Supplment to *A Practical Guide to Data Structures and Algorithms Using Java*

Adversary Lower Bound Technique

Sally A. Goldman and Kenneth J. Goldman *Handout 2*

For any given problem there are generally many algorithms that can be devised for it. For example, for the problem of finding the closest pair of points in the plane, known algorithms include the quadratic time brute force algorithm of computing the distance between each pair of points and the $\Theta(n \log n)$ time divide-and-conquer algorithm. Likewise, for the problem of sorting n comparable elements, we have already seen insertion sort, mergesort, and quicksort. We know that insertion sort and quicksort have worst-case $O(n^2)$ performance[1], whereas mergesort has worst-case $O(n \log n)$ performance. Is it possible to create a sorting algorithm with a better worst-case asymptotic time complexity? That is, is there a sorting algorithm with worst-case time complexity of $o(n \log n)$? A similar question can be asked about the expected time complexity, but here we focus on deterministic (non-randomized) algorithms.

So the question we address is whether there is a better algorithm for a particular problem of interest. While lower bounds tell us what cannot be done, they are very important in guiding us in our search for what can be done. Lower bounds provide a way for us to quantify the intrinsic difficulty of a problem in a way that is not dependent on any particular algorithm or approach to solve it. Informally, a lower bound characterizes the fastest possible performance by a **correct** algorithm to solve a problem. If we do not make a restriction that the algorithm is correct, then we could not possibly make any claims about it. Also, in order to define a lower bound for a problem we must make some assumption about the model of computation. For example, the cost of a computation on a standard digital computer might be very different than for a "biological computer."

Comparison-Based Model of Computation

A very natural model of computation for the problem of sorting (as well as many other problems that are defined over a set of n comparable elements) is the **comparison-based computation model**. In this model, there is a collection of n comparable elements, and the algorithm can only gain information about the relative order of these n elements by making a comparison between any pair of them. We say that an algorithm is a *comparison-based algorithm* if it can be formulated under this model of computation. Observe that insertion sort, merge sort, and quicksort (along with many others) are all comparison-based sorting algorithms.

Observe that since each comparison requires at least one statement to be executed, the total computation time is at least as large as the number of comparisons made. Thus if we can prove that $\Omega(f(n))$ comparisons are required by any comparison based algorithm to solve problem P then we have proven an intrinsic limitation of $f(n)$ on how fast an algorithm can solve P. That is, there could not exist a comparison based algorithm to solve P in $o(f(n))$ time.

With the exception of only considering comparison-based algorithms, we are not going to place any other restriction on the algorithm. Thus, we must prove something about

[1]Randomized quicksort has expected time complexity $\Theta(n \log n)$.

algorithms that we have not seen, and that might use very different techniques than we have ever seen. In this handout, we describe the *adversary lower bound technique* to create such lower bounds. While this is a very general technique that can be applied to a wide variety of models of computation, we limit our discussion here to the comparison-based model.

Adversary Lower Bound Technique

You can think of the adversary lower bound technique as devising a strategy to construct a worst case input for an unknown *correct* algorithm to solve problem P. We will view this process as a game between an algorithm A and an adversary (or devil) D. We assume that D has unlimited computational power. In each round of this game, algorithm A asks D if element i is less than element j (for a choice of i and j made by A). The adversary D must answer "yes" or "no". This technique can be easily extended to when there are the three possible answers of $<$, $=$, and $>$.

The goal of algorithm A is to minimize the number of rounds until its computation to solve P is completed. Observe, that A can do *any* computation it wants between rounds without any cost to it. The goal of the adversary D is to maximize the number of rounds until A could be done. However, D has no control over what comparison A will make. However, as long as there are at least two possible answers to P that are consistent with all answers given by D, A cannot be done.

As a simple example, let's consider playing the game of "20 questions" against the adversary D. In this game, D picks an integer x between 1 and n, and the algorithm A must determine x by asking questions (to D) of the form "Is the number you have picked less than y?" We now argue that D can force A to ask at least $\lceil \log_2 n \rceil$ questions before A can be certain about the value for x. We call the way in which the adversary D plays this game the *adversary strategy* to be sure we do not confuse it with the algorithm A is using to determine x. The adversary is allowed to keep changing his mind about x but must answer in a consistent manner. That is, at the end of the game, the adversary must be able to give a value for x that is consistent with all answers given throughout the game. So it is possible that D did indeed have x in mind from the start.

Here is an adversary strategy that leads to the stated lower bound. The adversary maintains a list L of all possible values that are still legal for x. So initially the n integers $\{1, 2, 3, \ldots, n-1, n\}$ are placed in L. Each time A asks D if $x < y$, D counts how many of of the integers in L are greater than x. If at least half of the numbers in L are greater than x then D will respond "yes," and otherwise D will respond "no." If D responds "yes" (i.e., $x < y$) then all elements in the list that are $\geq y$ must be removed from L (or otherwise the adversary would be lying). Likewise, if D responds "no" (i.e., $x \geq y$) then all elements in the list that are $< y$ must be removed from L.

Observe that a correct algorithm A cannot know the value of x (and thus not finish executing) until L contains only a single item. (If there are two or more items in L then whatever A outputs could be the wrong one.) We must now determine how many rounds the adversary D can force before L could possibly reach size 1. Initially $|L| = n$. Now let's consider any single round of this game and let $|L| = s$. Since D responds in a way that least half of the items are consistent with the response, at the next round $|L| \geq \lceil s/2 \rceil$. Let L_i be the list after round i where L_0 is the initial list. Then we have that

- $|L_0| = n$, and
- $|L_{i+1}| \geq \lceil |L_i|/2 \rceil$ for $i > 0$

From this it follows that $i = \lceil \log_2 n \rceil$ is the smallest possible value for i for which $|L_i| = 1$. (This could be proven by induction.) Let's do an example to illustrate this where $n = 100$. By recursive definition:

$$|L_0| = 100,\ |L_1| \geq 50,\ |L_2| \geq 25,\ |L_3| \geq 13,\ |L_4| \geq 7,\ |L_5| \geq 4,\ |L_6| \geq 2,\ |L_7| \geq 1$$

Thus after 6 rounds A could not be done since there must be *at least* two values for x that are consistent with all answers given so far. Thus for this problem when $n = 100$, the best any algorithm A can do is to make 7 questions. Notice that we have not placed any restrictions at all on A. We have just shown that regardless of the algorithm A, the adversary strategy described above guarantees that A could not possibly be done until at least 7 rounds. Observe that $\lceil \log_2 100 \rceil = 7$. In general, any algorithm to solve this problem requires at least $\lceil \log_2 n \rceil$ questions. Note that we have not shown that there exists an algorithm that could achieve this bound, we are just saying it is the best possible.

$\Omega(n \log n)$ Lower Bound for the Sorting Problem

We now apply the adversary lower bound technique to the problem of sorting n comparable elements. Notice that there are $n!$ different permutations (and thus solutions) that the sorting algorithm must decide between. The adversary D maintains a list L of all of the permutations that are consistent with the comparisons that the algorithm has made so far. Initially L contains all $n!$ permutations. The adversary's strategy for responding to "Is element i less than element j" is as follows. Let L_{yes} be the permutations in L for which element i is less than element j, let L_{no} be the permutations in L for which element i is greater than or equal to element j. (So $L = L_{yes} \cup L_{no}$). Then the adversary responds "yes" exactly when $|L_{yes}| \geq |L_{no}|$. In other words, the adversary answers in such a way to keep L as large as possible. Then the adversary updates L so that only those permutations consistent with this answer remain. So if "yes" is answered then the permutations in L_{no} are removed, and if "no" is answered the permutations in L_{yes} are removed.

Since at least half of the permutations in L remain, and the algorithm cannot be done until $|L| = 1$, the number of comparisons required is at least $\lceil \log(n!) \rceil$. Another way to view this game, is the permutation that remains at the end of this game is an input that causes algorithm A to make at least $\lceil \log(n!) \rceil$ comparisons. If this adversary played against a different algorithm, then the final permutation in L might be different.

By Stirling's approximation $n! \geq (n/e)^n$ where e is Euler's constant. So the number of comparisons required by any comparison-based sorting algorithm is at least

$$\lceil \log_2(n!) \rceil \geq \lceil \log_2((n/e)^n \rceil \geq \lceil n \log_2 n - n \log_2 e \rceil = \Omega(n \log n).$$

Thus the worst-case asymptotic time complexity for any comparison-based sorting algorithm is $\Omega(n \log n)$. A similar, though more involved, technique can be used to show a $\Omega(n \log n)$ lower bound on the expected time complexity for any comparison-based sorting algorithm.