Introduction to Algorithms

Sorting in Linear Time

CSE 680
Prof. Roger Crawfis

Comparison Sorting Review

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

Comparison Sorting Review

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort sub-arrays
    - Linear-time merge step
  - Pro's:
    - $O(n \lg n)$ worst case - *asymptotically optimal for comparison sorts*
  - Con's:
    - Doesn't sort in place

Comparison Sorting Review

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - Pro's:
    - $O(n \lg n)$ worst case - *asymptotically optimal for comparison sorts*
    - Sorts in place
  - Con's:
    - Fair amount of shuffling memory around

Comparison Sorting Review

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort
    - All of first sub-array < all of second sub-array
  - Pro's:
    - O(n lg n) average case
    - Sorts in place
    - Fast in practice (*why?*)
  - Con's:
    - O(n^2) worst case
      - Naïve implementation: worst case on sorted input
      - Good partitioning makes this very unlikely.

Non-Comparison Based Sorting

- Many times we have restrictions on our keys
  - Deck of cards: Ace->King and four suites
  - Social Security Numbers
  - Employee ID's
- We will examine three algorithms which under certain conditions can run in O(n) time.
  - Counting sort
  - Radix sort
  - Bucket sort

Counting Sort

- Depends on assumption about the numbers being sorted
  - Assume numbers are in the range *1.. k*
- The algorithm:
  - Input: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
  - Output: B[1..n], sorted (not sorted in place)
  - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

This is called a ***histogram***.

Counting Sort Example

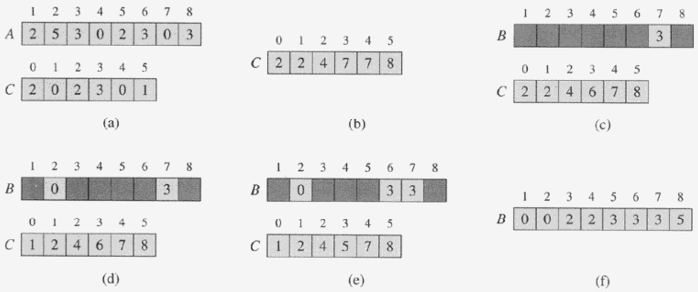

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after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

```
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 is the running time?

Counting Sort

- Total time: O(n + k)
  - Works well if k = O(n) or k = O(1)
- This algorithm / implementation is *stable*.
  - A sorting algorithm is ***stable*** when numbers with the same values appear in the output array in the same order as they do in the input array.

Counting Sort

- *Why don't we always use counting sort?*
  - Depends on range k of elements.
- *Could we use counting sort to sort 32 bit integers? Why or why not?*

Counting Sort Review

- **Assumption:** input taken from **small** set of **numbers** of size *k*
- Basic idea:
  - Count number of elements less than you for each element.
  - This gives the position of that number – similar to selection sort.
- Pro's:
  - Fast
  - Asymptotically fast - O(*n*+*k*)
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be ~~integers.~~ ***countable***
  - Requires O(*n*+*k*) extra storage.

Radix Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix Sort

- 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
```

Radix Sort Example

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

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 Correctness

- Sketch of an inductive proof of correctness (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

Radix Sort

- *What sort is used 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

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!
- Performs well compared to typical $O(n \lg n)$ comparison sort
  - Approx **lg**(1,000,000) $\cong$ 20 comparisons per number being sorted

Radix Sort Review

- **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
- Pro's:
  - Fast
  - Asymptotically fast (i.e., $O(n)$ when d is constant and $k=O(n)$)
  - Simple to code
  - A good choice
- Con's:
  - Doesn't sort in place
  - Not a good choice for floating point numbers or arbitrary strings.

Bucket Sort

Assumption: input elements distributed uniformly over some known range, e.g., [0,1), so all elements in A are greater than or equal to 0 but less than 1 . (Appendix C.2 has definition of uniform distribution)

```
Bucket-Sort(A)
1. n = length[A]
2. for i = 1 to n
3.        do insert A[i] into list B[floor of nA[i]]
4. for i = 0 to n-1
5.        do sort list i with Insertion-Sort
6. Concatenate lists B[0], B[1],…,B[n-1]
```

Bucket Sort

Bucket-Sort(A, *x, y*)
1. divide interval *[x, y)* into *n* equal-sized subintervals (buckets)
2. distribute the *n* input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort)
4. scan the (sorted) buckets in order and produce output array

<u>Running time of bucket sort:</u> *O(n)* expected time

Step 1: *O(1)* for each interval = *O(n)* time total.
Step 2: *O(n)* time.
Step 3: The expected number of elements in each bucket is *O(1)* *(see book for formal argument, section 8.4),* so total is *O(n)*
Step 4: *O(n)* time to scan the *n* buckets containing a total of *n* input elements

Bucket Sort Example

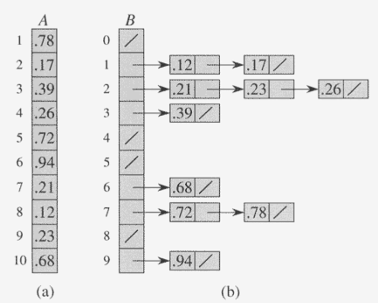

Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket Sort Review

- **Assumption:** input is uniformly distributed across a range
- Basic idea:
  - Partition the range into a fixed number of buckets.
  - Toss each element into its appropriate bucket.
  - Sort each bucket.
- Pro's:
  - Fast
  - Asymptotically fast (i.e., O(*n*) when distribution is uniform)
  - Simple to code
  - Good for a rough sort.
- Con's:
  - Doesn't sort in place

Summary of Linear Sorting

Non-Comparison Based Sorts

	Running Time			
	worst-case	average-case	best-case	in place
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	no
Radix Sort	$O(d(n + k'))$	$O(d(n + k'))$	$O(d(n + k'))$	no
Bucket Sort		$O(n)$		no

Counting sort assumes input elements are in range [0,1,2,..,k] and uses array indexing to count the number of occurrences of each value.

Radix sort assumes each integer consists of d digits, and each digit is in range [1,2,..,k'].

Bucket sort requires advance knowledge of input distribution (sorts *n* numbers uniformly distributed in range in O(*n*) time).