O vs Θ

Introduction

The notations O and Θ that we use to describe loosely the growth rate of certain expressions are part of a family of notations developed initially for use in (analytic) number theory and mathematical analysis. For our purposes though, we are only concerned with their use in describing the time-complexity of algorithms (certainly the area in which their use is now most widespread).

An excellent, though perhaps overly-detailed reference for this material is the wikipedia page on O-notation and, in particular, the section on related notations and following material.

Throughout this note you can assume that n stands for a positive integer, and that all the expressions we are concerned with have non-negative values, at least for sufficiently large values of n (otherwise, add some absolute value signs wherever they might be needed).

Review of formal definitions

Let $f(n)$ and $g(n)$ stand for two expressions that have positive values depending on some integer n. Usually, these will be mathematical expressions like $g(n) = n^3$, but in some applications $f(n)$ in particular will describe the run-time of some algorithm – so will be implicitly defined in terms of some sort of count of basic operations. That's all very vague (what exactly is a basic operation?), but fortunately the O and Θ notations happen to work in such a way that the precise details don't matter.

We say "f is big-oh of g" or "f of n is big-oh of g of n" and write:

$$f = O(g) \quad \text{or} \quad f(n) = O(g(n))$$

if there is some positive constant A such that for all sufficiently large n,

$$f(n) \leqslant A \times g(n).$$

Aside: The use of the equals sign in $f(n) = O(g(n))$ is what's called an *abuse of notation* since we're not actually saying $f(n)$ is equal to something else, but rather that the expression has some characteristic. But, it is widely used and very convenient for various reasons, so we live with it.

We say "f is big-theta of g" or "f of n is big-theta of g of n" and write:

$$f = \Theta(g) \quad \text{or} \quad f(n) = \Theta(g(n))$$

if there are positive constants A and B such that for all sufficiently large n,

$$B \times g(n) \leqslant f(n) \leqslant A \times g(n).$$

Intuitive meaning and use

We use this type of notation most commonly when f is something quite complicated (or perhaps not exactly known) and g is something simple. so

$$5 + 2n + 3n^2 = \Theta(n^2)$$

is true and looks natural. On the other hand,

$$n^2 = \Theta(5 + 2n + 3n^2)$$

is also true, but looks weird.

The point of both notations is to describe (or bound) the growth rate of $f(n)$ in terms of something familiar (like n^2 or $n \log n$).

Specifically, $f = O(g)$ says we can stretch g by some constant factor and wind up above f (eventually - but from then on always). So f grows no faster than g (up to constant factors) but *might* grow more slowly. On the other hand, $f = \Theta(g)$ says that there are two factors we can stretch g by that squeeze f between them – in effect, ignoring constant factors, f and g grow at the same rate.

Differences

If I'm trying to sell you an algorithm and I say "the running time of this algorithm is $O(n^2)$" then I'm saying that the run time is bounded by some constant multiple of the input size squared *under all circumstances*. Since n^3 is bigger than n^2 it would also be true for me to say "the running time of this algorithm is $O(n^3)$" but then you would be less likely to buy – because you imagine that it might be slower. So, when we make O statements we try to make the expression simple, but as small as possible.

What if I say "the running time of this algorithm is $\Theta(n^2)$"? Then I'm making a statement that *for all inputs* the run time is bounded both above and below by some (different) constant multiples of n^2. In other words, the algorithm is quite rigid – its behaviour is relatively unaffected by any special characteristics of the input.

Consider the following two Java methods. The first takes an `int[]` as input and returns the index in the array where the maximum value occurs. The second takes an `int[]` and an `int` as input and returns an index in the array where the integer value occurs (or -1 if it does not occur.)

```
public static int indexOfMax(int[] a) {
  int result = 0;
  int maxValue = a[0];
  for(int i = 1; i < a.length; i++) {
    if (a[i] > maxValue) { result = i; maxValue = a[i]; }
  }
  return result;
}
```

```
public static int search(int[] a, int v) {
  for(int i = 0; i < a.length; i++) {
    if (a[i] == v) { return i; }
  }
  return -1;
}
```

> **Aside:** The `indexOfMax` method would fail (with an index out of bounds exception) if we passed an array of length 0. Is that appropriate? My opinion is, probably yes – if you're looking for the index of the maximum value in an array it feels like an exception if there are no places to look! On the other hand `search` returns -1 for "not found" in that case. That's appropriate too – if you're searching for something in an empty box, it's perfectly appropriate to say that it's not there.

The `indexOfMax` method *always* looks at every element of the array. It has to, because if it skipped any, it might miss exactly the one where the maximum occurs! So, regardless of the input array, the time taken will be proportional to the length (n) of the array, and its performance is $\Theta(n)$.

On the other hand, the `search` method *sometimes* looks at every element of the array. In particular it has to if the value being searched for doesn't occur because if it skipped any places, it might miss exactly the place where the value occurs. **But**, it can get lucky – perhaps the value being searched for occurs in the very first place it looks. So, sometimes it takes time proportional to n but sometimes it takes only constant time (and it could take anything in between). All we can say is that its performance is $O(n)$. Now actually, this is *better* than $\Theta(n)$ (even though less precise) because it encapsulates the same upper bound, but doesn't imply a lower bound.

In summary, we get Θ bounds when the difference between the work we *must* do and the work we *might* do is a constant factor, but O bounds when the maximum time required for the work we *might* have to do (the upper bound) is much larger (more than a constant factor) than the work we *must* do.

Question: Suppose we knew in advance that we wanted to do these tasks on *sorted* arrays (smallest to largest). Would that change the estimates above? Would some other algorithms be more appropriate and what would their complexities be?

Problems

1. Let $f(n) = 5 + 2n + 3n^2$ and $g(n) = n^2$. Confirm that $f = \Theta(g)$ and $g = \Theta(f)$ are both true.
2. True or false:
   - $n^2 = O(3^n)$
   - $3^n = O(n^2)$
   - $n^2 = \Theta(3^n)$
   - $3^n = \Theta(n^2)$
   - $3^n + n^2 = O(3^n)$
   - $3^n + n^2 = O(n^2)$
   - $3^n + n^2 = \Theta(3^n)$
   - $3^n + n^2 = \Theta(n^2)$
   - $2^n = O(3^n)$
   - $2^n = \Theta(3^n)$
   - $n = O(n \log n)$
   - $n = \Theta(n \log n)$
3. Is it true in general that if $f = \Theta(g)$ then $g = \Theta(f)$? If not, give a counterexample. If so, explain why.
4. Both selection sort and insertion sort have run times bounded above by (a constant multiple of) n^2 and so are $O(n^2)$ algorithms. Do both have $\Theta(n^2)$ run times? Does either one?

5. (A little bit subtle). Suppose that $f_1 = \Theta(g)$ and $f_2 = O(g)$. Can we say that $f_1 + f_2 = O(g)$? Can we say that $f_1 + f_2 = \Theta(g)$?