Sorting

CS211
Fall 2000

Insertion Sort

- Corresponds to how most people sort cards
- Invariant: everything to left is already sorted
- Works especially well when input is *nearly sorted*
- Runtime
  - Worst-case
    - $O(n^2)$
    - Consider reverse-sorted input
  - Best-case
    - O(n)
    - Consider sorted input

```
// Code for sorting a[ ] an array of int
for (int i = 1; i < a.length; i++) {
    int temp = a[ i ];
    int k = i;
    for (; k > 0 && a[k–1] > temp; k – –)
        a[k] = a[k–1];
    a[k] = temp;
}
```

2

Merge Sort

- Uses recursion (Divide & Conquer)
- Outline (text has detailed code)
  - Split array into two halves
  - Recursively sort each half
  - Merge the two halves
- Merge = combine two sorted arrays to make a single sorted array
  - Rule: Always choose the smallest item
  - Time: O(n)
- Runtime recurrence
  - Let T(n) be the time to sort an array of size n
  - T(n) = 2T(n/2) + O(n)
  - T(1) = O(1)
  - Can show by induction that T(n) = O(n log n)
- Alternately, can show T(n) = O(n log n) by looking at tree of recursive calls

3

Quick Sort

- Also uses recursion (Divide & Conquer)
- Outline
  - *Partition* the array
  - Recursively sort each piece of the partition
- *Partition* = divide the array like this [$\le p$ | p | $\ge p$]
- p is the *pivot* item
- Best pivot choices
  - middle item
  - random item
  - median of leftmost, rightmost, and middle items
- Runtime analysis (worst-case)
  - Partition can work badly producing this: [p | $\ge p$]
  - Runtime recurrence
    T(n) = T(n–1) + O(n)
  - This can be solved by induction to show T(n) = $O(n^2)$
- Runtime analysis (expected-case)
  - More complex recurrence
  - Can solve by induction to show
    expected T(n) = O(n log n)
- Can improve constant factor by avoiding QSort on small sets

4

Heap Sort

- Not recursive
- Outline
  - Build heap
  - Perform removeMax on heap until empty
  - Note that items are removed from heap in sorted order
- Heap Sort is the only O(n log n) sort that uses *no* extra space
  - Merge Sort uses extra array during merge
  - Quick Sort uses recursive stack
- Runtime analysis (worst-case)
  - O(n) time to build heap (using bottom-up approach)
  - O(log n) time (worst-case) for each removal
  - Total time: O(n log n)

5

Sorting Algorithm Summary

- The ones we have discussed
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
- Other sorting algorithms
  - Selection Sort
  - Shell Sort (in text)
  - Bubble Sort
  - Radix Sort
  - Bin Sort
  - Counting Sort
- Why so many? Do Computer Scientists have some kind of sorting fetish or what?
  - Stable sorts: *Ins, Mer*
  - Worst-case O(n log n): *Mer, Hea*
  - Expected-case O(n log n): *Mer, Hea, Qui*
  - Best for nearly-sorted sets: *Ins*
  - No extra space needed: *Ins, Hea*
  - Fastest in practice: *Qui*
  - Least data movement: *Sel*

6

Lower Bounds on Sorting: Goals

- Goal: Determine the minimum time *required* to sort *n* items
- Note: we want *worst-case* not *best-case* time
  - Best-case doesn't tell us much; for example, we know Insertion Sort takes O(n) time on already-sorted input
  - We want to determine the *worst-case* time for the *best-possible* algorithm
- But how can we prove anything about the *best possible* algorithm?
  - We want to find characteristics that are common to *all* sorting algorithms
  - Let's try looking at *comparisons*

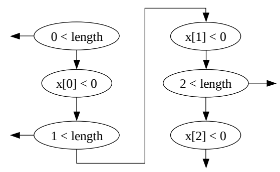
7

Comparison Trees

- Any algorithm can be "unrolled" to show the comparisons that are (potentially) performed

Example

```
for (int i = 0; i < x.length; i++)
  if (x[i] < 0) x[i] = – x[i];
```



- In general, you get a *comparison tree*
- If the algorithm fails to terminate for some input then the comparison tree is infinite
- The height of the comparison tree represents the *worst-case number of comparisons* for that algorithm

8

Lower Bounds on Sorting: Notation

- Suppose we want to sort the items in the array B[ ]
- Let's name the items
  - a_1 is the item initially residing in B[1], a_2 is the item initially residing in B[2], etc.
  - In general, a_i is the item initially stored in B[i]
- Rule: an item keeps its name forever, but it can change its location
  - Example: after swap(B,1,5), a_1 is stored in B[5] and a_5 is stored in B[1]

9

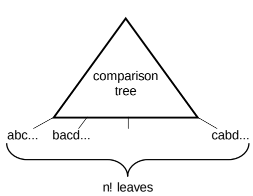
The *Answer* to a Sorting Problem

- An *answer* for a sorting problem tells where each of the a_i resides when the algorithm finishes
- How many answers are possible?
- The *correct* answer depends on the actual values represented by each a_i
- Since we don't know what the a_i are going to be, it has to be *possible* to produce each permutation of the a_i
- For a sorting algorithm to be valid it must be possible for that algorithm to give any of n! potential answers

10

Comparison Tree for Sorting

- Every sorting algorithm has a corresponding *comparison tree*
  - Note that other stuff happens during the sorting algorithm, we just aren't showing it in the tree
- The comparison tree must have n! (or more) leaves because a valid sorting algorithm must be able to get any of n! possible answers
- Comparison tree for sorting n items:



11

Time vs. Height

- The worst-case time for a sorting method must be ≥ the height of its comparison tree
  - The height corresponds to the worst-case number of comparisons
  - Each comparison takes Θ(1) time
  - The algorithm is doing more than just comparisons
- What is the minimum possible height for a binary tree with n! leaves?

  Height ≥ log(n!) = Θ(n log n)
- This implies that <u>any</u> comparison-based sorting algorithm <u>must</u> have a worst-case time of Ω(n log n)
  - Note: this is a lower bound; thus, the use of big-Omega instead of big-O

12

Using the Lower Bound on Sorting

Claim: I have a PQ
- Insert time: O(1)
- GetMax time: O(1)

- True or false?

False (for general sets) because if such a PQ existed, it could be used to sort in time O(n)

Claim: I have a PQ
- Insert time: O(loglog n)
- GetMax time: O(loglog n)

- True or false?

False (for general sets) because it could be used to sort in time O(n loglog n)

True for items with priorities in range 1..n [van Emde Boas] (Note: such a set can be sorted in O(n) time)

13

Sorting in Linear Time

There are several sorting methods that take linear time

- Counting Sort
  - sorts integers from a small range: [0..k] where k = O(n)
- Radix Sort
  - the method used by the old card-sorters
  - sorting time O(dn) where d is the number of "digits"
- How do these methods get around the Ω(n log n) lower bound?
  - They don't use comparisons
- What sorting method works best?
  - QuickSort is best general-purpose sort
  - Counting Sort or Radix Sort can be best for *some* kinds of data

14