The Complexity of Algorithms

William Stallings

A central issue in assessing the practicality of an algorithm is the relative amount of time it takes to execute the algorithm. Typically, one cannot be sure that one has found the most efficient algorithm for a particular function. The most that one can say is that for a particular algorithm, the level of effort for execution is of a particular order of magnitude. One can then compare that order of magnitude to the speed of current or predicted processors to determine the level of practicality of a particular algorithm.

A common measure of the efficiency of an algorithm is its time complexity. We define the **time complexity** of an algorithm to be f(n) if, for all n and all inputs of length n, the execution of the algorithm takes at most f(n) steps. Thus, for a given size of input and a given processor speed, the time complexity is an upper bound on the execution time.

There are several ambiguities here. First, the definition of a step is not precise. A step could be a single operation of a Turing machine, a single processor machine instruction, a single high-level language machine instruction, and so on. However, these various definitions of step should all be related by simple multiplicative constants. For very large values of n, these constants are not important. What is important is how fast the relative execution time is growing.

A second issue is that, generally speaking, we cannot pin down an exact formula for f(n). We can only approximate it. But again, we are primarily interested in the rate of change of f(n) as n becomes very large.

There is a standard mathematical notation, known as the "big-O" notation, for characterizing the time complexity of algorithms that is useful in this context. The definition is as follows: f(n) = O(g(n)) if and only if there exist two numbers a and M such that

$$|\mathrm{f}(n)| \leq a \times |\mathrm{g}(n)|, \qquad n \geq M \qquad \textbf{(1)}$$

An example helps clarify the use of this notation. Suppose we wish to evaluate a general polynomial of the form:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Consider the following simple-minded algorithm from [POHL81]:

```
algorithm P1;
    n, i, j: integer; x, polyval: real;
    a, S: array [0..100] of real;
    begin
        read(x, n);
        for i := 0 upto n do
        begin
            S[i] := 1; read(a[i]);
            for j := 1 upto i do S[i] := x × S[i];
            S[i] := a[i] × S[i]
        end;
        polyval := 0;
        for i := 0 upto n do polyval := polyval + S[i];
        write ('value at', x, 'is', polyval)
    end.
```

In this algorithm, each subexpression is evaluated separately. Each S[i] requires (i + 1) multiplications: i multiplications to compute S[i] and one to multiply by a[i]. Computing all n terms requires

$$\sum_{i=0}^{n} (i + 1) = \frac{(n + 2)(n + 1)}{2}$$

multiplications. There are also (n + 1) additions, which we can ignore relative to the much larger number of multiplications. Thus, the time complexity of this algorithm is $f(n) = (n + 2)(n + 1)/2$. We now show that $f(n) = O(n^2)$. From the definition of Equation (1), we want to show that for a

= 1 and $M = 4$, the relationship holds for $g(n) = n^2$. We do this by induction on n. The relationship holds for $n = 4$ because $(4 + 2)(4 + 1)/2 = 15 < 4^2 = 16$. Now assume that it holds for all values of n up to k [i.e., $(k + 2)(k + 1)/2 < k^2$]. Then, with $n = k + 1$:

$$\frac{(n+2)(n+1)}{2} = \frac{(k+3)(k+2)}{2}$$

$$= \frac{(k+2)(k+1)}{2} + k + 2$$

$$\leq k^2 + k + 2$$

$$\leq k^2 + 2k + 1 = (k+1)^2 = n^2$$

Therefore, the result is true for $n = k + 1$.

In general, the big-O notation makes use of the term that grows the fastest. For example:

1. $O[ax^7 + 3x^3 + \sin(x)] = O(ax^7) = O(x^7)$

2. $O(e^n + an^{10}) = O(e^n)$

3. $O(n! + n^{50}) = O(n!)$

There is much more to the big-O notation, with fascinating ramifications. For the interested reader, two of the best accounts are in [GRAH94] and [KNUT97].

An algorithm with an input of size n is said to be:

1. **Linear:** if the running time is $O(n)$
2. **Polynomial:** if the running time is $O(n^t)$ for some constant t
3. **Exponential:** if the running time is $O(t^{h(n)})$ for some constant t and polynomial $h(n)$

Generally, a problem that can be solved in polynomial time is considered feasible, whereas anything larger than polynomial time, especially exponential time, is considered infeasible. But you must be careful with these terms. First, if the size of the input is small enough, even very complex algorithms become feasible. Suppose, for example, that you have a system that can execute 10^{12} operations per unit time. Table 1 shows the size of input that can be handled in one time unit for algorithms of various complexities. For algorithms of exponential or factorial time, only very small inputs can be accommodated.

The second thing to be careful about is the way in which the input is characterized. For example, the complexity of cryptanalysis of an encryption algorithm can be characterized equally well in terms of the number of possible keys or the length of the key. For the Advanced Encryption Standard (AES), for example, the number of possible keys is 2^{128}, and the length of the key is 128 bits. If we consider a single encryption to be a "step" and the number of possible keys to be $N = 2^n$, then the time complexity of the algorithm is linear in terms of the number of keys [O(N)] but exponential in terms of the length of the key [O(2^n)].

References

GRAH94 Graham, R.; Knuth, D.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*. Reading, MA: Addison-Wesley, 1994.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997.

POHL81 Pohl, I., and Shaw, A. *The Nature of Computation: An Introduction to Computer Science*. Rockville, MD: Computer Science Press, 1981.

Table 1 Level of Effort for Various Levels of Complexity

Complexity	Size of Input	Operations
$\log_2 n$	$2^{10^{12}} = 10^{3\times10^{11}}$	10^{12}
n	10^{12}	10^{12}
n^2	10^6	10^{12}
n^6	10^2	10^{12}
2^n	39	10^{12}
$n!$	15	10^{12}