Algorithm Complexity

Classification of Worst Case Algorithm Complexity

Complexity refers to the growth in resources needed by an algorithm when the size of the input increases. *Resources* may be computing time or storage space or both.

Big-Oh notation, **O(f[n])**, identifies an asymptotic upper bound to the computational complexity of an algorithm.

O(1)	instructions executed once (constant time)
O(logN)	as input size N increases, running time increases by logN (logarithmic)
O(N)	instructions executed N times, proportional to size of input (linear)
O(N logN)	time proportional to input size N times logN (slightly more than linear)
O(N^2)	time proportional to size squared (quadratic)
O(N^3)	time proportional to cube of input size (cubic)
O(2^N)	exponential time, non-polynomial (NP)

In the above the function **f[n]** is represented by the argument to the function **O**.

Caution: Algorithm complexity is for the *worst case*; it does not indicate average performance. Very often the worst case is extremely rare for practical problems. Also initialization and non-loop running times may be large regardless of big-oh time.

Little-Oh notation, **o(f[n])**, identifies complexity that is strictly less than **f[n]**. Big-Oh allows the possibility that complexity is exactly equal to f[n], where Little-Oh does not. Little-Oh is rarely used.

Big-Oh Notation as an Approximation Technique

A polynomial has the form:

```
Ax^n + Bx^(n-1) + ... + Yx + Z.
```

For example, a third degree polynomial has the form

```
Ax^3 + Bx^2 + Cx + D
```

Big-oh notation is intended to tell you approximately how much effort an algorithm might require, given the worst case. Because the terms to the right contribute much less than accompanying terms on the left, big-oh notation drops all terms on the right. Further, because the multiplication factor in front of each term contributes a constant amount, it too is dropped. Thus, the complexity of a general polynomial is

```
O(x^n)
```

This approximation is used for three reasons:

1. We are usually concerned about only a rough approximation of how much effort an algorithm takes.
2. We usually change an algorithm only when the choice is significantly much better.
3. Our analysis tools aren't strong enough to tell us much more anyway.

Caution: Depending on the input size, the deleted coefficient, **A**, may have more of an influence than the determining function **f[n]**.

Classification of Lower Bounds of Complexity

Big-Omega notation, **Omega(f[n])**, identifies a lower bound to worst case complexity. Knowing that all inputs will take a minimum amount of time is useful, but is usually too difficult to determine. As well, we do not know how to design silicon circuits to take advantage of lower bounds.

Big-Theta notation, **Theta(f[n])**, means that complexity increases exactly the same as **f[n]**. This is fairly rare, but useful for analysis of families of algorithms.

Complexity Reduction

When a problem can be subdivided into parallel processes, or subdivided into multiple recursive processes, the complexity of the algorithm decreases, most usually from a function of x^n to a function of x^(n-1) * log n.

When input is sorted, as opposed to random, complexity usually decreases from a function of x^n to a function of x^(n-1). However, the sorting process takes at best O(N log N), so this gain is meaningful only when the algorithm using the input is more complex than O(N log N).

Algorithm analysis yields theoretical results, which are meaningful only when the implementation and compilation are maximally efficient. However, algorithm analysis does produce results which cannot be improved upon by any means (implementation can only make the situation worse).

Satisfiability

The Satisfiability Problem for propositional calculus (i.e. Boolean algebra, combinational circuitry) asks: Is there at least one binding for the variables in the statement of the problem which makes the entire problem True?

Satisfiability is the basis for almost all algorithm analysis which distinguishes between polynomial (i.e. tractable) and non-polynomial (i.e. intractable) problems. The complexity of the algorithm which answers the satisfiability problem is currently not known, even though it is about the most simple non-trivial mathematical problem possible. If the best case complexity of that algorithm is non-polynomial (i.e. exponential), which almost everyone believes is the case, then most interesting algorithms in Computer Science are intractable.