CS146: Data Structures and Algorithms
Lecture 9

LINEAR TIME SORTING
LOWER BOUNDS

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Sorting So Far – 1st Algorithm

- Insertion sort:
  - Easy implementation
  - 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 – 2nd Algorithm

- 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 – 3rd Algorithm

- Heap sort:
  - Uses the 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 – 4th Algorithm

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All elements of first subarray < all elements 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
  - by playing a different game

- 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 $\Omega(n \log n)$
    - A comparison sort must do $\Omega(n)$ comparisons (*why?*)
    - What about the gap between $\Omega(n)$ and $\Omega(n \log n)$

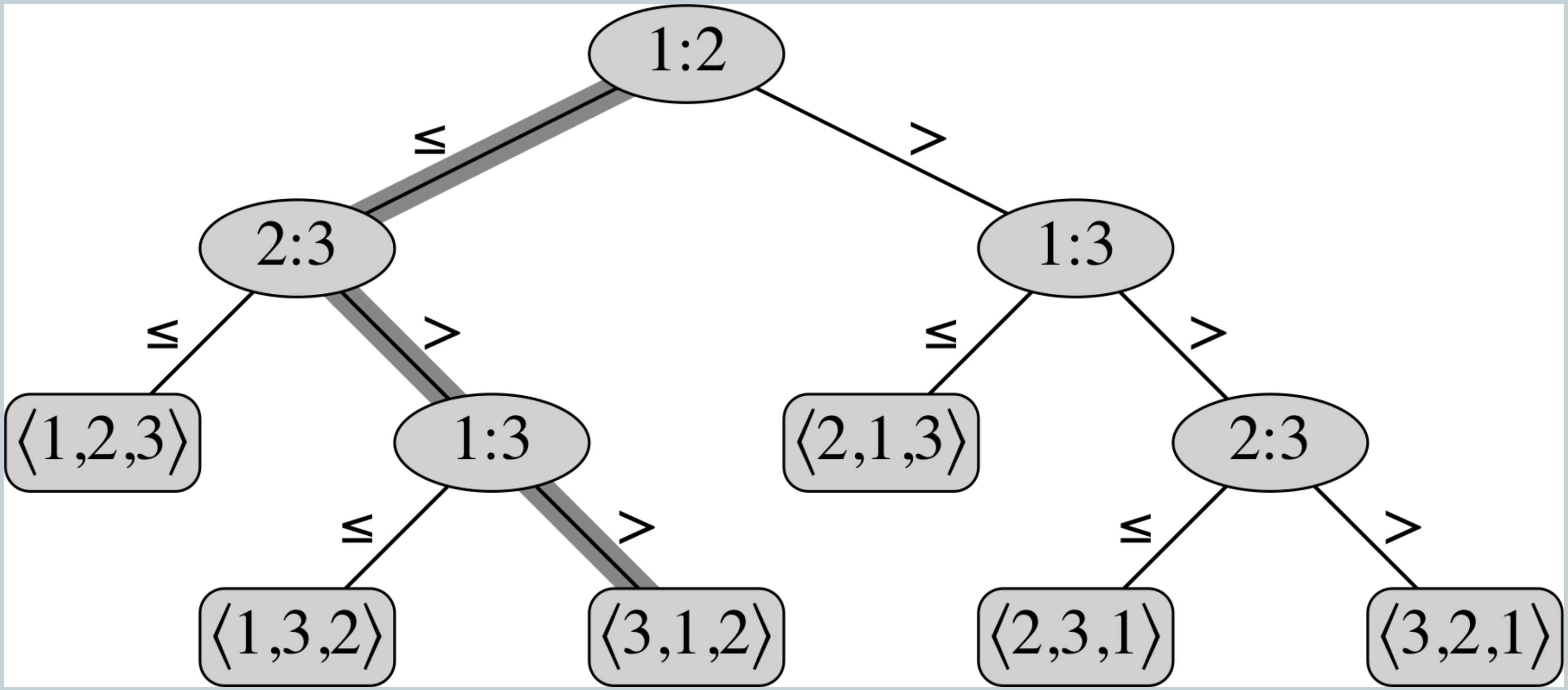
Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Everything else is ignored
- *What do the leaves represent?*
  - leaf is labeled by the permutation of orders that the algorithm determines
- *How many leaves must there be?*
  - There are $\geq n!$ leaves, because every permutation appears at least once.

Note: Permutations (Appendix C)

- A ***permutation*** of a finite set S is an ordered sequence of all the elements of S, each element appearing exactly once.
-
- For example, if S= {a, b, c}, then S has 6 permutations: *abc, acb, bac, bca, cab, cba*
- There are n! permutations of a set of n elements
  - we can choose the first element of the sequence in n ways, the second in n-1 ways, the third in n-2 ,etc.

Example: insertion sort for 3 numbers



Decision Trees

- Decision trees can model comparison sorts. For a given algorithm:
  - One tree for each *n*
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? For merge sort?*
- *What is the asymptotic height of any decision tree for sorting n elements?*
- Answer: $\Omega(n \log n)$ (proof follows)

Lower Bound For Comparison Sorting

- Thm: Any decision tree that sorts n elements has height $\Omega(n \log n)$
- *What's the maximum # of leaves of a binary tree of height h?*
- Lemma: Any binary tree of height h has $k \leq 2^h$, where k: # of leaves (proof by induction)

Lower Bound For Comparison Sorting

- So we have…
  $n! <= 2^h$
- Taking logarithms:
  $\lg (n!) <= h$
- Stirling's approximation tells us:

$$n! > \left(\frac{n}{e}\right)^n$$

- Thus:

$$h \geq \log\left(\frac{n}{e}\right)^n$$

Lower Bound For Comparison Sorts

- So we have

$$h \geq \log\left(\frac{n}{e}\right)^n$$

$$= n \log n - n \log e$$

$$= \Omega(n \log n)$$

- Thus the minimum height of a decision tree is $\Omega(n \log n)$

Lower Bound For Comparison Sorts

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

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] $\in$ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting 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;
```

Counting Sort

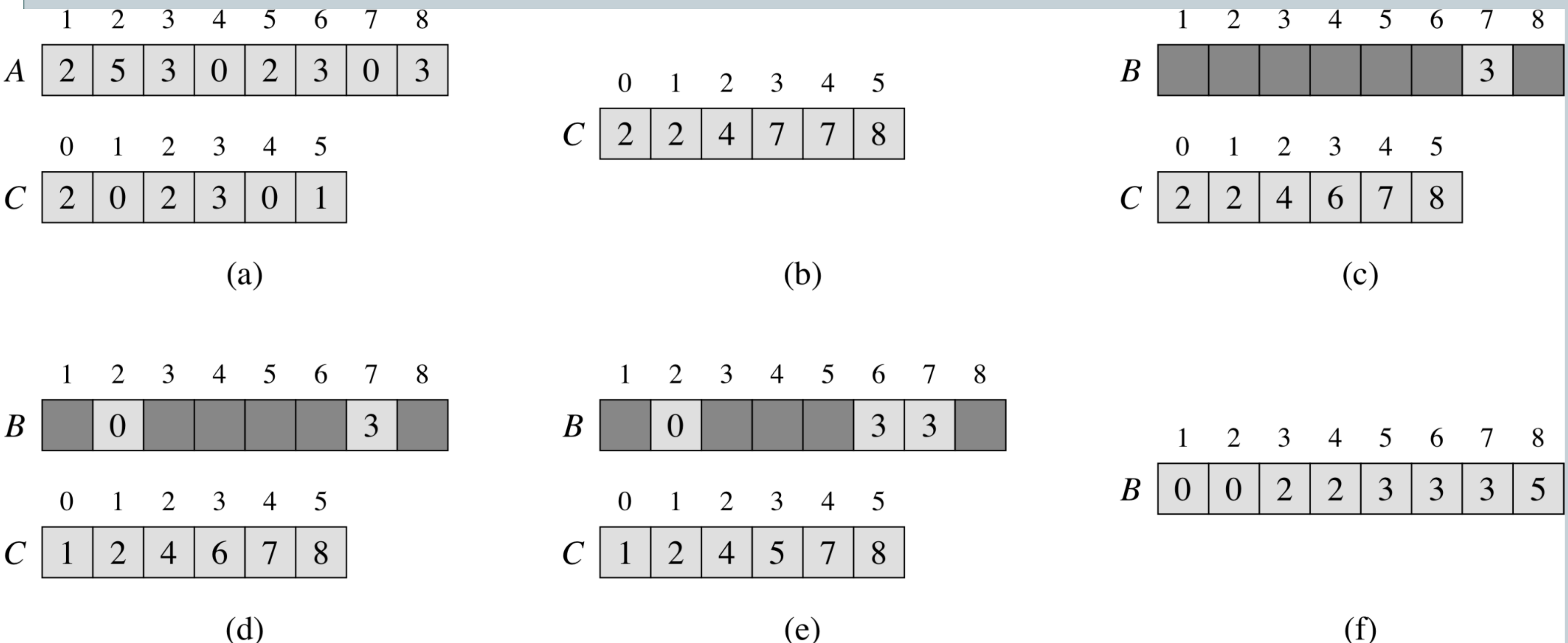

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

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 0 2 3 0 1
(a)
0 1 2 3 4 5
C 2 2 4 7 7 8
(b)
1 2 3 4 5 6 7 8
B 3
0 1 2 3 4 5
C 2 2 4 6 7 8
(c)
1 2 3 4 5 6 7 8
B 0 3
0 1 2 3 4 5
C 1 2 4 6 7 8
(d)
1 2 3 4 5 6 7 8
B 0 3 3
0 1 2 3 4 5
C 1 2 4 5 7 8
(e)
1 2 3 4 5 6 7 8
B 0 0 2 2 3 3 3 5
(f)

Counting Sort

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

- **Stable: keys with same value appear in same order in output as they did in input**

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? How many possible (distinct) numbers can we have?*
- Answer: no, *k* too large ($2^{32} = 4,294,967,296$)

Counting 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
```

Example of Radix Sort

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

Radix Sort

- *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

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?*

How to break each key into digits?

- n words
- b bits/word
- Break into r-bit digits. Have $d = \lceil b/r \rceil$ digits
- Use counting sort with $k = 2^r - 1$
- Example: 32-bit words, 8-bit digits. $b = 32$, $r = 8$, $d = 32/8 = 4$, $k = 2^8 - 1 = 255$.
- • Time = $\Theta(b/r(n + 2^r))$.
- Choose r ≈ log n gives: Θ(bn/ log n).

Bucket Sort

- Assumes the input is generated by a random process that distributes 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 buckets in order, listing elements in each one.

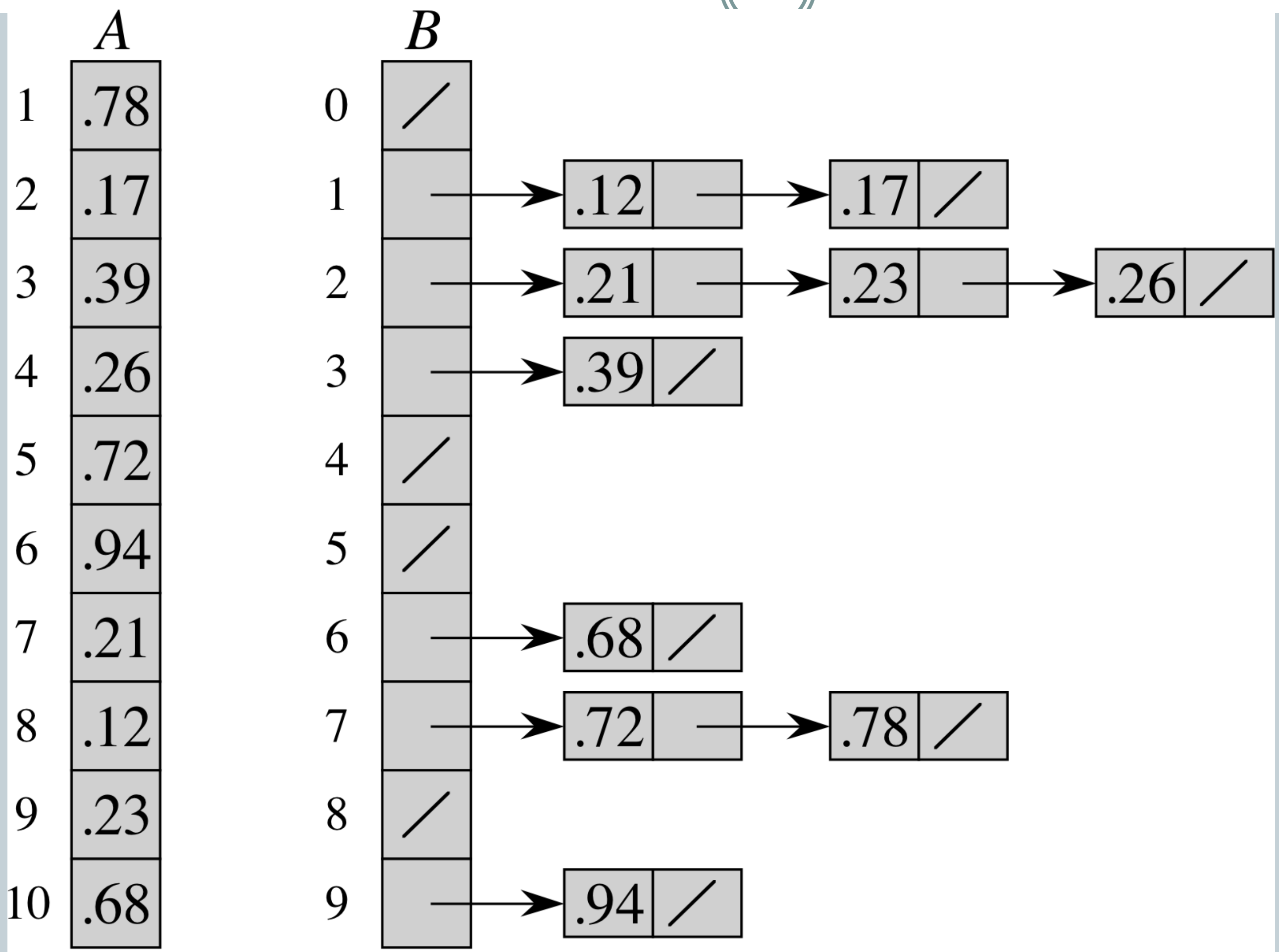
Input: $A[1 \,.\,.\, n]$, where $0 \le A[i\,] < 1$ for all i .

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

Bucket sort Code


```
BUCKET-SORT(A, n)
for i ← 1 to n
   insert A[i ] into list B[⌊n · A[i]⌋]
for i ← 0 to n − 1
    sort list B[i ] with insertion sort
    concatenate lists B[0], B[1], . . . , B[n
− 1] together in order
return the concatenated lists
```

Bucket Sort Example



(a) (b)

Correctness

- Consider $A[i\ ]$, $A[\ j\ ]$.
- Assume without loss of generality that $A[i\ ] \leq A[\ j\ ]$.

  Then $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$.
- Case1*:* $A[i\ ]$ is placed into the same bucket as $A[\ j\ ]$ or
- Case2: into a bucket with a lower index (Case 2).

If same bucket (Case1), insertion sort fixes up.

If earlier bucket (Case2), concatenation of lists fixes up.

Analysis:

- Relies on *no bucket* getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- 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.
- Uniform input distribution has O(1) bucket size and expected time is O(n)
- Later in Hash Tables again the same idea

Structures…

- Done with sorting and order statistics

- Next part is *data structures Ch 10 (skim)*
- *Ch 11*