Sorting in Linear Time

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 case

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
  - 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

- Why it's not a comparison sort:
  - Assumption: input - integers in the range 0..k
  - No comparisons made!
- Basic idea:
  - determine for each input element x its rank: the number of elements less than x.
  - once we know the rank r of x, we can place it in position r+1
- 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] {1, 2, 3, …, k}
  - Output: B[1..n], sorted (not sorted in place)
  - Also: Array C[1..k] for auxiliary storage

Counting Sort 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

$C =$

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

$B =$

1	2	3	4	5	6	7	8
						3	

$C =$

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

Counting Sort Example

	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	6	7	8

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

	0	1	2	3	4	5
$C =$	1	2	4	6	7	8

Counting Sort 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	2	4	6	7	8

$B =$

1	2	3	4	5	6	7	8
	0				3	3	

$C =$

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

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;
```

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 $(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are no comparisons at all!)
  - Notice that this algorithm is stable
  - If numbers have the same value, they keep their original order

Stable Sorting Algorithms

- A sorting algorithms is stable if for any two indices i and j with i < j and ai = aj, element ai precedes element aj in the output sequence.

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

Observation: *Counting Sort is stable.*

Counting Sort

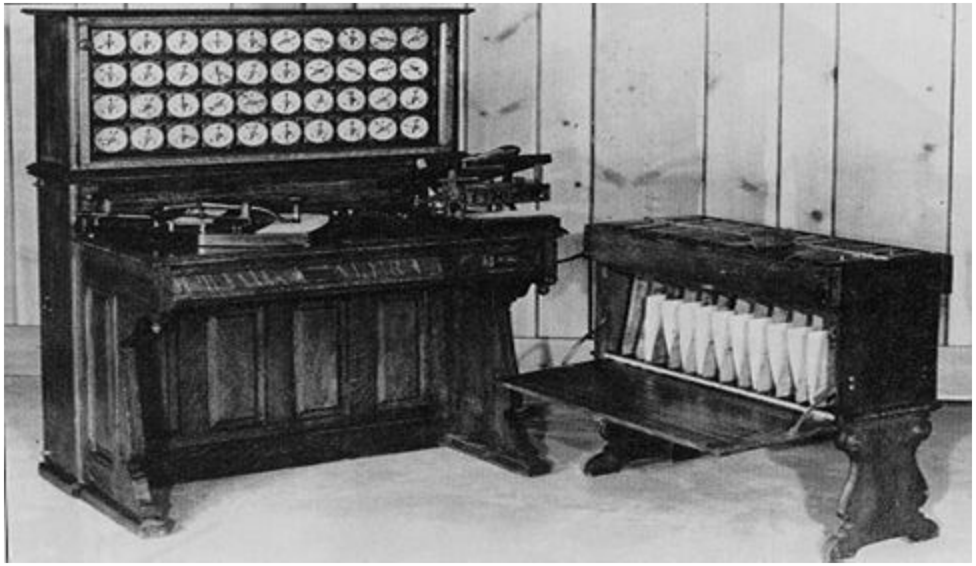
- Linear Sort! Cool! Why don't we always use counting sort?
- Because it depends on range k of elements
- Could we use counting sort to sort 32 bit integers? Why or why not?
- Answer: no, k too large (2^{32} = 4,294,967,296)

Counting Sort

- 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

- 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

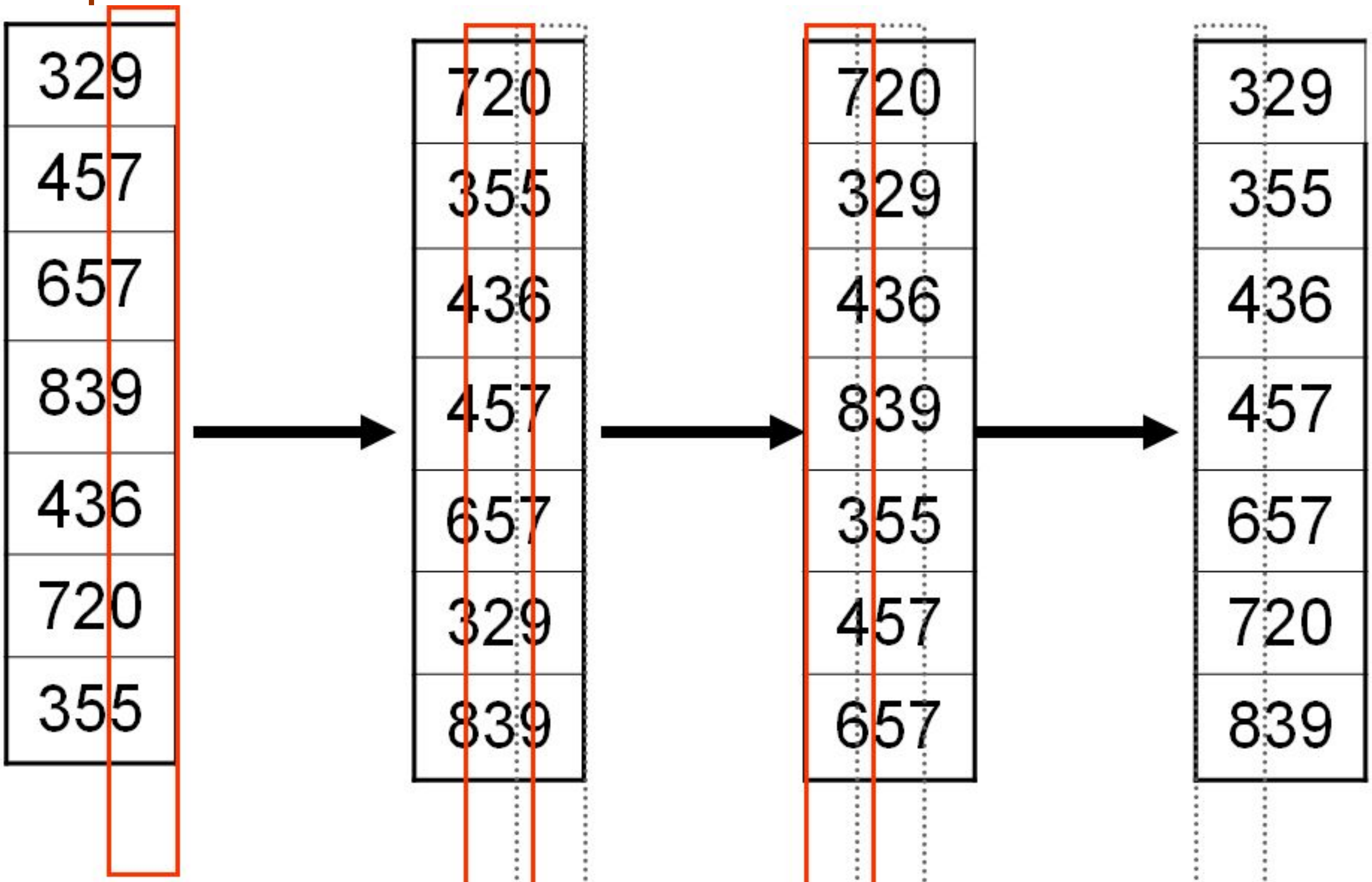
IBM 083
punch card
sorter

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



Radix Sort

- What is the running time of radix sort?
  - Each pass over the 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
- Stable, Fast
- Doesn’t sort in place (because counting sort is used)

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 216 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) 20 comparisons per number being sorted

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
- 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

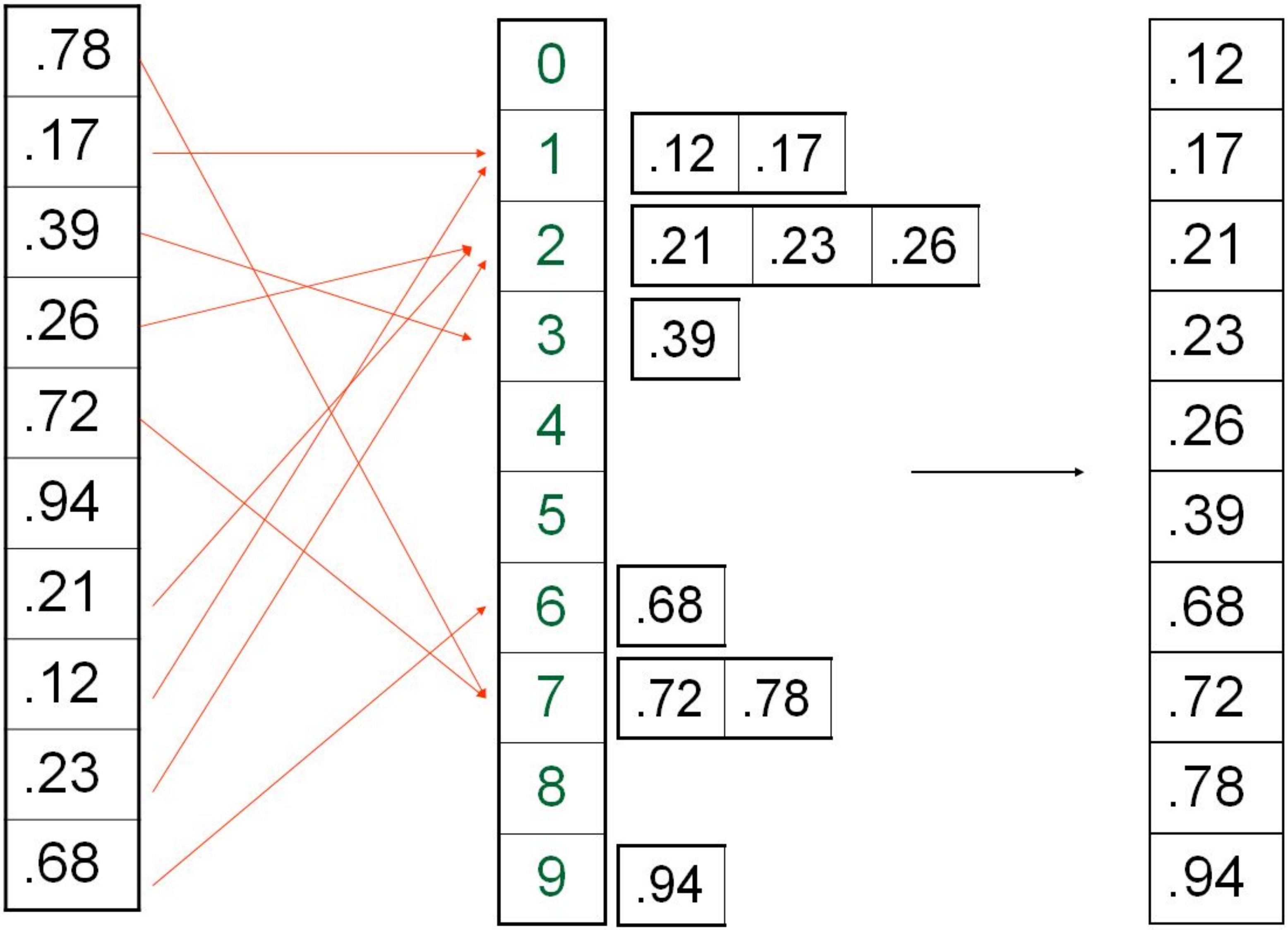
- Assumption: input - n real numbers from [0, 1)
- Basic idea:
  - Create n linked lists (buckets) to divide interval [0,1) into subintervals of size 1/n
  - Add each input element to appropriate bucket and sort buckets with insertion sort
- Uniform input distribution O(1) bucket size
  - Therefore the expected total time is O(n)

Bucket Sort

<u>**Bucket-Sort**</u>(A)

1. $n \leftarrow$ length(A)
2. **for** $i \leftarrow 0$ **to** n *Distribute elements over buckets*
3. **do** insert $A[i]$ into list $B[\text{floor}(n*A[i])]$
4. **for** $i \leftarrow 0$ to $n-1$ *Sort each bucket*
5. **do** Insertion-Sort($B[i]$)
6. Concatenate lists $B[0]$, $B[1]$, … $B[n-1]$ in order

Bucket Sort Example



Bucket Sort-Running Time

- All lines except line 5 (Insertion-Sort) take O(n) in the worst case.
- In the worst case, O(n) numbers will end up in the same bucket, so in the worst case, it will take O(n2) time.
- Lemma: Given that the input sequence is drawn uniformly at random from [0,1), the expected size of a bucket is O(1).
- So, in the average case, only a constant number of elements will fall in each bucket, so it will take O(n) (see proof in book).
- Use a different indexing scheme (hashing) to distribute the numbers uniformly.

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).