A Simple Introduction To Complexity Theory

Tom Gannon

November 21, 2016

1 What Is An Algorithm?

It's more of a philosophical question than anything, but for our purposes, an algorithm is a series of steps that a computer can perform. Another way to think of it is a series of steps you could write out and give to someone who had a calculator on them, but didn't know much math themselves.

2 Examples

2.1 Dividing an integer n>2 by 2

In this hypothetical example, you know how to do multiplication but not division. (It's not really an algorithm people use, it's just to illustrate the point.)

Here's one way to do it. Let's let q (think quotient) be an integer. If n $>$ 2q, then overwrite q with q+1. If n$<$2q, then return q.

Here's one of the most obvious examples that a computer can solve–Given a sorted list of length n, return the list in sorted order.

2.2 "Every Possibility Sort"

Here's one way you can sort a list of elements.

Algorithm: Consider every possible ordering of the list (for the sake of time, we won't go into how to do this, although if you know how to program it's not hard to write). and for each possible ordering, test if the list is sorted.

That seems like a pretty naive way to sort something though, maybe there's something better:

2.3 "Bubble Sort"

First step: compare the whole line, and use "swap" to get the largest element at the last.

After this, the last element is largest. (we know nothing else about the list).

Next step: Look at the first $n-1$ elements and get the next largest on the right.

Next next step: look at the first $n-2$ elements and get the next largest on the right.

...

Finally, you'll have two elements left. Swap them if necessary.

3 Speed of Algorithms

So the question is, which one of these is faster? Intrinsically it seems like the second one is faster. But suppose we're given the list 2,1 and the first thing that the "Every Possible Sort" algorithm checks is the reverse of the list. Then, (without getting too much into the CS detail), the "every possible list" algorithm takes "1," and the bubble sort takes "2." So how do we really say which is better?

Loosely, we want our "classes of algorithms" to be considered in the long run:

1. "Tail doesn't matter (think sequences)" Meaning that $n^3 - 1$ should be considered larger than $n^2 + 1000000$

2. Constants don't matter as n gets large. $2n^5$ and n^5 are the same class.

Definition 3.1. *We say an algorithm that takes $g(n)$ steps with input size n is $O(f(n))$ if there is an n_0 such that $n > n_0$ implies $g(n) \leq cf(n)$ for some constant $c > 0$.*

(Disclaimer: There are two ways to put algorithms into speed classes like this that people use, "worst case" and "average case." Worst case is easier to define, so we'll use it here.)

Example 3.2. *(Our sorting algorithms) The bubblesort algorithm earlier can do no worse than $n\frac{n+1}{2}$ moves. So the bubble sort algorithm is $O(n^2)$ with constant 1 and $n_0 = 2$. The "every option sort" is $O(2^n)$, and (possibly more importantly) is NOT $O(n^2)$. (Hand waive this)*

Example 3.3. *(What we've defined is just an upper bound)*
The bubblesort algorithm is also $O(n^100)$ with a constant $c = 99$ and $n_0 = 4$ in our definition. Our "every option sort" is also $O(2^n + n^n + 15)$.

Even though we don't *have* to, we tend to just associate it with the leading term.

Proposition 3.4. *If an algorithm has takes $a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ time, it is $O(x^k)$ if $k \geq n$. In particular, it is $O(x^n)$.*

Now we'll talk about some faster sorting algorithms:

Example 3.5. *(QuickSort) This example uses **recursion**, which is the computer science analog of induction. Here's how it goes. If you're given a list that's length 0 or 1 (base case of induction), the list is already sorted. Else, pick something in the list and call it the "pivot" (*you should aim for something in the middle). Then using swap, put everything that's less than pivot to the left of pivot on the list, and everything that's larger than pivot to the right of pivot. Then do quicksort on the smaller list. This sort of looks like (draw the picture):*

Proposition 3.6. *On the "average case", Quicksort is $O(nlogn)$.*

Proof. (Going to be a little fuzzy–will point out where). Let $g(n)$ = "number of steps taken given a list of n elements. Then since we picked something "in the middle" $g(n) = g(\frac{n}{2}) + g(\frac{n}{2}) = 2g(\frac{n}{2})$. So we have the identities:

$$g(n) = 2g(n/2)$$
$$2g(n/2) = 4g(n/4)$$

...

$$2^{n-1}g(2) = 2^n g(1) = n$$

Thus $g(n) = n$ for all n. (We're sort of assuming n is a power of two–round it up to make it more rigorous). We'll then have to do for our quicksort:
$g(n) + 2g(n/2) + 4g(n/4) + ... + ng(1) = log_2(n)n$ operations. □

Remark 3.7. *This is actually the best sorting algorithm, given a fixed amount of memory. There is an algorithm called bucketsort (give a brief explanation).*