Algorithm Analysis

We have spent most of the semester finding algorithms to solve problems. That invites a question: Suppose we have found a way to solve a problem. Is our solution a good one? And another question: Suppose we know of two ways to solve a problem. Which one is better?

What makes for a "good" algorithm can vary from one situation to the next, but the most common criteria assume that running quickly on large data sets is an important property of "good" algorithms and that, other things being equal, an algorithm that runs faster on very large data sets is better than one that runs slower.

In specific situations you can argue with that assumption, but that is the framework underlying most current algorithm analysis.

Question: What is a good reason to give your employer that you don't care about performance on very large data sets?

A) You only have small data sets

B) You are feeling lazy

C) Your algorithm runs so fast it isn't worth the time to find a faster one

D) Your personal psychic assured you that your algorithm is the best one possible.

We usually try to rank algorithms by the "Orders of Growth" of their runtimes on large data sets. To do that we need some notation:

Suppose f(n) and p(n) are two functions of a variable n. We say

$$f(n) = O(p(n))$$

[read that as "f(n) is Big Oh of p(n)"]

if there is a constant K so that $f(n) <= K*p(n)$ whenever n is very large.

Function p is called the "order of growth" of f.

Here is how to make some sense of this. Suppose f(n) is a quadratic polynomial, such as $f(n) = 25n^2+16n+235$.

First of all, when n is large $f(n) <= 25n^2+16n^2+235n^2=276n^2$,

so $f(n) = O(n^2)$

.

It is easy to show that any quadratic polynomial is $O(n^2)$

In fact, if f is a polynomial of degree k (i.e., k is the highest exponent in f) then f(n) is $O(n^k)$

Now, here is what this has to do with orders of growth. Go back the polynomial $f(n) = 25n^2+16n+235$ and suppose this represents the running time of some algorithm on a data set of size n. To see what happens if we double the size of the data set, consider the ratio

$$f(2n)/f(n) = \frac{25*(4n^2)+16*(2n)+235}{25n^2+16n+235} = \frac{25*4+16*\left(\frac{2}{n}\right)+\frac{235}{n^2}}{25+\frac{16}{n}+\frac{235}{n^2}}$$

When n is large this is approximately 25*4/25, or 4.

Since the order of growth of f is $O(n^2)$, we can get this more simply as $(2n)^2/n^2$, which is 4. Remember that f represents running time on a data set of size n. If we double the size of the data set, we increase the run time by a factor of 4. If we increase the data size by a factor of 10 we increase the runtime by a factor of 100.

Suppose we know that an algorithm has running time $f(n)=O(2^n)$. Note that $2^{n+1}/2^n = 2$: increasing the data size by 1 doubles the computation time. If the algorithm can handle 1000 data values in 1 minute it takes 2 minutes for 1001 values, 4 minutes for 1002 values, and 1024 minutes (17 hours) for 1010 values. If we have a data set much larger than 1000 values we need to find a different algorithm.

When n is large, which is larger, 2^n or n^2?

A) 2^n

B) n^2

When n is large, which is larger, 2^n or n^{1000}?

A) 2^n

B) n^{1000}

Here is one example of estimating the running time of an algorithm.

Remember the SelectionSort algorithm. We sort a list L by making repeated passes through L. On the first pass we look through all of L to find the index of the smallest item, and swap that item with L[0]. On the next pass we look through all of starting at index 1, find the index of the smallest item, and swap that item with L[1]. This continues until L is sorted.

Let n be the size of list L. On the first pass we do n-1 comparisons; on the second n-2 comparisons, on the third n-3, and so forth. Each pass uses one fewer comparison than the previous pass.

Altogether this does (n-1)+(n-2)+(n-3)+...1 comparisons. You probably saw in high school a formula that says these numbers sum to n*(n-1)/2 = $O(n^2)$. If you don't remember that formula just think that you are adding up n-1 numbers that are all less than n, so the sum is no more than n^2. Since half of the numbers are larger than n/2 the sum is at least (n-1)/2*n/2. These two bounds say that the sum is $O(n^2)$.

So the running time of SelectionSort on a list of size n has order of growth $O(n^2)$. If we can sort 1000 items in 1 second we can sort 10,000 items in 100 seconds, or about 1.5 minutes.

Here are the most common orders of growth:

O(log(n)) It doesn't matter which base you use for the logarithm; all logs are proportional

O(n) "Linear time"

O(n^2) "Quadratic time"

O(n^3) "Cubic time"

O(n^k) "Polynomial time with degree k"

O(2^n) "Exponential time"

Exponential algorithms are much worse than any polynomial time algorithms. Unfortunately, there are lots of important practical problems for which the only known solutions are exponential time.