Design and Analysis of Algorithms

Sorting in Linear Time

Lecture 15-16

Instructor: Dr. G P Gupta

Comparison-based Sorting

- **Comparison sort**
  - » Only comparison of pairs of elements may be used to gain order information about a sequence.
  - » Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- **All our sorts have been comparison sorts**
- The best worst-case complexity so far is $\Theta(n \lg n)$ (merge sort and heapsort).
- We prove a lower bound of $n \lg n$, (or $\Omega(n \lg n)$) for any comparison sort, implying that merge sort and heapsort are optimal.

How fast can we sort?

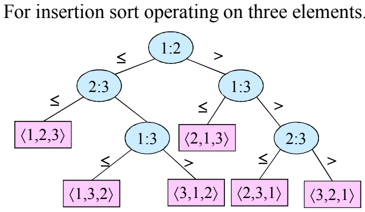
- All the sorting algorithms we have seen so far are ***comparison sorts*: only use comparisons to determine the relative order of elements.**
  - » ***E.g.*, insertion sort, merge sort, quick sort, heap sort.**
- The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.
- ***Is O(n lg n) the best we can do?***
- ***Decision trees can help us answer this question.***

Decision Tree

- Binary-tree abstraction for any comparison sort.
- Represents comparisons made by
  - » a specific sorting algorithm
  - » on inputs of a given size.
- Abstracts away everything else – control and data movement – counting only comparisons.
- Each internal node is annotated by $i{:}j$, which are indices of array elements from their original positions.
- Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ of orders that the algorithm determines.

Decision Tree – Example

For insertion sort operating on three elements.



Contains 3! = 6 leaves.

Decision Tree cont…

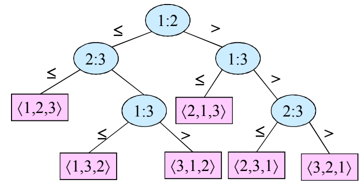
- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \le a_j$ is made.
  - » If $a_i \le a_j$, follow left subtree, else follow right subtree.
  - » View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - » Hence, each of the $n!$ permutations must appear at one or more of the leaves of the decision tree.

A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is
  - » Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - Which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - » A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

Optimal sorting for three elements

Any sort of six elements has 5 internal nodes.



There must be a wost-case path of length ≥ 3.

A Lower Bound for Worst Case

***Theorem* 8.1:**
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

- From previous discussion, suffices to determine the height of a decision tree.
- ***h*** – height, ***l*** – no. of reachable leaves in a decision tree.
- In a decision tree for n elements, $l \geq n!$. **Why?**
- In a binary tree of height h, no. of leaves $l \leq 2^h$. **Prove it.**
- Hence, $\mathbf{n! \leq l \leq 2^h}$.

Proof – Contd.

- $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Taking logarithms, $h \geq \lg(n!)$.
- $n! > (n/e)^n$. (Stirling's approximation, Eq. 3.19.)
- Hence, $h \geq \lg(n!)$

$$\begin{aligned} &\geq \lg(n/e)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Counting Sort

- No comparisons between elements
- Sorting in linear time
- Depends on a **key *assumption***:
  - » numbers to be sorted are integers in $\{0, 1, 2, \ldots, k\}$.
- **Input:** $A[1..n]$, where $A[j] \in \{0, 1, 2, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.
- **Output:** $B[1..n]$ sorted. B is assumed to be already allocated and is given as a parameter.
- **Auxiliary Storage:** $C[0..k]$
- Runs in linear time if $\boldsymbol{k = O(n)}$.

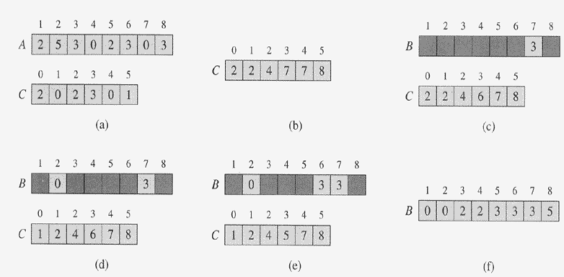


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort: Algorithm

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ⊳ $C[i] = |\{\text{key} \le i\}|$
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Counting-sort example

Loop 1

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

Counting-sort example

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$

Counting-sort example

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

B:					

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$

Counting-sort example

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$

Counting-sort example

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	1	2

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$

Counting-sort example

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ⊳ $C[i] = |\{\text{key} = i\}|$

Counting-sort example

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ⊳ $C[i] = |\{\text{key} \le i\}|$

Counting-sort example

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ⊳ $C[i] = |\{\text{key} \le i\}|$

Counting-sort example

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ⊳ $C[i] = |\{\text{key} \le i\}|$

Counting-sort example

Loop 4

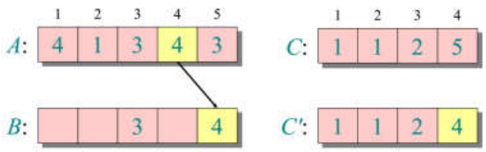
	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	3	5
C':	1	1	2	5

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

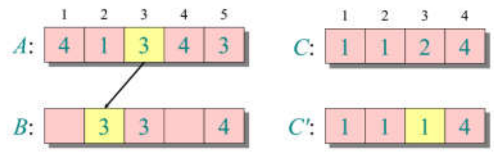
Loop 4



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

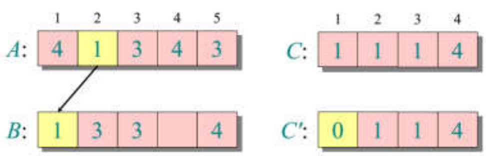
Loop 4



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

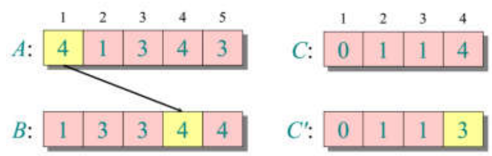
Loop 4



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

Loop 4



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Analysis of Counting-sort

$\Theta(k)$ { **for** $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$ }

$\Theta(n)$ { **for** $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ }

$\Theta(k)$ { **for** $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ }

$\Theta(n)$ { **for** $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$ }

$\Theta(n + k)$

Exercises

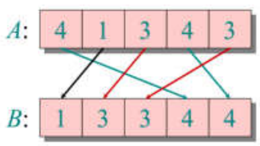
- illustrate the operation of COUNTING-SORT on the array A = < 6; 0; 2; 0; 1; 3; 4; 6; 1; 3; 2 >
- Prove that COUNTING-SORT is stable.
- Suppose that we were to rewrite the for loop header ***in line 10*** of the COUNTINGSORT as

  10 for j =1 to A:length

  Show that the algorithm still works properly. **Is the modified algorithm stable?**

Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.



Counting-sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Counting-Sort (*A*, *B*, *k*)

```
CountingSort(A, B, k)
1. for i ← 1 to k                      } O(k)
2.      do C[i] ← 0
3. for j ← 1 to length[A]              } O(n)
4.      do C[A[j]] ← C[A[j]] + 1
5. for i ← 2 to k                      } O(k)
6.      do C[i] ← C[i] + C[i –1]
7. for j ← length[A] downto 1          } O(n)
8.      do B[C[A[ j ]]] ← A[j]
9.          C[A[j]] ← C[A[j]]–1
```

Algorithm Analysis

- The *overall time is $O(n+k)$*. When we have $k=O(n)$, the worst case is $O(n)$.
  - » for-loop of lines 1-2 takes time $O(k)$
  - » for-loop of lines 3-4 takes time $O(n)$
  - » for-loop of lines 5-6 takes time $O(k)$
  - » for-loop of lines 7-9 takes time $O(n)$
- ***Stable**, but <u>not</u> in place.*
- ***No comparisons made:*** it uses actual values of the elements to index into an array.

Radix Sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

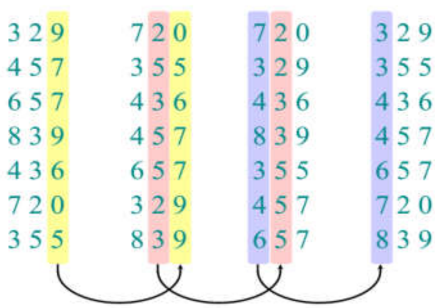
Radix Sort

- It was ***used by the card-sorting machines***.
- Card sorters worked on one column at a time.
- It is the algorithm for using the machine that extends the technique to ***multi-column sorting.***
- The human operator was part of the algorithm!
- ***Key idea:***
  - » sort on the "***least significant digit***" first and on the remaining digits in sequential order.
  - » *The sorting method used to sort each digit must be* "stable".
    - **If we start with the "most significant digit", we'll need extra storage.**

An Example

Input		After sorting on LSD		After sorting on middle digit		After sorting on MSD
392		631		928		356
356		392		631		392
446		532		532		446
928	⇒	495	⇒	446	⇒	495
631		356		356		532
532		446		392		631
495		928		495		928
		↑		↑		↑

Operation of radix sort



Radix-Sort(A, d)

RadixSort(A, d)
1. for $i \leftarrow 1$ to d
2. do *use a stable sort to sort array A on digit i*

Correctness of Radix Sort :

- By induction on the number of digits sorted.
- Assume that radix sort works for $d - 1$ digits.
- Show that it works for d digits.
- Radix sort of d digits ≡ radix sort of the low-order $d - 1$ digits followed by a sort on digit d .

Correctness of Radix Sort

By induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d , the elements are in order according to their low-order $d - 1$ digits. The sort on digit d will order the elements by their d^{th} digit.

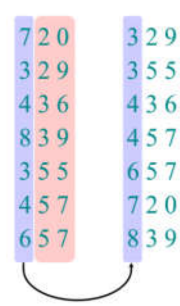
Consider two elements, a and b, with d^{th} digits a_d and b_d:

- If $a_d < b_d$, the sort will place a before b, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will place a after b, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order, since the sort is stable. But that order is already correct, since the correct order of is determined by the low-order digits when their d^{th} digits are equal.

Correctness of Radix Sort

Induction on digit position

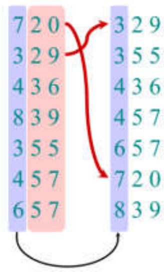
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



Correctness of Radix Sort

Induction on digit position

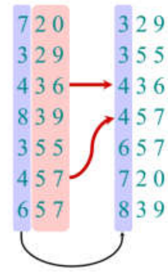
- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of Radix Sort

Induction on digit position

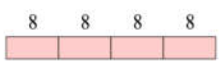
- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis of radix sort cont..

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into b/r equal pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

Choose r to minimize $T(n,b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.

Analysis of radix sort cont..

Choosing *r*

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(d\,n)$ time.

Algorithm Analysis

- Each pass over ***n* *d*-digit numbers** then takes time $\Theta(n+k)$. (Assuming counting sort is used for each pass.)
- There are d passes, so the **total time for radix sort is** $\Theta(d\ (n+k))$.
- When d is a constant and $k = O(n)$, radix sort runs in linear time.
- Radix sort, if uses counting sort as the intermediate stable sort, does not sort in place.
  - » If primary memory storage is an issue, quicksort or other sorting methods may be preferable.

Exercises

8.3-1
Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

Solution-1

COW		SEA		TAB		BAR
DOG		TEA		BAR		BIG
SEA		MOB		EAR		BOX
RUG		TAB		TAR		COW
ROW		DOG		SEA		DIG
MOB		RUG		TEA		DOG
BOX		DIG		DIG		EAR
TAB	$\Rightarrow$	BIG	$\Rightarrow$	BIG	$\Rightarrow$	FOX
BAR		BAR		MOB		MOB
EAR		EAR		DOG		NOW
TAR		TAR		COW		ROW
DIG		COW		ROW		RUG
BIG		ROW		NOW		SEA
TEA		NOW		BOX		TAB
NOW		BOX		FOX		TAR
FOX		FOX		RUG		TEA

Bucket Sort

- Assumes input is generated by a random process that distributes the elements uniformly over [0, 1).
- **Idea:**
  - » Divide [0, 1) into n equal-sized buckets.
  - » Distribute the n input values into the buckets.
  - » Sort each bucket.
  - » Then go through the buckets in order, listing elements in each one.

An Example

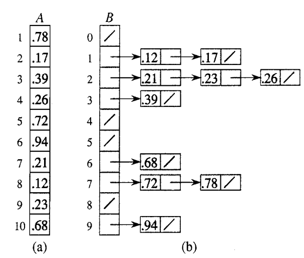


Figure 9.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket-Sort (A)

Input: $A[1..n]$, where $0 \le A[i] < 1$ for all i.

Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(A)
1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ to n
3. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n - 1$
5. **do** sort list $B[i]$ with insertion sort
6. concatenate the lists $B[i]$s together in order
7. **return** the concatenated lists

Analysis

- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ ***O(n)* sort time for all buckets.**
- We "expect" each bucket to have few elements, since the average is 1 element per bucket.

Exercises

Illustrate the operation of BUCKET-SORT on the array

$$A = [.79, .13, .16, .64, .39, .20, .89, .53, .71, .42].$$

Exercises

Illustrate the operation of BUCKET-SORT on the array

$$A = [.79, .13, .16, .64, .39, .20, .89, .53, .71, .42].$$

$$
\begin{array}{r|lllll}
0 & / & & & & \\
1 & \rightarrow & .13 & \rightarrow & .16 & / \\
2 & \rightarrow & .20 & / & & \\
3 & \rightarrow & .39 & / & & \\
4 & \rightarrow & .42 & / & & \\
5 & \rightarrow & .53 & / & & \\
6 & \rightarrow & .64 & / & & \\
7 & \rightarrow & .71 & \rightarrow & .79 & / \\
8 & \rightarrow & .89 & / & & \\
9 & / & & & &
\end{array}
$$