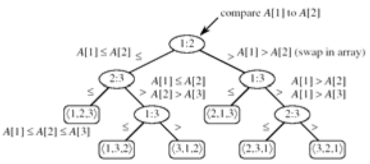
- Worst-case number of comparisons （最坏情况）
  - the length of the longest path from the root to a leaf （从根到叶最长的路径）



8

判定树模型(Decision Tree Model)

- Goal: finding a lower bound on the running time on any comparison sort algorithm(目的是为了确定基于比较的排序算法的下限)
  - find a lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf (判定树中，所有元素的排列情况作为叶节点出现，目的是确定判定树的高度)



9

引理（Lemma）

- Any binary tree of height h has at most 2^h leaves

（任何树高为h的二分树至多有2^h 个叶节点）

Proof: induction on h （证明，归纳法）

Basis: h = 0 ⇒ tree has one node, which is a leaf （只有叶节点）

$2^h = 1$

Inductive step: assume true for h-1 （假设对于h-1成立）

- Extend the height of the tree with one more level
- Each leaf becomes parent to two new leaves

（当高度增加1时，即h-1 →h，原来的叶节点都成为2个新叶节点的父节点）

No. of leaves at level h = 2 ∗ (no. of leaves at level h-1)

$= 2 * 2^{h-1}$

$= 2^h$

10

比较排序下限（Lower Bound for Comparison Sorts）

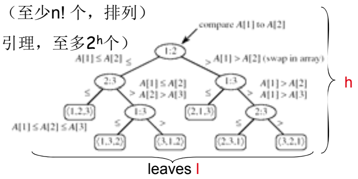
Theorem: Any comparison sort algorithm requires Ω(nlgn) comparisons in the worst case. （任何基于比较的排序算法在最坏情况下至少需要做Ω(nlgn)次比较）

Proof: How many leaves does the tree have? （叶节点的数目）

- At least n! (each of the n! permutations if the input appears as some leaf) ⇒ n! ≤ l （至少n! 个，排列）
- At most 2^h leaves （引理，至多2^h个）

⇒ $n! \le l \le 2^h$

⇒ $h \ge \lg(n!) = \Omega(n\lg n)$



We can beat the Ω(nlgn) running time if we use other operations than comparisons!（放弃比较的排序思想，可以获得更优开销，突破Ω(nlgn)，但是是有条件的）

11

计数排序（Counting Sort）

- Assumption:
  - The elements to be sorted are integers in the range 0 to k（将要排序的元素是0到k之间的整数）
- Idea:（算法思想）
  - Determine for each input element x, the number of elements smaller than x （对于每一个元素x，判断小于或等于元素x的元素的个数）
  - Place element x into its correct position in the output array（然后根据计数的情况将元素放入到相应的位置）

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

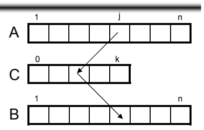
	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

12

计数排序（COUNTING-SORT）

Alg.: COUNTING-SORT(A, B, n, k)

for i ← 0 **to** k
　　do *C*[i] ← 0
for j ← 1 **to** n
　　do *C*[*A*[j]] ← *C*[*A*[j]] + 1
　▷*C*[i] contains the number of elements equal to i （ *C*[i]包含等于i的元素的个数）
for i ← 1 **to** k
　　do *C*[i] ← *C*[i] + *C*[i -1]
　▷*C*[i] contains the number of elements ≤ i （ *C*[i]包含小于或等于i的元素的个数）
for j ← n **downto** 1
　　do B[*C*[*A*[j]]] ← *A*[j]
　　　C[*A*[j]] ← *C*[*A*[j]] - 1



Example

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	0	2	3	0	1

C[A[j]] ← C[A[j]] + 1

C

0	1	2	3	4	5
2	2	4	7	7	8

C[i] ← C[i] + C[i -1]

B

1	2	3	4	5	6	7	8
						3	

C

0	1	2	3	4	5
2	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0					3	

C

0	1	2	3	4	5
1	2	4	6	7	8

B[C[A[j]]] ← A[j] ; C[A[j]] ← C[A[j]] – 1 ; j : n→1

B

1	2	3	4	5	6	7	8
	0				3	3	

C

0	1	2	3	4	5
1	2	4	5	7	8

B

1	2	3	4	5	6	7	8
	0		2		3	3	

C

0	1	2	3	4	5
1	2	3	5	7	8

Example (cont.)

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

B

1	2	3	4	5	6	7	8
0	0		2		3	3	

C

0	1	2	3	4	5
0	2	3	5	7	8

B

1	2	3	4	5	6	7	8
0	0		2	3	3	3	5

C

0	1	2	3	4	5
0	2	3	4	7	7

B

1	2	3	4	5	6	7	8
0	0		2	3	3	3	

C

0	1	2	3	4	5
0	2	3	4	7	8

B

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

算法分析（Analysis of Counting Sort）

Alg.: COUNTING-SORT(A, B, n, k)

for i ← 0 **to** k
　　do *C*[i] ← 0 $\Big\}\ \Theta(k)$
for j ← 1 **to** n
　　do *C*[*A*[j]] ← *C*[*A*[j]] + 1 $\Big\}\ \Theta(n)$
　▷*C*[i] contains the number of elements equal to i
for i ← 1 **to** k
　　do *C*[i] ← *C*[i] + *C*[i -1] $\Big\}\ \Theta(k)$
　▷*C*[i] contains the number of elements ≤ i
for j ← n **downto** 1
　　do B[*C*[*A*[j]]] ← *A*[j]
　　　C[*A*[j]] ← *C*[*A*[j]] - 1 $\Big\}\ \Theta(n)$

Overall time: Θ(n + k)

分析（Analysis of Counting Sort）

- Overall time: Θ(n + k)
- In practice we use COUNTING sort when k = *O*(n)

　⇒ running time is Θ(n) （小范围整数的排序）
- Counting sort is **stable**（稳定的）
  – Numbers with the same value appear in the same order in the output array（相等的输入元素的位置先后顺序在排列后输出中的先后顺序不变）

Stable的定义

- A sorting algorithms is ***stable*** if for any two indices *i* and *j* with $i < j$ and $a_i = a_j$, element a_i precedes element a_j in the output sequence.(稳定的定义)

Input

2_1	7_1	4_1	4_2	2_2	5_1	2_3	6_1

Output

2_1	2_2	2_3	4_1	4_2	5_1	6_1	7_1

基数排序(Radix Sort)

- Extending the Range of Sortable Integers
- (扩大了可排序的整数范围)

基数排序思想(The Idea of Radix Sort)

- Any number can be represented in radix-k notation.(数的基表示)
- **Examples:**
  - $16536 = 1 \cdot 10^4 + 6 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0$
  - $\langle 101101 \rangle_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 45$

基数排序（RADIX-SORT）

Alg.: RADIX-SORT(*A*, d)

for i ← 1 to d

do use a stable sort to sort array *A* on digit i （用稳定的排序列方法对第i位排序）

- **1 is the lowest order digit, d is the highest-order digit**（从最后，也就是最不重要的那一位开始）

sorted

326	→	690	→	704	→	326
453		751		608		435
608		453		326		453
835		704		835		608
751		835		435		690
435		435		751		704
704		326		453		751
690		608		690		835

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $\Theta(d(n+k))$ （n个d位的数，以k为基）
  - One pass of sorting per digit takes $\Theta(n+k)$ assuming that we use counting sort （如果用计数排序，排一位需要$\Theta(n+k)$ ）
  - There are d passes (for each digit) （排d位需要$\Theta(d(n+k))$ ）

基数排序的正确性（Correctness of Radix sort）

- We use induction on number of passes through each digit （归纳法）
- **Basis:** If d = 1, there's only one digit, trivial （1位，显然）
- **Inductive step:** assume digits 1, 2, . . . , d-1 are sorted （1, 2, . . . , d-1已排序）
  - Now sort on the d-th digit （三种情况）
  - If $a_d < b_d$, sort will put a before b: correct $a < b$ regardless of the low-order digits
  - If $a_d > b_d$, sort will put a after b: correct $a > b$ regardless of the low-order digits
  - If $a_d = b_d$, sort will leave a and b in the same order and a and b are already sorted on the low-order d-1 digits（等于，低位已经排好）WHY?

704	→	326
608		435
326		453
835		608
435		690
751		704
453		751
690		835

桶排序（Bucket Sort）

- **Assumption:**
  - **the input is generated by a random process that distributes elements uniformly over [0, 1)** （输入是在[0, 1)上一致均匀分布）
- **Idea:**
  - **Divide [0, 1) into n equal-sized buckets** （将[0, 1) n等分，称为空桶）
  - **Distribute the n input values into the buckets** （将n个输入分派到这些空桶之中）
  - **Sort each bucket** （每个桶内排序）
  - **Go through the buckets in order, listing elements in each one** （按顺序遍历每个桶中的元素）
- **Input:** $A[1 \mathinner{..} n]$, where $0 \le A[i] < 1$ for all i
- **Output:** elements a_i sorted
- **Auxiliary array:** $B[0 \mathinner{..} n - 1]$ of linked lists, each list initially empty （桶，初始为空）

桶排序算法（BUCKET-SORT）

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n

  do insert A[i] into list B[⌊nA[i]⌋] （注意下标）

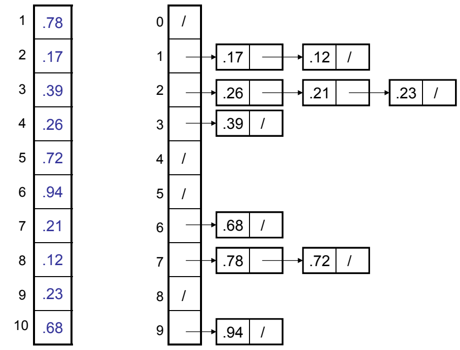
for i ← 0 **to** n - 1

  do sort list B[i] with insertion sort （桶内插入排序）

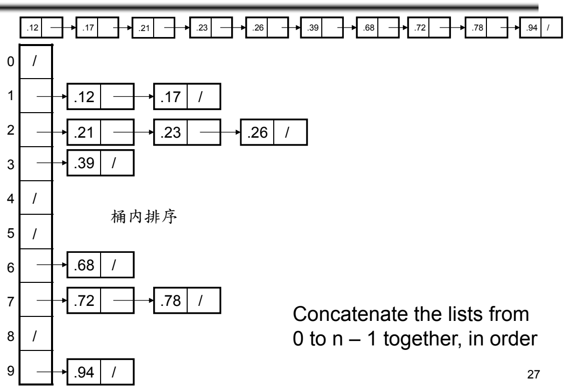
concatenate lists B[0], B[1], . . . , B[n -1] together in order

return the concatenated lists

Example - Bucket Sort



Example - Bucket Sort



Concatenate the lists from 0 to n – 1 together, in order

桶排序的正确性(Correctness of Bucket Sort)

- Consider two elements A[i], A[j] (考虑两元素A[i], A[j])
- Assume without loss of generality that A[i] ≤ A[j] (不失一般性可假设A[i] ≤ A[j])
- Then ⌊nA[i]⌋ ≤ ⌊nA[j]⌋ (有⌊nA[i]⌋ ≤ ⌊nA[j]⌋)
  - A[i] belongs to the same group as A[j] or to a group with a lower index than that of A[j] (A[i]要么与A[j]在一个桶中,要么在比A[j]更低的桶中)
- If A[i], A[j] belong to the same bucket: (同一桶中,桶内排序保障A[i] ≤ A[j] 顺序正常)
  - insertion sort puts them in the proper order
- If A[i], A[j] are put in different buckets: (不同桶中,连接过程保障正确)
  - concatenation of the lists puts them in the proper order

平均情况(Average-Case Running Time of Bucket Sort)

- **Lemma:** ***Given that the input sequence is drawn uniformly at random from [0,1), the average-case running time of Bucket Sort is O(n).(输入满足一致性随机分布,桶排序的平均运行开销为O(n).证明可阅读教材)***

结论(Conclusion)

- Any comparison sort will take at least nlgn to sort an array of n numbers(基于比较: nlgn)
- We can achieve a better running time for sorting if we can make certain assumptions on the input data: (对输入作更细致的考虑)
  - Counting sort: each of the n input elements is an integer in the range 0 to k (计数排序:0到k的整数)
  - Radix sort: the elements in the input are integers represented with d digits (基数排序:d整数位)
  - Bucket sort: the numbers in the input are uniformly distributed over the interval [0, 1) (桶排序:[0,1)一致性分布)