Order Notation and Estimating Complexity

We have looked at a few number of algorithms in class thus far. However, we have not looked at how to judge the efficiency or speed of an algorithm, which is one of the goals of this class.

We will use order notation to approximate two things about algorithms: how much time they take, and how much memory (space) they use.

The first thing to realize is that it will be nearly impossible to exactly figure out how much time an algorithm will take to execute on a particular computer. Each algorithm instruction gets translated into more small machine instructions, each of which take various amounts of time to execute on different computers. Also, we want to judge algorithms independent of their implementation. Thus, rather than figuring out an algorithm's exact running time, we will only want an approximation. The type of approximation we will be looking for is a Big-O approximation.

We will assume that each statement and each comparison in C takes some constant amount of time.

Also, most of the problems we will look at will have an input size. (For example, in sorting, the size of the input is the number of numbers to be sorted.) The time and space used by an algorithm will typically be a function of this input size. (The input size will typically be referred to as n.)

Big-Oh

Since we can't usually determine an exact number of steps an algorithm will take we'll be happy to make the two following simplifications in counting the number of steps an algorithm takes:

1) Eliminate any term whose contribution to the total ceases to be significant as n becomes large.

2) Eliminate constant factors.

Thus, if we happen to count that the number of steps a algorithm takes is $4n^2+3n - 5$, then we will

1) ignore 3n-5 because that accounts for a small number of steps as n gets large

2) Eliminate the constant factor of 4 in front of the n^2 term.

In doing so, we will conclude that the algorithm takes $O(n^2)$ steps.

In CS2, you will be introduced to the actual definition of Big-Oh. This is a simplification of the actual definition that is useful for most practical situations.

How does Order Notation Help Us Evaluate Algorithms?

Since we've determined that it may be too difficult to count up the exact number of steps an algorithm will take, we only want to be able to approximate an upper bound for the number of steps, within a constant factor. Hence, rather than saying that an algorithm will run $n^2 + n$ steps, we will be content to say that it runs $O(n^2)$ steps, since $n^2 + n = O(n^2)$.

One of the classic case studies in algorithms, (and usually the first taught to students), is the sorting problem. You may have informally seen this problem in COP 3223, but in this class we'll analyze the problem in far more detail and review even algorithms to sort a list of elements that you'll probably be sick of them by the end of it! We will use this problem as an introduction to algorithm analysis.

Average Case and Worst Case

When we are talking about the running time of an algorithm, you'll notice that depending on the input, a program may run more quickly or slowly. For example, if you are given a list of already sorted numbers, Insertion sort will run more quickly than if you give it a list of numbers sorted in descending order. (In this case, the maximum number of swaps occurs at each loop iteration.)

Thus, when we try to analyze running times of algorithms, we must acknowledge the fact that these running times may vary based on the actual input to the algorithm.

In our analysis then, we are typically concerned with two things:

1) **What is the worst possible running time an algorithm can achieve, given any input, AND**
2) **What is the average, or expected running time of an algorithm, averaged over all possible inputs.**

As you might imagine, #2 is very useful information but may be difficult to compute. To compute #1, you have to usually figure out what input will cause the algorithm to act most inefficiently (such as a descending list of numbers in Insertion Sort), and then simply calculate how long the algorithm would take to run based on that worst-case input.

However, if we can show that the best case(the fastest possible running of an algorithm on any input) and worst case running times an algorithm can achieve are the same big-O bound. We can make the same claim for the average case as well. This is the case for Selection Sort but NOT Insertion Sort.

Other things to consider when analyzing algorithms

When computing the average case running time, we may assume that all inputs are random, or equally likely. However, this may not always be the case. For example, it is possible that you are running an algorithm where the user has several choices from a menu. It may turn out that a particular choice gets chosen far more often than the other choices. In this case, assuming that each case is chosen equally may not give you an accurate average case running time.

Practical problems that can be solved utilizing order notation

For example, if you are told that Algorithm A runs in O(n) time, and for an input size of 10, the algorithm runs in 2 milliseconds, you can expect it to take 100 milliseconds to run on an input size of 500.

Here is how you can handle problems like this in general:

Since we are assuming that the Big-O bound given for an algorithm is relatively accurate (to within a constant factor) to the actual running time, if we say an algorithm runs in O(f(n)) time, assume that the exact running time is c*f(n), where c is some constant. Using this assumption, you can use the information given in the problem to solve for c. Once you have done that, you should be able to answer the question being asked. Let's do a couple other examples:

Practice Problems

Algorithm A runs in $O(n^2)$ time, and for an input size of 4, the algorithm runs in 10 milliseconds, how long can you expect it to take to run on an input size of 16?

Let $T(n) = cn^2$. Using the given information, we have
$T(4) = c4^2 = 10ms$, so
$c = 10/16$ ms

To answer the given question solve for $T(16)$:

$T(16) = (10/16)*16^2 = 160$ ms.

Algorithm A runs in $O(\lg_2 n)$ time, and for an input size of 16, the algorithm runs in 28 milliseconds, how long can you expect it to take to run on an input size of 64?

Let $T(n) = c\lg_2 n$. Using the given information we have
$T(16) = c\lg_2 16 = 28$ ms
$4c = 28ms$
$c = 7$ ms

Now, solve for $T(64)$:

$T(64) = 7* \lg_2 64$ ms $= 42$ ms.

Here's one for you guys to do:

If a given $O(n^3)$ algorithm runs in 12 ms for an input of size 25, and for another input runs in 324 ms, how big was that input, in all likelyhood?

Reasonable vs. Unreasonable Algorithms

One thing we can use order notation for is to decide whether or not an algorithm can be implemented in practice and run in a reasonable amount of time.

In a real general sense, algorithms that run in polynomial time with respect to the input, are considered to be REASONABLE. So, this would include any algorithm that runs in $O(n^k)$ time, where k is some constant. In most everyday problems that get solved, k is never more than 3 or so. While $O(n^3)$ algorithms do run quite slow for larger input sizes, they will still finish in a reasonable amount of time, and there are some common problems that take at least that much time no matter what.

However, there are mathematical functions that are "larger" than polynomials. In particular, exponential functions grow more quickly than polynomials. If an algorithm is exponential, meaning it runs in $O(c^n)$ time where c is some constant, it is considered to be an UNREASONABLE algorithm. Running such an algorithm would simply take too much time for any substantial value of n. (Consider computing 2^{100}...)

Often times, exhaustive search algorithms are UNREASONABLE. In a chess game, one way for a computer player to choose a move is to map out all possible moves by the computer and the opponent, several moves into the future. Then, by judging which move would eventually lead to a better board position, the computer can then make its move. Unfortunately, there are simply too many possible board positions to consider them all. Thus, such an algorithm is unreasonable. Most computer chess programs only search a few possible moves, not all of them. And only consider a few of the opponents responses.