Lower Bounds

When studying algorithms, we are typically interested in their time complexity. An interesting question that comes up is whether a given algorithm is considered to be *optimal* or not. It turns out that this can be measured by computing the lower bound for a given problem.

In this class, we measure runtime by the number of uses of an oracle. Furthermore, an oracle is said to be a k-oracle if it can give k possible responses. For example:

- **Binary Comparison Oracle:** takes two keys as input (say x and y) and returns *true* if $x < y$ and *false* otherwise.
- **Binary Identification Oracle:** returns *true* if $x = y$ and *false* otherwise.
- **Ternary Oracle:** returns whether $x < y$, $x = y$, or $x > y$.
- **k!-Oracle:** takes in k keys and returns a permutation that is sorted.

In general, the complexity $T(A)$ is the number of times the algorithm A makes use of an oracle before halting. We would like to compare the worst-case complexity of an algorithm to a computed universal lower bound so that we can gauge how optimal an algorithm is.

Note: a particular algorithm can indeed run faster than the universal lower bound on some inputs, but we are interested in the worst-case runtime.

Decision Trees

When using an oracle that can give one of k responses, it can be fairly easily visualized as a decision tree, where each edge represents a possible response that the oracle can give and each vertex represents one use of the oracle. A leaf in the decision tree represents a point where the algorithm halts. Then, it makes sense that the worst-case runtime for an algorithm occurs when the deepest leaf is reached. If l is the deepest leaf in the tree, then we can express the worst-case runtime as:

$h = depth(l)$

We can observe that if there are L possible outcomes for an algorithm, then there must be at least L leaves in the tree. If there were less than L leaves, then the tree would not cover all possible outputs of the algorithm. There can also be more than L leaves if different paths through the tree yield the same result.

The important thing to know here is that for any k-ary tree with L leaves:

$h \geq log_k L$, and

$min_{\forall A} T_n(A) \geq \lceil log_k L \rceil$

This can be shown with induction. We can use this relationship to determine the lower bound for an algorithm.

Examples

Example 1: Sorting

When given a array of n numbers and a binary comparison oracle, we know that there are $n!$ possible permutations of the array. Of these permutations, at least one must give a sorted ordering. If we were to represent a sorting algorithm with a decision tree, it would have at least $n!$ leaves, hence:

$$min_{\forall A} T_n(A) \geq \lceil log_2 n! \rceil$$

Where A is an arbitrary algorithm. So far no algorithm has reached this lower bound, but algorithms like Mergesort come fairly close.

Example 2: Searching

Given a sorted array with n elements, we would like to find the index of a given element in the array. If it does not exist, we should return the index of where it would be inserted to maintain the sorted ordering. Again, we use a binary comparison oracle.

It is clear that the target element can be in one of n places if it is present in the array. However, if it is not present in the array, there are $n+1$ places where it could be inserted. Therefore the total number of outcomes is $n + (n+1) = 2n + 1$. Hence $L = 2n + 1$ and:

$$min_{\forall A} T_n(A) \geq \lceil log_2 (2n+1) \rceil$$

The binary search algorithm is known to be optimal for this problem.

Example 3: Smaller Subset

Given an array of n elements, we would like to return (using the binary comparison oracle) all elements that are smaller than a given target value. There are 2^n possible subsets, so the lower bound is:

$$min_{\forall A} T_n(A) \geq \lceil log_2 2^n \rceil = n$$

No algorithm can do better than this, since every element must be examined (in an unsorted array).