Introduction to Computational Complexity

Michael Levet

May 19, 2013

This tutorial will cover basic algorithm analysis, specifically the time complexity of algorithms. The tools used include Big-O, Big-Omega, and Big-Theta. This tutorial will also discuss some of the mathematical properties of Big-O, Big-Omega, and Big-Theta.

Big-O and Big-Omega

First, let's define Big-O and Big-Theta. A function f(n) is O(g(n)) if and only if $\exists C, k \in \mathbb{Z}^+$ such that $|f(x)| \leq C * |g(x)|$, $\forall x \geq k$. In other words, Big-O is an upper bound. Some examples include:
$f(n) = 3n$ is $O(n)$

$f(n) = 2^n + n^2$ is $O(2^n)$, as 2^n is the dominant term. Looking at this more mathematically, $2^n + n^2 \leq 2 * 2^n$. So let $C = 2$ and $k = 4$ to satisfy the definition of Big-O.

Similarly, a function $f(n)$ is $\Omega(g(n))$ if and only if $\exists C, k \in \mathbb{Z}^+$ such that $|f(x)| \geq C * |g(x)|$, $\forall x \geq k$. In other words, Big-Omega is a lower bound. Some examples include:
$f(n) = 3n$ is $\Omega(log(n))$
$f(n) = 2^n + n^2$ is $\Omega(n)$.

From the definitions of Big-O and Big-Omega, it can be concluded that f(n) is O(g(n)) if and only if g(n) is Omega(f(n)). More exactingly, if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$, then $f(n)$ and $g(n)$ belong to the same complexity class. The same can be said of Big-Omega. Thus, Big-O and Big-Omega are anti-symmetric relations.

With a little more work, it is easy to show that Big-O and Big-Omega are partial-order relations. A partial order relation is reflexive, anti-symmetric, and transitive. Consider reflexivity: $f(n)$ is $O(f(n))$, and $f(n)$ is $\Omega(f(n))$.

Big-O and Big-Omega are also transitive. Consider if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, this shows that $|f(n)| \leq C_1|g(n)| \leq C_2|h(n)|$. Thus, $f(n)$ is $O(h(n))$. Replacing Big-O with Big-Omega, and inverting the $\leq$ operator to the $\geq$ operator shows that Big-Omega is transitive as well.

Now let's develop a little more intuition for the information provided by Big-O and Big-Omega. Consider the following statements about an algorithm:
It is $O(n^3)$
It is $O(n!)$
It is $\Omega(n^2)$
It is $\Omega(n)$

Here, knowing the algorithm is $O(n^3)$ provides significantly more information than that the algorithm is $O(n!)$. Similarly, knowing that an algorithm is $\Omega(n^2)$ provides more information than that the algorithm is $\Omega(n)$.

Big-Theta

While Big-O and Big-Theta are the upper and lower bounds, respectively, Big-Theta is the exact bound. A function $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$. While Big-O and Big-Omega are partial order relations, Big-Theta is an equivalence relation. That means that Big-Theta is reflexive, symmetric, and transitive.

Since Big-O and Big-Omega are both reflexive and transitive, Big-Theta is reflexive and transitive as well. To prove symmetry, consider $f(n)$ is $\Theta(g(n))$. This implies that $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. By anti-symmetry of Big-O, it can be inferred that $g(n)$ is $\Omega(f(n))$. Similarly, by anti-symmetry of Big-Omega, $g(n)$ is $O(f(n))$. Thus, $g(n)$ is $\Theta(f(n))$. So Big-Theta is symmetric; and thus, it is an equivalence relation.

As Big-O, Big-Omega, and Big-Theta all describe the long-term asymptotic behavior of a function, the limit test is useful in determining the bounds for a function. Consider the following theorem. Let $L = lim_{n\to\infty}\frac{f(n)}{g(n)}$.

If $L = 0$, the $g(n)$ strictly dominates $f(n)$. Thus, it can be concluded that $f(n)$ is $O(g(n))$.

If $L = c$, where c is finite, then $f(n)$ and $g(n)$ grow at the same asymptotic rate. Thus, $f(n)$ is $Theta(g(n))$.

Finally, if $L = \infty$, then the $f(n)$ term dominates the $g(n)$ term, so $g(n)$ is a lower bound for $f(n)$. Thus, $f(n)$ is $\Omega(g(n))$.

This theorem is incredibly useful in determining the complexities of functions, especially when the dominant term isn't clear. If the dominant term is clear, then this theorem implies that the function is Theta(dominant term). So if $f(n) = 3n^2 + 2n + 1$, then $f(n)$ is $Theta(n^2)$.

Let's look at an example. Let $f(n) = n^4 + n^2 log(n) + 3^n$, and $g(n) = 8n^6$. The $lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty$, so $f(n)$ is $\Omega(g(n))$. Looking at the complexity of $f(n)$, it is easy to see that 3^n is the dominant term, so $f(n)$ is $\Theta(3^n)$. The $lim_{n\to\infty}\frac{f(n)}{3^n} = 1$, which confirms the intuition.

Deriving The Runtime Function of an Algorithm

While the previous sections covered more of the theory behind computational complexity, this section will provide some application. Analyzing an algorithm relies on a set of rules for deriving T(n), the runtime function. While these rules may vary slightly, a good rule of thumb is that input, output, use of operators (mathematical, logical, bitwise, etc.), return statements, and array references all take Theta(1) time, so just treat them as 1 unit when constructing T(n). In most cases, these are language constructs, and really don't have a long-term impact on the algorithm implementation.

When evaluating a loop or conditional block, always evaluate the worst case. In other words, assume that the loop runs the maximal number of times. On each iteration, the condition counts is evaluated. So if the loop runs n times, then the condition is evaluated n times.

Let's look at an example:

```
for i = 0, i < array.length, i = i + 1
        array[i] = 0
end for
```

Let $n = array.length$.

Let's start with what we know:

- $i = 0$ is an assignment, so count that as a constant 1.
- $i < array.length$ is a constant 2 (1 for array.length and 1 for the comparison), and is evaluated n times, so count it as $2n$.
- $i = i + 1$ is composed of a mathematical operation and an assignment, and is evaluated n times. So its runtime complexity is 2n.
- Finally, $array[i]$ =0 is an assignment and an array reference, so an individual operation is 2 units. It is evaluated n times, so its complexity is $2n$.

Thus, $T(n) = 6n + 1$. Thus, it is easy to see that this algorithm is $\Theta(n)$. It can be proven by taking $lim_{n\to\infty}\frac{T(n)}{n} = 6 + \frac{1}{\infty} = 6 + 0 = 6$. Thus, as the limit converges to a finite constant, $T(n)$ is $\Theta(n)$.

Let's look at one more example- the binary search. The time intervals are marked with comments.

```
function binarySearch(array arr, target)
    low = 0 //1
    high = arr.length - 1  //2

    while(low <= high) //1
        mid = (low + high)/2  //assignment is 1, and 2 math statements, so 3

        if(arr[mid] == target) //array[mid] is 1, comparison is 1, so 2
            return mid  //1

        else if(arr[mid] > target) //array[mid] is 1, comparison is 1, so 2
            low = mid + 1 //math is 1, assignment is 1, so 2

        else
            hi = mid - 1 //2
    end while

    return -1 //1
```

So $T(n) = 5 + cost(loop)$. So if the loop never executes, then the condition is still checked, hence why the constant term is 5 rather than 4.

On one iteration of the loop, the worst case cost is 10. The first iteration of the loop evaluates the entire array. It then partitions the array in half on subsequent iterations. Thus, it takes $log(n + 1)$ iterations to find the element or determine it doesn't exist in the loop. Thus, $T(n) = 5 + 10log(n + 1)$. Thus, $T(n)$ is $\Theta(log(n))$, as the logarithm is the dominant term.

Conclusion

Algorithm analysis gets more involved, with more advanced and complex algorithms. Often times, recurrence relations are used to determine runtime complexities. This tutorial is merely to introduce basic algorithm analysis techniques, rather than delve into more complex approaches. I hope you have found this tutorial helpful.