Bounds on Sorting

Lecture 14
CS211 – Fall 2006

Prelim Announcements

- Prelim 1
  - Tonight 7:30 – 9:00pm
  - Last names starting with **A-B**: HO 314
  - Last names starting with **C-E**: HO 206
  - Last names starting with **F-Z**: OH 155
- Grades will be available tomorrow (Friday)
  - This is the last day to drop a course
- Check course website for latest info

More Announcements

- Using consultants
  - Do not work in consulting room after receiving help
    - Work somewhere else so other students can ask questions
  - Do not use consultants as "human compilers"
    - You are responsible for testing your code on your own
    - Not incrementally with a consultant

Sorting Algorithm Summary

- The ones we have discussed
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort
- Other sorting algorithms
  - Heap Sort (come back to this)
  - Shell Sort (in text)
  - Bubble Sort (nice name)
  - Radix Sort
  - Bin Sort
  - Counting Sort
- Why so many? Do Computer Scientists have some kind of sorting fetish or what?
  - Stable sorts: Ins, Sel, 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, Sel, Hea
  - Fastest in practice: Qui
  - Least data movement: Sel

Programming Problem Strategies

- Goal: Make it easier to solve programming problems
- Basic Data Structures
  - I recognize this; I can use this well-known data structure
  - Examples: Stack, Queue, Priority Queue, Dictionary
- Algorithm Design Methods
  - I can design an algorithm to solve this
  - Examples: Divide & Conquer, Greedy, Dynamic Programming
- Problem Reductions
  - I can change this problem into another with a known solution
  - Or, I can show that a reasonable algorithm is most-likely impossible
  - Examples: reduction to network flow, NP-complete problems

Recall: Analysis of MergeSort

- Time for Merge is O(n) where n is the number of elements being merged
- Time for MergeSort

  T(n) = 2T(n/2) + O(n)
  and T(1) = O(1)

  Recurrence can be simplified to T(n) = 2T(n/2) + n

  Solution is T(n) = O(n log n)

- One solution method for this recurrence

  Can divide by n to get
  T(n)/n = T(n/2)/(n/2) + 1

  Define S(n) = T(n)/n

  S(n) = S(n/2) + 1

  Easy to see that
  S(n) = 2 + log n

  Thus T(n) = n(2 + log n) or
  T(n) = O(n log n)

Solving Recurrences

Recurrences are important when using Divide & Conquer to design an algorithm

Solution techniques:

- Can sometimes change variables to make it into a simpler recurrence
- Make a guess then prove the guess correct by induction
- Build a recursion tree and use it to determine solution
- Can use the *Master Method*
  - A "cookbook" scheme that handles many common recurrences

To solve $T(n) = aT(n/b) + f(n)$ compare $f(n)$ with $n^{\log_b a}$

- Solution is $T(n) = O(f(n))$ if f(n) grows more rapidly
- Solution is $T(n) = O(n^{\log_b a})$ if $n^{\log_b a}$ grows more rapidly
- Solution is $T(n) = O(f(n) \log n)$ if both grow at same rate

- Not an exact statement of the theorem [f(n) must be "well-behaved"]
- See text for a similar theorem

Recurrence Relation Examples

- T(n) = T(n-1) + 1 [Linear Search]
  - T(n) = O(n)
- T(n) = T(n-1) + n [QuickSort worst-case]
  - $T(n) = O(n^2)$
- T(n) = T(n/2) + 1 [Binary Search]
  - T(n) = O(log n)
- T(n) = T(n/2) + n
  - T(n) = O(n)
- T(n) = 2 T(n/2) + n [MergeSort]
  - T(n) = O(n log n)

Recurrences & CS211

- Solving recurrences is like integration
  - No general technique works for all recurrences
- For CS 211, we just expect you to remember a few common patterns

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*

Lower Bounds on Sorting: Notation

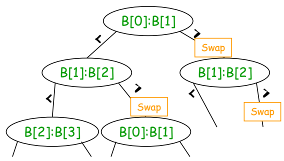
- Suppose we want to sort the items in the array B[ ]
- Let's name the items
  - a_1 is the item initially residing in B[1], a_2 is the item initially residing in B[2], etc.
  - In general, a_i is the item initially stored in B[i]
- Rule: an item keeps its name forever, but it can change its location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

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

Comparison Trees

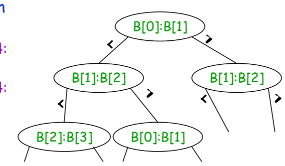
- *Any sorting algorithm performs some sequence of comparisons, depending on input*
  - Insertion sort on 2 3 1 4: 2<3? 3<1? 2<1? 3<4?
  - Insertion sort on 1 3 2 4: 1<3? 2<3? 1<2? 3<4?
- We can display a sorting algorithm as a tree showing the comparisons that can occur
- Insertion sort on B[0], B[1], B[2], B[3]



- We don't really need to show everything

Comparison Trees

- *Any sorting algorithm performs some sequence of comparisons, depending on input*
  - Insertion sort on 2 3 1 4: 2<3? 3<1? 2<1? 3<4?
  - Insertion sort on 1 3 2 4: 1<3? 2<3? 1<2? 3<4?
- We can display a sorting algorithm as a tree showing the comparisons that can occur
- Insertion sort on B[0], B[1], B[2], B[3]



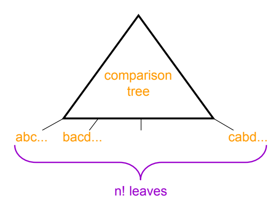
- We don't really need to show everything
  - Let's just show the comparisons

Comparison Tree Comments

- Any algorithm can be "unrolled" to show the comparisons that are (potentially) performed
  - 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 (plus one) represents the *worst-case number of comparisons* for that algorithm

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:



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 $\Theta(1)$ 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

Using the Lower Bound on Sorting

<u>Claim</u>: 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)

<u>Claim</u>: 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)

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"
  - Others…
- How do these methods get around the Ω(n log n) lower bound?
  - They don't use comparisons

Best Sorting Method?

- What sorting method works best?
  - QuickSort is best general-purpose sort
    - But it's not stable
  - MergeSort is a good choice if you need a stable sort
  - Counting Sort or Radix Sort can be best for some kinds of data