CMPSCI 575/MATH 513

Combinatorics and Graph Theory

Lecture #10: Sorting and Decision Trees
(Tucker Section 3.4)
David Mix Barrington
28 September 2016

Sorting and Decision Trees

- Review of Sorting Algorithms
- Decision Trees for Sorting
- The Sorting Lower Bound
- Adversary Arguments
- Lower Bound for Max or Min
- Lower Bound for Max *and* Min
- The Median Problem

Review of Sorting Algorithms

- Today's brief section of Tucker deals with sorting algorithms and the use of trees to analyze them.

- The **sorting problem** is to take an array of n items and move the items within the array so that they are in order.

- A general (**comparison-based**) sorting algorithm will work on any data type that supports a comparison operation.

Review of Sorting Algorithms

- We will assume for simplicity that there are no equal elements.
- Typically we count the comparison operations as they are the most expensive.
- The simplest sorting algorithms, Insertion and Selection, take $O(n^2)$ comparisons in the worst case. (In fact exactly n(n-1)/2.)
- In CS 187 and CS 311 we look at several algorithms that sort with O(n log n).

Review of Sorting Algorithms

- MergeSort splits the array in half, sorts each half recursively, and merges the results. We can make a tree of subarrays, halving in size as we go down each level, with depth log n.
- Merging takes O(n) comparisons (you'll look at the exact number on HW#3), so each merge on level i takes $O(n/2^i)$ comparisons. This amounts to O(n) on each level or O(n log n) total.

Review of Sorting

- HeapSort puts the items in a tree structure where each parent comes before both of its children. Of course this tree has depth log n.
- In O(log n) operations, it is possible to either (a) remove the root item and fix the structure, or (b) insert a new item and fix the structure.
- Simply put the items in one by one, then take them out one by one, and they are sorted.

Decision Trees for Sorting

- These algorithms each give an O(n log n) **upper bound** on the number of comparisons needed to sort. We'll use decision trees to get a matching **lower bound**.
- A decision tree has a node for each point in the algorithm where we compare two inputs. The two children of the node represent the two outcomes of the comparison. Leaf nodes represent outcomes where we know the entire order of the items.

Decision Trees for Sorting

- Note that a decision tree can represent any comparison-based algorithm, no matter what internal data it uses or how it decides which items to compare. Since the tree need not be arranged on any systematic basis, we call our argument a **non-uniform** lower bound.
- Any sorting algorithm yields a decision tree. For the algorithm to be correct in the worst case, it must be prepared to give any of the n! possible answers.

The Sorting Lower Bound

- If the number of leaves is n!, the height of tree must be at least log(n!)
- What is log(n!)? It's easy to see that log(n!) ≤ log(n^n) = n log n. But note also that log(n!) ≥ log($(n/2)^{n/2}$) = (n/2)(log n - 1). This proves log(n!) = Θ(n log n).
- Calculus estimates the sum for i from 1 to n of (ln i) as n((ln n) -1), telling us that n! is about $(n/e)^n$. Stirling's Formula gives a closer estimate.

Adversary Arguments

- My father has been a programmer since the 1950's. When I was young he had me write a computer program that *purported* to be an example of binary search.
- The program tells the user that it is thinking of a number in the range from 1 to 64, and invites them to make up to six guesses, after each of which it says "too high" or "too low".
- But the user always fails, as the answers are consistent with a number they don't guess.

Adversary Arguments

- This is an example of an **adversary argument**, showing that no possible algorithm can solve this guessing problem *in the worst case*.
- In general, you need ceiling(log(n+1)) guesses for n items. By creating an adversary that can choose a bad input *based on the guesses*, we show that a bad input exists.
- We can reframe the sorting lower bound in these terms.

Adversary Arguments

- At first, n! possible orderings of the n items exist.
- For each user comparison, the adversary gives the answer that is consistent with more orderings than the other.
- After the first guess, there are at least n!/2 consistent orderings. After g guesses, there are at least $n!/2^g$.
- If $g \leq \log(n!)$, there are two or more consistent orderings left, and the adversary picks one that the user didn't. Thus the user fails in the worst case.

Lower Bound for Max or Min

- Suppose the user just wants to find the maximum of the n items. There are n possible answers, so any decision tree has n leaves and depth at least log n. But we can prove a better bound with an adversary argument.
- If there are two or more items that are undefeated in comparisons, the adversary can pick one that the user doesn't. This must happen unless there have been at least n-1 comparisons to create n-1 losses.

Lower Bound for Max *and* Min

- What if the user wants both max and min?
- We first pair up the items and make n/2 comparisons (assume n is even).
- We then find the max of the winners and the min of the losers (n/2 - 1 each) for a total of 3n/2 - 2.
- This is actually optimal, as we can show with an adversary argument.

Lower Bound for Max *and* Min

- The adversary maintains for categories of items: A of untouched, B with wins and no losses, C with losses and no wins, and D with both wins and losses.
- We start with all n items in A, and we can beat the user at the end unless all but two items are in D (with one each in B and C).
- By picking winners of comparisons, we can make sure that the user needs n/2 to move the items from A to B and C, and n-2 more to get n-2 of them from B and C into D.

The Median Problem

- These are examples of **selection** problems. The hardest single selection problem is to find the **median** of the n elements.
- Again there are n possible answers, so decision trees give an $\Omega(\log n)$ lower bound.
- An adversary argument gives an $\Omega(n)$ bound.
- It's not obvious how to do it without sorting, but we can do it in O(n) on average by the main idea of QuickSort, and in O(n) deterministic time by a more complicated (and less practical) algorithm.