Algorithm Analysis

This is based on Chapter 5 of the text.

John and Mary have each developed new sorting algorithms. They are arguing over whose algorithm is better.

John says "Mine runs in 0.5 seconds."

Mary says "Mine runs in 1 millisecond"

Does this mean Mary's is better?

Not necessarily. John might be sorting 100,000 items and Mary might be sorting 10. Obviously, the size of the problem matters.

John says "Mine sorts 10,000 items in 0.5 seconds."

Mary says "Mine sorts 10,000 items in 1 millisecond."

Does this mean Mary's is better?

Not necessarily. John might be running his on a TRS-80 from 1980; Mary might be running hers on new MacBook Pro.

Obviously, the platform affects running time.

John says "Mine sorts 10,000 items in 0.5 seconds on a new MacBook Pro."

Mary says "Mine sorts 10,000 items in 1 millisecond on a new MacBook Pro."

Does this mean Mary's is better?

Not necessarily.   John might be sorting names taken randomly from the NYC phone book; Strings have a fairly complex comparison method.

Mary might be sorting integers, which can be compared quickly. Mary's data might be almost in order before she starts.

Obviously, the nature of the data and the actual data values matter.

John says "Mine sorts this particular list of random strings in 0.5 seconds."

Mary say "Mine sorts the same list in 1 millisecond."

Does this mean Mary's algorithm is better?

Not necessarily, unless the only thing you are interested in is sorting that particular data set.

Some algorithms are good on particular data sets but bad in general.

For example, BubbleSort, which is not a very good algorithm in general, works very well on data which is nearly sorted.

Some algorithms work very well on small data sets and badly on very large data sets.

So what do we do? We want some method for comparing two algorithms that doesn't depend on a particular machine or a particular set of data.

There are many possible answers to that question -- there are many possible ways that algorithms can be compared.

The most common method is to **compare the number of basic steps** the algorithm takes in the **worst-case** when running on data of **size n**, where **n is extremely large**.

This may not tell you everything you need to know to decide if you should use the algorithm, but at least it provides a standard basis for comparison.

Example 1
Searching for an element in an ArrayList

LinearSearch(a, L) runs by comparing *a* to L[0], to L[1], and so forth. It stops and returns *true* when it finds a, or returns *false* when it gets to the end of the list.

Suppose the list has size n. In the worst-case LinearSearch does n comparisons. The worst case comes when *a* is not in the list.

If L is sorted in increasing order we can do *BinarySearch(a, L*). We maintain a search region -- all of the list between index *LO* and index *HI*. Initially *LO=0* and *HI=L.length-1*.

At each step we compute *MID = (LO+HI)/2* and compare *a* to the entry of *L[MID]*. If *a* is greater then we search the elements between *MID+1* and *HI*; if *a* is lower we search between *LO* and *MID-1*. We eventually get to a search region with only 1 element, and it is either *a* or it isn't.

Note that at each step we divide the search region in half, and we stop when it gets to size 1.

n can be divided in half $\log_2(n)$ times before it gets to 1 (e.g., if n is 16 the divisions have size 8, 4, 2, and 1 $2^4=16$, so $\log_2(16)=4$).

So *BinarySearch(a, L)* takes $\log_2(n)$ comparisons, where n is the size of L.

Which is better when n is really big -- n or $\log_2(n)$??

n	$\log_2(n)$
100	6
1000 (2^{10})	10
1,000,000 (2^{20})	20
1,000,000,000 (2^{30})	30

It looks like $\log_2(n)$ is winning here....

Example 2

Some of you know the BubbleSort algorithm. One way to sort a list is to repeatedly make passes from the first element to the last. For each index i we compare the i^{th} and $(i+1)^{st}$ elements; if they are out of order interchange them. Keep doing these passes until there is no reason to do an interchange, which means the data is completely in order.

How many steps will that take? Note that in the first pass through the list, we end up with the largest element in the right location. After that there is no need to compare the last element of the list with anything else, so we can stop the next pass one element sooner. With n elements in the list we do (n-1) comparisons and at most (n-1) interchanges for the first pass. The next pass we will do (n-2) comparisons; the next pass (n-3) and so forth.

Altogether in the worst case we will do this many comparisons and the same number of interchanges:

(n-1) + (n-2) + (n-3) + ... + 1

It isn't hard to show that this sum comes to n*(n-1)/2.

So we will do at most n(n-1)/2 comparisons and at most the same number of interchanges. Each interchange takes 3 assignment statements. So if we think of comparisons and assignments as both being basic operations, this means

4n(n-1)/2 = 2n(n-1) basic operations in the worst case.

Is that good?

We need some way to categorize the running times of algorithms. The most common way is to look at broad categories that represent how fast the running times grow as the problem size gets larger. The thing that stands out most for the function 2n(n-1) is that it is quadratic. There are two aspects of this.

For one thing, for REALLY large values of n, such as n=1,000,000 2n(n-1) is pretty much the same thing as $2n^2$.

For another, what happens if we increase the size of the list by a factor of k, from n to kn?

The number of basic operations will increase by a factor of $2(kn)^2/2n^2 = k^2$

So increasing the size of the list by a factor of 3 increases the worst-case running time by a factor of 9; increasing the size of the list by a factor of 10 increases the running time by a factor of 100.

We say that BubbleSort is a quadratic time algorithm. Another way to say this is that the running time is $O(n^2)$ (also pronounced "order n^2“ or “Big Oh of n^2”).

In general, we say that an algorithm is O(f(n)) if there is a size N and a constant a so that for $n > N$ the number of basic operations the algorithm does on an input of size n is no more than a*f(n).

BubbleSort is $O(n^2)$ because we showed that it does at most $2n(n-1)$ basic operations and $2n(n-1) < 2n^2$.

Note that looking at orders of growth allows us to make many simplifications:

$5n^3+3n^2+177$ is still $O(n^3)$

The most common orders of growth, in order of increasing badness, are

constant	$O(1)$
logarithmic	$O(\log(n))$
linear	$O(n)$
n*log(n)	$O(n*\log(n))$
polynomial	$O(n^k)$ for some k
exponential	$O(2^n)$

For logarithms, note that all logs are proportional, so it doesn't matter what base you use for the log.