Big O notation

For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by $T(n) = 4\ n^2 - 2\ n + 2$. If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms, we could say "$T(n)$ grows at the *order* of n^2" and write:$T(n) = \mathrm{O}(n^2)$.

In mathematics, it is often important to get a handle on the error term of an approximation. For instance, people will write

$$e^x = 1 + x + x^2 / 2 + \mathrm{O}(x^3) \quad \text{for } x \to 0$$

to express the fact that the error is smaller in absolute value than some constant times x^3 if x is close enough to 0.

[redacted]

Here is a list of classes of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. c is some arbitrary constant.

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Note that $O(n^c)$ and $O(c^n)$ are very different. The latter grows much, much faster, no matter how big the constant c is. A[redacted] An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest algorithms known for integer factorization.

[redacted]

[redacted]

Example: If $f(n) = 10 \log(n) + 5 (\log(n))^3 + 7 n + 3 n^2 + 6 n^3$, then $f(n) = O(n^3)$.

One caveat here: the number of summands has to be constant and may not depend on n. This notation can also be used with multiple variables and with other expressions on the right side of the equal sign. The notation:

$$f(n,m) = n^2 + m^3 + O(n+m)$$

represents the statement:

$$\exists C \, \exists N \, \forall \, n,m>N : f(n,m) \leq n^2+m^3+C(n+m)$$

Obviously, this notation is abusing the equality symbol, since it violates the axiom of equality: "things equal to the same thing are equal to each other". To be more formally correct, some people (mostly mathematicians, as opposed to computer scientists) prefer to define $O(g(x))$ as a set-valued function, whose value is all functions that do not grow faster then $g(x)$, and use set membership notation to indicate that a specific function is a member of the set thus defined. Both forms are in common use, but the sloppier equality notation is more common at present.

Another point of sloppiness is that the parameter whose asymptotic behaviour is being examined is not clear. A statement such as $f(x,y) = O(g(x,y))$ requires some additional explanation to make clear what is meant. Still, this problem is rare in practice.

Related notations

In addition to the big O notations, another Landau symbol is used in mathematics: the little o. Informally, $f(x) = o(g(x))$ means that f grows much slower than g and is insignificant in comparison.

Formally, we write $f(x) = o(g(x))$ (for $x \to \infty$) if and only if for every $C>0$ there exists a real number N such that for all $x > N$ we have $|f(x)| < C\ |g(x)|$; if $g(x) \neq 0$, this is equivalent to $\lim_{x\to\infty} f(x)/g(x) = 0$.

Also, if a is some real number, we write $f(x) = o(g(x))$ for $x \to a$ if and only if for every $C>0$ there exists a positive real number d such that for all x with $|x - a| < d$

we have $|f(x)| < C\ |g(x)|$; if $g(x) \neq 0$, this is equivalent to $\lim_{x \to a} f(x)/g(x) = 0$.

Big O is the most commonly-used of five notations for comparing functions:

Notation	Definition	Analogy
$f(n) = O(g(n))$	see above	$\leq$
$f(n) = o(g(n))$	see above	$<$
$f(n) = \Omega(g(n))$	$g(n)=O(f(n))$	$\geq$
$f(n) = \omega(g(n))$	$g(n)=o(f(n))$	$>$
$f(n) = \theta(g(n))$	$f(n)=O(g(n))$ and $g(n)=O(f(n))$	$=$

The notations θ and Ω are often used in computer science; the lowercase o is common in mathematics but rare in computer science. The lowercase ω is rarely used.

A common error is to confuse these by using O when θ is meant. For example, one might say "heapsort is O(n log n)" when the intended meaning was "heapsort is θ(n log n)". Both statements are true, but the latter is a stronger claim.

The notations described here are very useful. They are used for approximating formulas for analysis of algorithms, and for the definitions of terms in complexity theory (e.g. polynomial time).

Source : "http://www.wikipedia.org/w/wiki.phtml?title=Big_O_notation"

Understanding Big O

Introduction

How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we will mostly talk about time complexity (CPU usage).

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

The time required by a function/procedure is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, *).
- one assignment (e.g. x := 0)
- one test (e.g., x = 0)
- one read (of a primitive type: integer, float, character, boolean)
- one write (of a primitive type: integer, float, character, boolean)

Some functions/procedures perform the same number of operations every time they are called. For example, StackSize in the Stack implementation always returns the number of elements currently in the stack or states that the stack is empty, then we say that StackSize takes ***constant time***.

Other functions/ procedures may perform different numbers of operations, depending on the value of a parameter. For example, in the BubbleSort algorithm, the number of elements in the array, determines the number of operations performed by the algorithm. This parameter (number of elements) is called the ***problem size/ input size***.

When we are trying to find the complexity of the function/ procedure/ algorithm/ program, we are ***not*** interested in the ***exact*** number of operations that are being performed. Instead, we are interested in the relation of the ***number of operations*** to the ***problem size***.

Typically, we are usually interested in the ***worst case***: what is the ***maximum*** number of operations that might be performed for a given problem size. For example, inserting an element into an array, we have to move the current element and all of the elements that come after it one place to the right in the array. In the worst case, inserting at the beginning of the array, ***all*** of the elements in the array must be moved. Therefore, in the worst case, the time for insertion is proportional to the number of elements in the array, and we say that the worst-case time for the insertion operation is ***linear*** in the number of elements in the array. For a linear-time algorithm, if the problem size doubles, the number of operations also doubles.

Big-O notation

We express complexity using **big-O notation**.

For a problem of size N:

- a constant-time algorithm is "order 1": O(1)
- a linear-time algorithm is "order N": O(N)
- a quadratic-time algorithm is "order N squared": $O(N^2)$

Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time algorithm will be faster than a linear-time algorithm, which will be faster than a quadratic-time algorithm).

Formal definition:

A function T(N) is O(F(N)) if for some constant c and for values of N greater than some value n_0:

$$T(N) <= c * F(N)$$

The idea is that T(N) is the ***exact*** complexity of a procedure/function/algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e., the actual time/space or whatever for a problem of size N will be no worse than F(N)).

In practice, we want the smallest F(N) -- the ***least*** upper bound on the actual complexity. For example, consider:

$$T(N) = 3 * N^2 + 5.$$

We can show that T(N) is $O(N^2)$ by choosing $c = 4$ and $n_0 = 2$.

This is because for all values of N greater than 2:

$$3 * N^2 + 5 <= 4 * N^2$$

T(N) is **not** O(N), because whatever constant c and value n_0 you choose, There is always a value of $N > n_0$ such that $(3 * N^2 + 5) > (c * N)$

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).

If-Then-Else

```
if (cond) then
   block 1 (sequence of statements)
else
   block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

max(time(block 1), time(block 2))

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

Loops

```
for I in 1 .. N loop
   sequence of statements
end loop;
```

The loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Nested loops

```
for I in 1 .. N loop
   for J in 1 .. M loop
      sequence of statements
   end loop;
end loop;
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).

In a common special case where the stopping condition of the inner loop is `J < N` instead of `J < M` (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Statements with function/ procedure calls

When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/ procedure *f* takes constant time, and that function/procedure *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

f(k) has O(1)
g(k) has O(k)

When a loop is involved, the same rule applies. For example:

```
for J in 1 .. N loop
     g(J);
end loop;
```

has complexity (N^2). The loop executes N times and each function/procedure call g(N) is complexity O(N).