Big-O Notation

When trying to characterize an algorithm’s efficiency in term of execution time, independent of any particular program or computer, it is important to quantify the number of operations of steps that the algorithm will require. If each of these steps is considered to be basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on the appropriate basic unit of computation can be complicated problem and will depend on how the algorithm is implemented.

A good basic unit of computation for comparing the summation algorithms shown earlier might be to count the number of assignment statements performed to compute the sum. In the function “sumOfN”, the number of assignment statements is 1 (theSum = 0) plus the value of n(the number of times we perform theSum = theSum + i). We can denote this by a function, call it T, where T(n) = 1 + n. The parameter n is often referred to as the “size of the problem,” and can read this as “T(n) is the time it takes to solve a problem of size n, namely 1+n steps.”

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. We can then say the sum of the first 100,000 integers is bigger instance of the summation problem thn the sum of the first 1,000. Because of this, it might seem reasonable that the time require to solve the larger case would be greater than for the smaller case. Our goal then is to show how the algorithm time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out the exact number of operations is not important as determining the most dominant part of the ***T****(n)* function*.* In other words, as the problem gets larger, some portion of the *T(n)* function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The **order of magnitude** function describes the part of *T(n)* that increases the fastest as the value of *n* increases. Order of magnitude is often called **Big-O** notation (for “order”) and written as *O(f(n))*. It provides a useful approximation to the actual number of steps in computation. The function *f(n)* provides a simpler representation of the dominant part of the original *T(n)*.

In the above example, *T(n) = 1 + n*. As *n* gets large, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for *T(n)*, then we can drop the 1 and simply say that the running time is *O(n)*. It is important to note that the 1 is certainly significant for *T(n)*. However, as *n* gets large, our approximation will be just as accurate without it.

As another example, suppose that for some algorithm, the exact number of steps is 𝑇(𝑛)=5𝑛2+27𝑛+1005T(n)=5n2+27n+1005. When *n* is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as *n* gets larger, the 𝑛2n2 term becomes the most important. In fact, when *n* is really large, the other two terms become insignificant in the role that they play in determining the final result. Again, to approximate 𝑇(𝑛)T(n) as *n* gets large, we can ignore the other terms and focus on 5𝑛25n2. In addition, the coefficient 55 becomes insignificant as *n* gets large. We would say then that the function 𝑇(𝑛)T(n) has an order of magnitude 𝑓(𝑛)=𝑛2f(n)=n2, or simply that it is 𝑂(𝑛2)O(n2).

Although we do not see this in the summation example, sometimes the performance of an algorithm depends on the exact values of the data rather than simply the size of the problem. For these kinds of algorithms we need to characterize their performance in terms of best case, **worst case**, or **average case** performance. The worst case performance refers to a particular data set where the algorithm performs especially poorly. Whereas a different data set for the exact same algorithm might have extraordinarily good performance. However, in most cases the algorithm performs somewhere in between these two extremes (average case). It is important for a computer scientist to understand these distinctions so they are not misled by one particular case.