CSC 325 Adv Data Structures

Lecture 13

Omega, Theta, Oh… my!

Recap

By now you should be well aware of what asymptotic time complexity is (you know, that Big Oh stuff?). There is the simple, empirically verified understanding of it, for example this code:

```
int i = 0;
float[] array = { /* insert array contents */ };
while (i < array.length) {
  System.out.println(array[i]);
  i++;
}
```

Is obviously linear (i.e. O(n) where n in this case is the size of the array), whereas this code:

```
int i = 0;
float[] array = { /* insert array contents */ };
if (array.length >= 10) {
  while (i < 10) {
    System.out.println(array[i]);
    i++;
  }
}
```

Is obviously constant (i.e. O(1)). You should be able to easily see the time complexity of both of these coding examples without putting too much thought into it (if not, I strongly advise you to revisit your 220 notes on this subject).

There are a few things to recall about this. One is that, just because you see a loop that does NOT mean that the time complexity is linear. It could literally be anything and loops are just where we focus our attention when analyzing the time complexity (and recursion too for that matter). Second, to realize the time complexity of a loop, you normally examine the following:

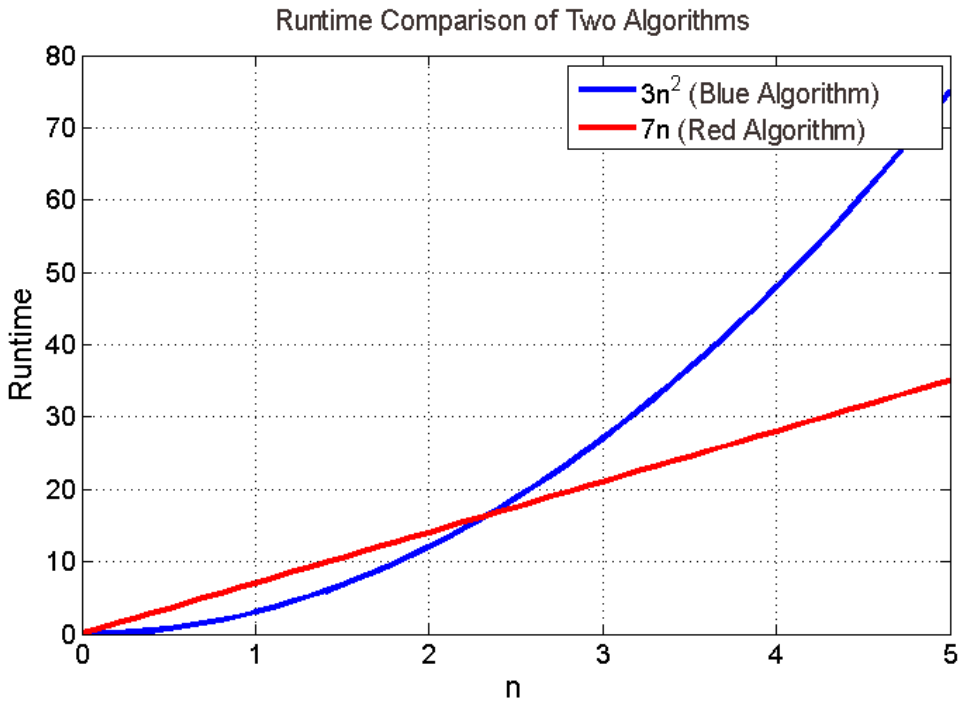
1. Identify the loop variable
2. See what value the loop variable starts at
3. See what value the loop variable ends at
4. See how the loop variable changes with each iteration

In the first code example, the loop variable is i, it starts at 0 and ends at the length of the array, and it changes by +1 each iteration. This means it goes from 0 -> array length, one unit at a time. Which is clearly linear. The second code example uses i again, it starts at 0 but ends at 10, and increase by +1 each time. This means whether the array contains 10 elements or 10 million elements, the number of iterations is unaffected. Now you could say that *sometimes* (when the array contains less than 10 elements), the code doesn't hardly do any

work at all. So you might be tempted to say this code *sometimes* does less work. However, recall that the definition of Big Oh is actually an upper bound. It describes how much work is done in the worst case. What is the most amount of work that could be done. To place an upper bound on the maximum amount of work is to find the Big Oh limit of the code (or algorithm). To be fair, in either case this code either does practically no work at all, or ten units of work (i.e. ten iterations), both of which is described by using the constant value 1 and saying this is O(1).

In CSC 220, we talked about time complexity and Big Oh as if they were synonyms. This might lead you to believe that Big Oh (i.e. an upper bound) is the only way we measure time complexity. Actually, far from it! I mean, why not create a lower bound? Or just label the time complexity exactly. In other words, if Big Oh can be described by less than or equal to, then why not have a less than, greater than, greater than or equal to, and finally an equal to. But before we can go any further, let's talk about the other way of thinking about time complexity. Let's talk about the math behind it.

Let's look at two hypothetical algorithms. The actual code behind them is irrelevant for this discussion, so let's just say that after analysis, this is their relative T(n) values:



As you can see from the legend, the Blue Algorithm has a $T(n) = 3n^2$ and the Red Algorithm has a $T(n) = 7n$. When n is equal to 1 or 2, the blue algorithm is faster (that is it has a lower runtime cost). However, for all values of n equal to or larger than 3, the red algorithm is faster. We say that any algorithm, labelled as f(n), "grows faster" than another algorithm, labelled as g(n) if there exists a point n_0 such that for all values of n equal to or greater than n_0, $f(n) > g(n)$. In the case where f(n) represents T(n) of our blue algorithm and g(n) represents T(n) of our red algorithm, this can be shown by setting n_0 to 3. For all values of n greater than or equal to 3, we can clearly see from our graph above that the blue algorithm grows faster with respect to runtime. In fact, it grows much faster!

Notice how the coefficients don't matter to the definition of "grows faster than". In fact, the coefficients only serve to increase or decrease the value of n_0 but they don't change the fact that if there exists an n_0 such that the statement above is true, then we can say definitively that f(n) grows faster than g(n). Let's say, for example, that the blue algorithm had a constant factor of 1 (so it was 1 * n^2 or just n^2). Let's suppose further that the red algorithm had a constant factor of 50 (so it was 50n). When comparing n^2 to 50n, we simply pick a value for n_0 of 51 (since at 50 they are both equal and at 51, n^2 takes more time, i.e. 51*51 vs 50*51). Now we can see that with n_0 being 51, that n^2 still grows faster than 50n. So, the constant factors (i.e. coefficients to the dominant term) really just modify our choice of n_0 but they don't modify the fact that if n_0 exists, and if all values of n greater than or equal to n_0 makes f(n) > g(n), then f(n) does indeed grow faster than g(n).

Let's state this a different way. Given f(n) and g(n) which represent T(n) computed for two different algorithms, the following shows their relationship:

Big Oh

if c*f(n) <= d*g(n) for all n >= n_0 (where c and d are constants), then we say f(n) grows no faster than g(n). We could also say that g(n) provides an upper bound for f(n) or, equivalently, that f(n) provides a lower bound for g(n).

This provides motivation for our time complexities (which cover more than just Big Oh). In fact, we even have special names and symbols for the various types of bounds we can place on an algorithm. Given an algorithm whose total time cost is represented by T(n), the following shows the bounds we can use:

```
Big Oh:        f(n) is O(g(n)) if c*f(n) <= d*g(n) for all n >= n0
Little Oh:     f(n) is o(g(n)) if c*f(n) <  d*g(n) for all n >= n0
Big Omega:     f(n) is Ω(g(n)) if c*f(n) >= d*g(n) for all n >= n0
Little Omega:  f(n) is ω(g(n)) if c*f(n) >  d*g(n) for all n >= n0
Theta:         f(n) is θ(g(n)) if c*f(n) == d*g(n) for all n >= n0
```

As an example, if T(n) is O(n), then T(n) <= c*n which means that T(n) grows no faster than n multiplied by a constant term. The choice of c is up to you. Another way to look at this is that if there exists a value for c such that T(n) <= c*n for all n >= n_0, then T(n) is O(n). The coefficient that goes with the dominant term in T(n) is irrelevant since we can also choose a higher value for c to compensate. If all possible values for c make T(n) <= c*n true for all n >= n_0, then T(n) is O(n). That would be the case if T(n) grows slower than a constant times n, such is the case if T(n) is, say, logarithmic.

Another way to look at this is the following.

Name	Symbol	Comparison	Bound	Case	Growth
Big Oh	O(g(n))	c*f(n) <= d*g(n)	Upper bound	Worst case	f(n) grows asymptotically no faster than g(n)
Little Oh	o(g(n))	c*f(n) < d*g(n)	Strict upper bound		
Big Omega	Ω(g(n))	c*f(n) >= d*g(n)	Lower bound	Best case	f(n) grows asymptotically no slower than g(n)
Little Omega	ω(g(n))	c*f(n) > d*g(n)	Strict lower bound		
Theta	θ(g(n))	c*f(n) == d*g(n)	Tight bound	Average case	f(n) grows asymptotically at the same rate as g(n)

We actually don't even need two different algorithms to use these time complexities. If we only have f(n) we can pick our g(n) that makes the statement true. For example:

`f(n) is O(n) if f(n) <= d*n`

For example, if f(n) represents an algorithm with a time of 7*n, then choosing a value of 7 for d makes this:

`7*n <= 7*n`

which is a true statement and choosing our n_0 is quite trivial. We could have also chosen any other value for d as long as its greater than or equal to 7. For example, the following works as well:

`7*n <= 8*n`

We could also say that.

`f(n) is o(n2) since 7*n < n2   (where d = 1 and n0 is 8)`

`f(n) is θ(n) since 7*n == 7*n  (where d = 7)`

`f(n) is Ω(n) since 7*n >= 7*n (where d = 7 or any value less than 7)`

`f(n) is ω(1) since 7*n > 1 (where d = 1)`

By choosing the right value for g(n), we can analyze f(n) for a given algorithm. Here are some choices for g(n) and the corresponding complexity class name. Note that this is **not** an exhaustive list. There are other combinations we could make.

Class	**Notation**
Constant	$O(1)$, $\Omega(1)$, $\omega(1)$, $\theta(1)$ *note that o(1) is not possible*
Logarithmic	$O(\lg n)$, $o(\lg n)$, $\Omega(\lg n)$, $\omega(\lg n)$, $\theta(\lg n)$
Linear	$O(n)$, $o(n)$, $\Omega(n)$, $\omega(n)$, $\theta(n)$
Linearithmic	$O(n \lg n)$, $o(n \lg n)$, $\Omega(n \lg n)$, $\omega(n \lg n)$, $\theta(n \lg n)$
Quadratic	$O(n^2)$, $o(n^2)$, $\Omega(n^2)$, $\omega(n^2)$, $\theta(n^2)$
Cubic	$O(n^3)$, $o(n^3)$, $\Omega(n^3)$, $\omega(n^3)$, $\theta(n^3)$
Polynomial	includes n^2, n^3, n^4, n^5, etc where exponent is a constant
Exponential	includes 2^n, 3^n, 4^n, 5^n, etc where base is a constant

Time Complexity Examples:

Given an algorithm where $f(n) = 3n^2 + 4n + 3$, we could say the following:

- The algorithm is $O(n^2)$ since it grows no faster than n^2 multiplied by a constant (for some choice of constant, in this case anything greater than or equal to 3 as this is the coefficient of the dominant term).
- The algorithm is $o(n^3)$ since it grows strictly slower than n^3 multiplied by a constant (for all choices of constant, there will always be an n_0 where this is true for all $n >= n_0$).
- The algorithm is $\Omega(n^2)$ since it grows no slower than n^2 multiplied by a constant (for some choice of constant, in this case anything less than or equal to 3).
- The algorithm is $\omega(n)$ since it grows strictly faster than n multiplied by a constant (for all choices of constant).

- Finally, it is $\theta(n^2)$ since it is both upper and lower bounded by n^2 multiplied by a constant (for some choice of constant, in this case 3).

Given an algorithm, where f(n) = 7n + 4, we could say the following:

- The algorithm is O(n) since it grows no faster than n multiplied by a constant (for some choice of constant, in this case anything greater than or equal to 7 as this is the coefficient of the dominant term).
- The algorithm is o(n lg n) since it grows strictly slower than n * lg(n) multiplied by a constant (for all choices of constant, there will always be an n_0 where this is true for all n >= n_0).
- The algorithm is Ω(n) since it grows no slower than n multiplied by a constant (for some choice of constant, in this case anything less than or equal to 7).
- The algorithm is ω(lg n) since it grows strictly faster than lg(n) multiplied by a constant (for all choices of constant).
- Finally, it is θ(n) since it is both upper and lower bounded by n^2 multiplied by a constant (for some choice of constant, in this case 7).

Here are some more examples:

f(n)	Big Oh	Little Oh	Big Omega	Little Omega	Theta
n	O(n)	o(n^2)	Ω(n)	ω(lg n)	θ(n)
45*lg n	O(lg n)	o(n)	Ω(lg n)	ω(1)	θ(lg n)
2n + 3m	O(n)	o(n*m)	Ω(m)	ω(1)	θ(n + m)

Even though Big Oh and Big Omega form upper and lower bounds, respectively, we still try to bound things as tight as we can. So often times, if we know the time complexity exactly, Big Oh and Big Omega will be the same and will essentially tell us what Theta is. When we can't find a tight upper bound, then Big Oh will differ from Theta. When we can't find a tight lower bound, then Big Omega will differ from Theta. Note that Little Oh and Little Omega are strict bounds and so they will never be the same as Theta, however if we can't find a tight upper bound, then Big Oh and Little Oh may be the same. If we can't find a tight lower bound, then Big Omega and Little Omega may be the same. In the end, I think comparing each of these to their logical operator (i.e. <=, <, >=, >, and ==) makes the most sense.