Linear-Time Sorting Algorithms

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn't sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$
    - A comparison sort must do O(n) comparisons (*why?*)
    - What about the gap between O(n) and O(n lg n)

Lower Bound For Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is "Sorting in linear time"!
  - *How can we do better than $\Omega(n \lg n)$?*

Sorting in Linear Time

- Comparison sort:
  - Lower bound: $\Omega(n \lg n)$.
- Non comparison sort:
  - Bucket sort, counting sort, radix sort
  - They are possible in linear time (under certain assumption).

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1*.. *k*
  - The algorithm:
    - Input: A[1..*n*], where A[j] ∈ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting in place)
    - Also: Array C[1..*k*] for auxiliary storage

Counting Sort

- Assumption: *n* input numbers are integers in range [0,*k*], *k*=*O*(*n*).
- Idea:
  - Determine the number of elements less than *x*, for each input *x*.
  - Place *x* directly in its position.

Counting Sort

```
1     CountingSort(A, B, k)
2           for i=1 to k
3                 C[i]= 0;
4           for j=1 to n
5                 C[A[j]] += 1;
6           for i=2 to k
7                 C[i] = C[i] + C[i-1];
8           for j=n downto 1
9                 B[C[A[j]]] = A[j];
10                C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
1     CountingSort(A, B, k)
2           for i=1 to k
3                 C[i]= 0;
4           for j=1 to n
5                 C[A[j]] += 1;
6           for i=2 to k
7                 C[i] = C[i] + C[i-1];
8           for j=n downto 1
9                 B[C[A[j]]] = A[j];
10                C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort

- Total time: O(n + k)
  - Usually, k = O(n)
  - Thus counting sort runs in O(n) time
- But sorting is $\Omega(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*

Example of Counting Sort

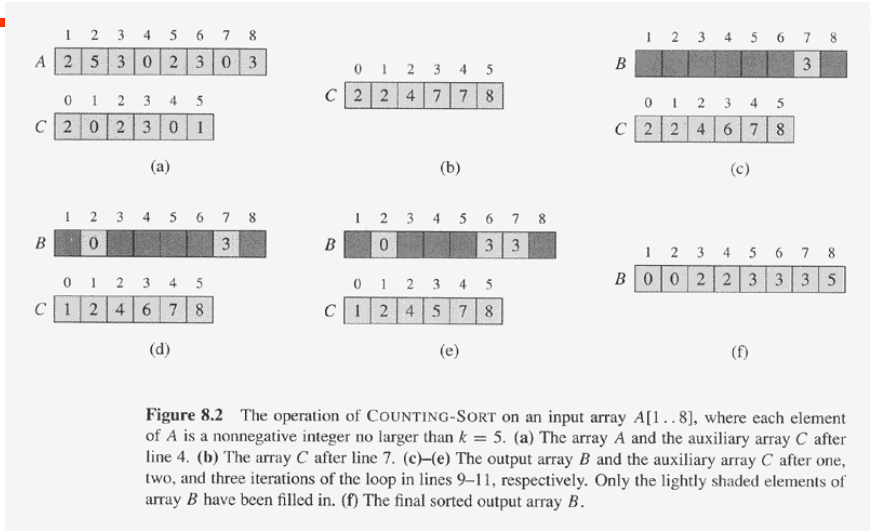

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after 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

- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)

Review: Counting Sort

- Counting sort:
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element i
    - Use that number to place i in position k of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place:
    - O(n) array to hold sorted output
    - O(k) array for scratch storage

Radix sort – Example 1

- Suppose a group of people, with last name, middle, and first name (each has one letter).
- For example: (z, x, k), (z,j,y), (f,s,f), …
- Sort it by the last name, then by middle, finally by the first name
- Solution 1:
  - sort by last name first as into (possible) 26 bins,
  - Sort each bin by middle name into (possible) 26 more bins (26*26 =512)
  - Sort each of 512 bins by the first name into 26 bins
- So if many names, there may need possible 26*26*26 bins.
- Suppose there are *n* names, there need possible *n* bins.

What is the efficient solution?

Radix sort – Example 1 contd.

- By first name, then middle, finally last name.
- Then after every pass of sort, the bins can be combined as one file and proceed to the next sort.
- Radix-sort(A,d)
  - For i=1 to d do
    - use a stable sort to sort array A on digit i.
- 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.
  - If d is constant and $k=O(n)$, then time is $\Theta(n)$.

Radix Sort – Example 2

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort – Example 2 contd.

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from 1..k
  - Time: O(n + k)
- Each pass over n numbers with d digits takes time O(n+k), so total time O(dn+dk)
  - When d is constant and k=O(n), takes O(n) time
- *How many bits in a computer word?*

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approx lg n = 20 operations per number being sorted
- *So why would we ever use anything but radix sort?*

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Summary: Radix Sort

- Radix sort:
  - Assumption: input has d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast! Stable! Simple!
  - Doesn't sort in place

Bucket Sort

- Bucket sort
  - Assumption: input is n reals from [0, 1)
  - Basic idea:
    - Create n linked lists (*buckets*) to divide interval [0,1) into subintervals of size $1/n$
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when we study *hash tables*

The End