Lower Bounds

There are several types of lower bounds that are relevant in algorithm analysis. All are described in terms of big-omega notation.

1. A lower bound on a function — for example, one described by a recurrence relation, such as $T(n) = n + \frac{2}{n} \sum_{i=2}^{n-1} T(i)$ and $T(1) = 0$. We can state without knowing anything about where $T(n)$ came from that $T(n) \in \Omega(n \lg n)$.

   If $T(n)$ is an accurate description of the number of basic operations (worst- or average-case) of some algorithm, then the lower bound also applies to the algorithm.

2. A lower bound on the (worst-case) performance of an algorithm. For example, if we can show that there exist $c > 0$ and $n_0$ such that for all $n \geq n_0$, there exists at least one input of size $n$ that causes heapsort to perform $cn \lg n$ comparisons, we can say that the (worst-case) number of comparisons for heapsort is $\Omega(n \lg n)$.

   Important: we must exhibit a method for constructing “bad” inputs to justify our pessimism.

3. A lower bound on “all” algorithms solving a given problem. In the case of sorting, for example, what we would have to prove is: For any sorting algorithm $A$, there exist $c > 0$ and $n_0$.

   The worst-case example for a given $n$ may differ for different algorithms, but every algorithm must have such an example for every $n$.

   In general, it is impossible to prove such a statement. We usually need to restrict the class of algorithms we are willing to consider, so that we can devise a mathematical model for the behavior of a generic algorithm.

Comparison-based algorithms

A lower bound that applies to all comparison-based algorithms is based on the assumption that the only method by which the algorithm accesses a key is to compare two keys with each other.

General-purpose sorting algorithms are required to obey this assumption. For example, the Unix sort utility in the C/C++ library requires a function-pointer argument for specifying the key-comparison method. Therefore the implementation of the utility cannot access the keys directly.

An “adversary” argument

One model for proving worst-case lower bounds involves an adversary “playing against” an arbitrary algorithm and using a strategy that forces as many comparisons as possible to take place, as pictured below.

![Adversary Diagram]

For example, suppose we are sorting three distinct keys $x_1$, $x_2$, and $x_3$. There are 6 possible permutations:

- $x_1 < x_2 < x_3$
- $x_1 < x_3 < x_2$
- $x_2 < x_1 < x_3$
- $x_2 < x_3 < x_1$
- $x_3 < x_1 < x_2$
- $x_3 < x_2 < x_1$
The algorithm might begin by asking whether $x_1 < x_2$ (doesn’t matter — all comparisons are equivalent at this point). Regardless of what the adversary answers, exactly 3 possible permutations are eliminated.

The adversary’s answer to the next comparison, however, is critical to their strategy. One answer might eliminate 2 permutations while the other eliminates only one. The adversary must pick the latter to force the algorithm to do a third comparison (if the algorithm does not do another comparison, it cannot decide which permutation is the correct one).

The adversary has a simple strategy that will work against any comparison-based sorting algorithm: whenever the algorithm makes a comparison, choose the answer that will eliminate the fewest possible permutations.

If there are $n$ keys instead of 3, there are initially $n!$ possible permutations. After the algorithm has done $k$ comparisons, the adversary can guarantee that $\lceil \frac{n!}{2^k} \rceil$ permutations will still be possible.

If the algorithm claims to be done sorting when $\lceil \frac{n!}{2^k} \rceil > 1$, the adversary can always “reveal” values for the keys that are consistent with all $k$ answers, but not with the sorted order produced by the algorithm.

So the algorithm is not done until $\lceil \frac{n!}{2^k} \rceil \leq 1$. This translates to $2^k \geq n!$ or $k \geq \lceil \lg(n!) \rceil$.

It is not hard to show (as you’ve already done in homework) that $\lg(n!) = \lg(n(n-1)(n-2)\cdots1) = \sum_{i=1}^{n} \lg i$ is $\Omega(n \log n)$:

$$\sum_{i=1}^{n} \lg i \geq \sum_{\frac{n}{2} \leq i \leq n} \lg i \geq \frac{n}{2} \lg \frac{n}{2} \geq \frac{n}{4} \lg n$$

for sufficiently large $n$.

Thus any comparison-based sorting algorithm must do at least $\lceil \lg(n!) \rceil \in \Omega(n \log n)$ comparisons.

**Decision trees**

Any comparison-based algorithm can be represented by a decision tree that shows all possible sequences of comparisons that the algorithm could make. Each interior node of a decision tree represents a single comparison. The two children of a node represent the two outcomes of the comparison (yes/no, true/false). Each leaf represents a possible output. In the case of sorting algorithms, the outputs are the permutations of the keys (again, we assume all keys to be distinct).

For example, the decision tree representing insertion sort of 3 keys is shown below.